Mengenal dan Mengeksploitasi Buffer Overflow di Sistem Linux x86 32-bit

Oleh: Akhdan Arif Prayoga

Apa yang akan kita bahas?

Apa itu Buffer Overflow
Dampak Buffer Overflow
Memory Management
Stack
Instruction
Register Memory (EIP,ESP,EBP)
Stack Frame
Return Address
GDB PEDA
Praktek Mengganti Nilai Variable Untuk Mengubah Alur Program
Praktek Memanggil Fungsi yang Tidak Dipanggil di dalam Program
Praktek Menjalankan Shellcode di Dalam Stack untuk Membuka Shell
Buffer Overflow Mitigations di Linux
Praktek Bypass NX Menggunakan Teknik Return to Libc untuk Membuka Shell
Komponen Return to Libc

Apa itu Buffer Overflow?

Buffer overflow adalah kondisi ketika program menulis data melebihi batas ruang yang disediakan (buffer), seperti menuangkan air ke gelas yang sudah penuh, sehingga data bisa tumpah dan menimpa bagian memori lain — menyebabkan crash atau celah keamanan.

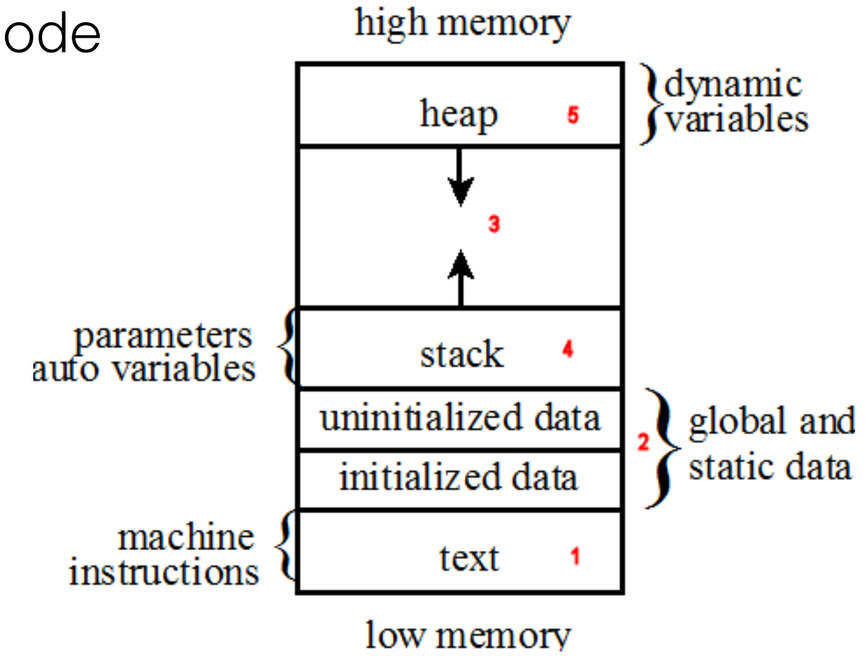
Dampak Buffer Overflow

- Mengubah nilai variabel lain
- Menjalankan fungsi lain atau fungsi yang tidak dijalankan
- Mendapatkan shell akses (shellcode)
- Melewati login atau validasi jika struktur memori rusak.
- Overflow ke arah luar buffer bisa bocorkan isi memory lain (leak).
- Privilege Escalation

Memory Management

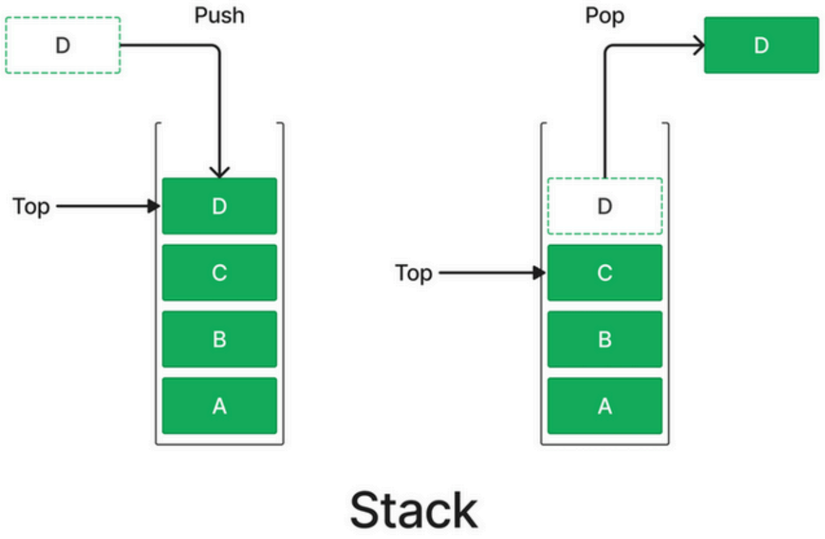
Di CPU ada beberapa bagian memory:

- Stack -> Bagian memory untuk menyimpan variable lokal
- Heap -> Bagian memory dinamik yang dapat diatur sesuka hati
- Static/Global -> Bagian memory untuk menyimpan variable global
- Text -> Bagian memory untuk menyimpan source code



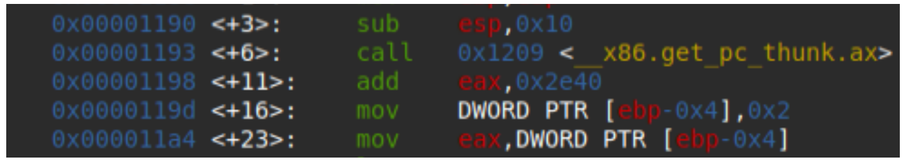
Stack?

Stack bisa diibaratkan seperti tumpukan piring, di mana piring terakhir yang ditaruh di atas akan jadi yang pertama diambil; prinsip ini disebut LIFO (Last In, First Out) dan digunakan untuk menyimpan data seperti return address dan variabel saat fungsi berjalan.



Instruction

Instruction bisa diibaratkan sebagai langkah atau perintah tunggal yang diberikan ke CPU untuk melakukan satu tugas spesifik, seperti menghitung, memindahkan data, atau melompat ke bagian lain dalam program.



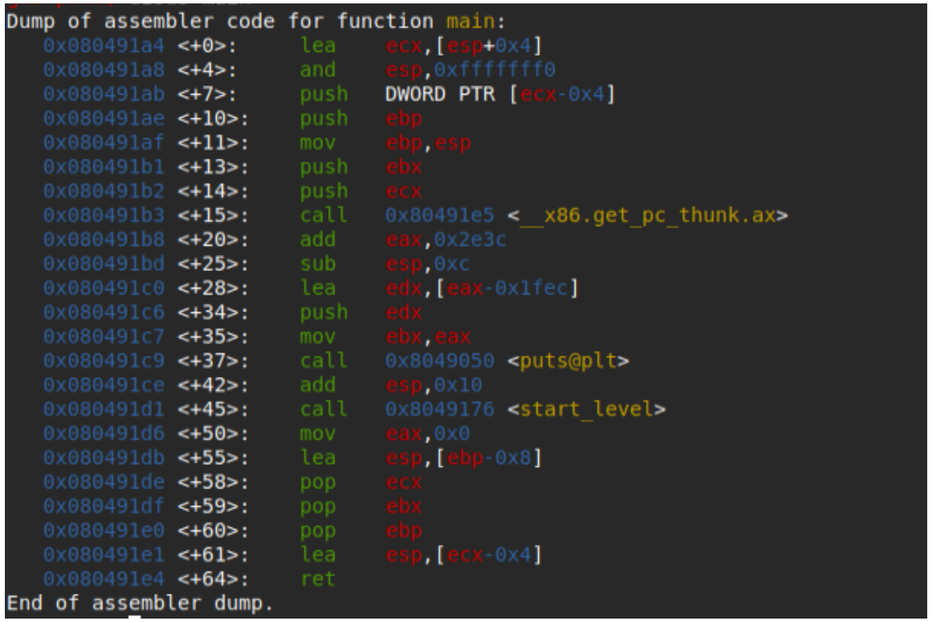
Register Memory

Register adalah memori kecil dan sangat cepat di dalam CPU yang digunakan untuk menyimpan data sementara seperti hasil perhitungan atau alamat, dan sangat penting dalam menjalankan instruksi program dengan efisien. Dalam analisis buffer overflow, yang perlu diperhatikan adalah register memory, antara lain:

- **EIP**
- **ESP**
- **EBP**

EIP (Instruction Pointer)

Instruction Pointer adalah register yang menyimpan alamat instruksi berikutnya yang akan dijalankan oleh CPU—ibarat sutradara yang menentukan adegan mana yang harus dijalankan selanjutnya, dan sering jadi target eksploitasi dalam buffer overflow.



ESP (Stack Pointer)

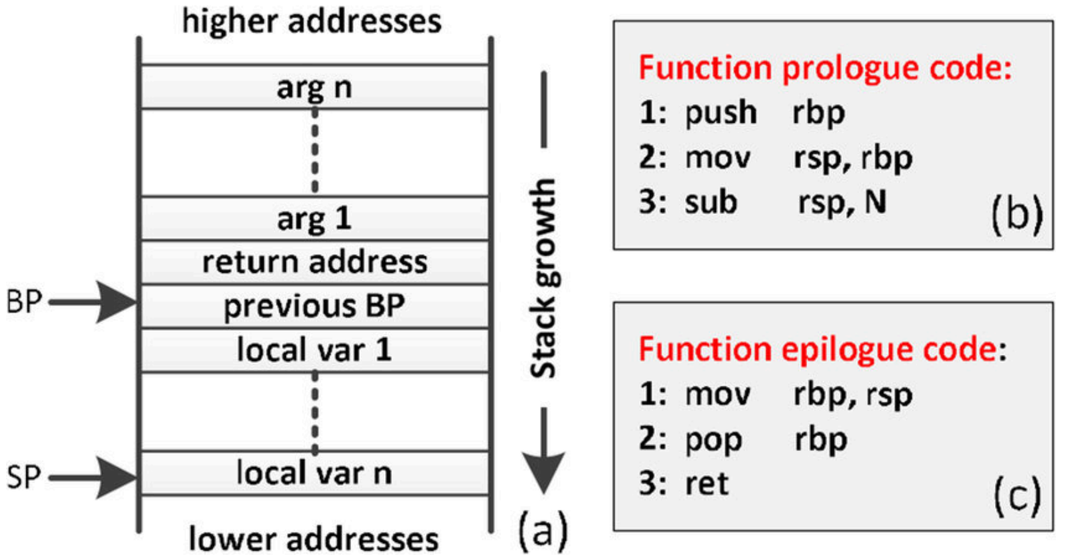
Stack Pointer adalah register yang menunjuk ke posisi teratas di stack—ibarat tangan yang selalu menunjuk piring paling atas dalam tumpukan, menentukan di mana data terakhir disimpan atau diambil.

EBP (Base Pointer)

Base Pointer adalah register yang menunjuk ke dasar stack—ibarat lantai dasar sebuah ruang kerja, tempat semua variabel dan parameter fungsi tersusun rapi untuk diakses selama eksekusi.

Stack Frame?

Stack frame bisa diibaratkan seperti rumah bagi sebuah fungsi—di dalamnya tersimpan semua yang dibutuhkan fungsi untuk bekerja, seperti variabel lokal, parameter, dan alamat kembali; jika terjadi stack overflow, bagian penting dari “rumah” itu bisa rusak, sehingga alur program bisa kacau atau diambil alih.



Return Address

Return Address adalah alamat di stack yang menunjukkan ke mana program harus kembali setelah sebuah fungsi selesai—ibarat bookmark yang menandai posisi sebelum “melompat” ke fungsi lain, dan sering jadi target dalam serangan buffer overflow, karena return address menentukan ke mana program kembali setelah fungsi selesai, penyerang bisa menimpanya untuk mengarahkan eksekusi ke kode berbahaya yang mereka siapkan.

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

char *gets(char *);

void start_level() {
  char buffer[128];
  gets(buffer);
}

int main(int argc, char **argv) {
  printf("%s\n", BANNER);
  start_level();
}
```

Stack Frame Main

```
Dump of assembler code for function main:
   0x080491a4 <+0>:     lea    ecx,[esp+0x4]
   0x080491a8 <+4>:     and    esp,0xfffffff0
   0x080491ab <+7>:     push   DWORD PTR [ecx-0x4]
   0x080491ae <+10>:    push   ebp
   0x080491af <+11>:    mov    ebp,esp
   0x080491b1 <+13>:    push   ebx
   0x080491b2 <+14>:    push   ecx
   0x080491b3 <+15>:    call   0x80491e5 <__x86.get_pc_thunk.ax>
   0x080491b8 <+20>:    add    eax,0x2e3c
   0x080491bd <+25>:    sub    esp,0xc
   0x080491c0 <+28>:    lea    edx,[eax-0x1fec]
   0x080491c6 <+34>:    push   edx
   0x080491c7 <+35>:    mov    ebx,eax
   0x080491c9 <+37>:    call   0x8049050 <puts@plt>
   0x080491ce <+42>:    add    esp,0x10
   0x080491d1 <+45>:    call   0x8049176 <start_level>
   0x080491d6 <+50>:    mov    eax,0x0
   0x080491db <+55>:    lea    esp,[ebp-0x8]
   0x080491de <+58>:    pop    ecx
   0x080491df <+59>:    pop    ebx
   0x080491e0 <+60>:    pop    ebp
   0x080491e1 <+61>:    lea    esp,[ecx-0x4]
   0x080491e4 <+64>:    ret
End of assembler dump.
```

Stack Frame Start Level

```
Dump of assembler code for function start_level:
   0x08049176 <+0>:     push   ebp
   0x08049177 <+1>:     mov    ebp,esp
   0x08049179 <+3>:     push   ebx
   0x0804917a <+4>:     sub    esp,0x84
   0x08049180 <+10>:    call   0x80491e5 <__x86.get_pc_thunk.ax>
   0x08049185 <+15>:    add    eax,0x2e6f
   0x0804918a <+20>:    sub    esp,0xc
   0x0804918d <+23>:    lea    edx,[ebp-0x88]
   0x08049193 <+29>:    push   edx
   0x08049194 <+30>:    mov    ebx,eax
   0x08049196 <+32>:    call   0x8049040 <gets@plt>
   0x0804919b <+37>:    add    esp,0x10
   0x0804919e <+40>:    nop
   0x0804919f <+41>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x080491a2 <+44>:    leave
   0x080491a3 <+45>:    ret
End of assembler dump.
```

Singkatnya...

- Fungsi punya buffer lokal (di stack)
- Program tidak mengecek panjang input
- User input terlalu panjang → menimpa stack frame
- Return address tertimpa alamat palsu
- Fungsi selesai → CPU ambil return address → EIP diarahkan ke alamat palsu
- Kode asing/shellcode dijalankan → attacker menang

GDB PEDA

GDB (GNU Debugger) adalah alat debugging powerful untuk program C/C++ di Linux, dan PEDA (Python Exploit Development Assistance) adalah ekstensi yang memperkaya GDB dengan fitur tambahan seperti visualisasi stack, register, dan memory secara otomatis, serta perintah eksploitasi yang memudahkan analisis kerentanan seperti buffer overflow dan format string.

https://github.com/longld/peda
https://github.com/Kuro-orzz/peda

Praktek Mengganti Nilai Variable Untuk Mengubah Alur Program

https://exploit.education/phoenix/stack-zero/

Praktek Memanggil Fungsi yang Tidak Dipanggil di dalam Program

https://exploit.education/phoenix/stack-four/

Alur Eksploitasi:

1. start_level() dipanggil.
2. gets() menerima input.
3. Input > 64 byte ⇒ overflow ⇒ timpa return address.
4. Fungsi selesai → stack frame dibersihkan → program lompat ke alamat palsu (yang di set ke `complete_level()`).
5. Jika sukses, `complete_level()` akan dijalankan dan mencetak pesan berhasil.

Praktek Menjalankan Shellcode di Dalam Stack untuk Membuka Shell

https://exploit.education/phoenix/stack-five/

Buffer Overflow Mitigations di Linux

Address space randomization (ASLR)

Mengacak lokasi stack, heap, dan library untuk menyulitkan prediksi alamat oleh attacker.

Data execution prevention (NX)

Menandai stack/heap sebagai non-eksekusi untuk mencegah shellcode dijalankan di sana.

RELRO

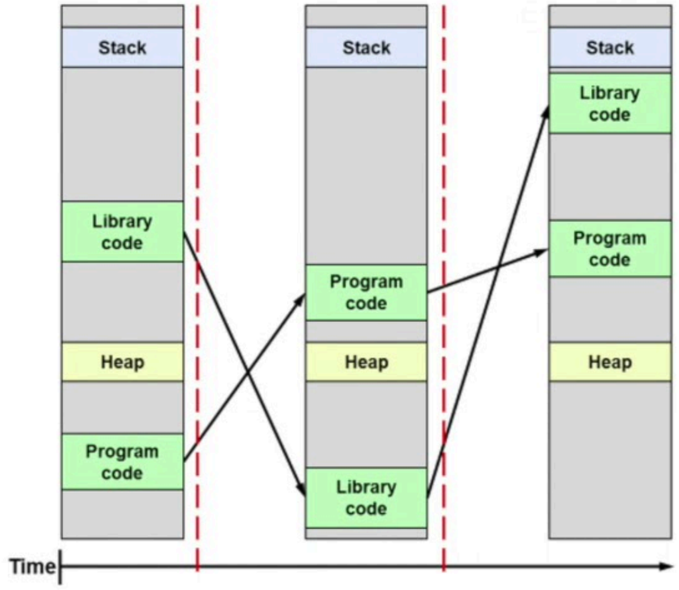
Membuat GOT (Global Offset Table) menjadi read-only agar tidak dapat dioverwrite oleh attacker.

Stack Canary

Menyisipkan nilai rahasia di stack untuk mendeteksi buffer overflow sebelum return address dirusak.

Address space randomization (ASLR)

ASLR: Randomize Addresses per Each Execution




```
$ ./aslr-check
Executing myself for five times
$ Address of stack: 0xbf943a70 heap 0x9913008 libc 0xb7e26670
Address of stack: 0xbfc76330 heap 0x973b008 libc 0xb7dd7670
Address of stack: 0xbfedeea0 heap 0x9716008 libc 0xb7e31670
Address of stack: 0xbf93d7d0 heap 0x9601008 libc 0xb7dcc670
Address of stack: 0xbfa9dd60 heap 0x9f7e008 libc 0xb7dbc670
```

Oregon State University

Praktek Bypass NX Menggunakan Teknik Return to Libc untuk Membuka Shell

https://exploit.education/phoenix/stack-five/

Komponen Return to Libc

system()

- Fungsi system() dari libc akan menjalankan perintah shell
- Jadi kalau kamu memanggil system("/bin/sh"), kamu akan mendapat shell interaktif.

exit()

- Setelah system() selesai, program akan tetap melanjutkan eksekusi.
- Jadi kamu arahkan return selanjutnya ke exit() agar program keluar

“/bin/sh”

- Dibutuhkan sebagai argumen untuk system() untuk membuka shell.

Sekian, terimakasih.