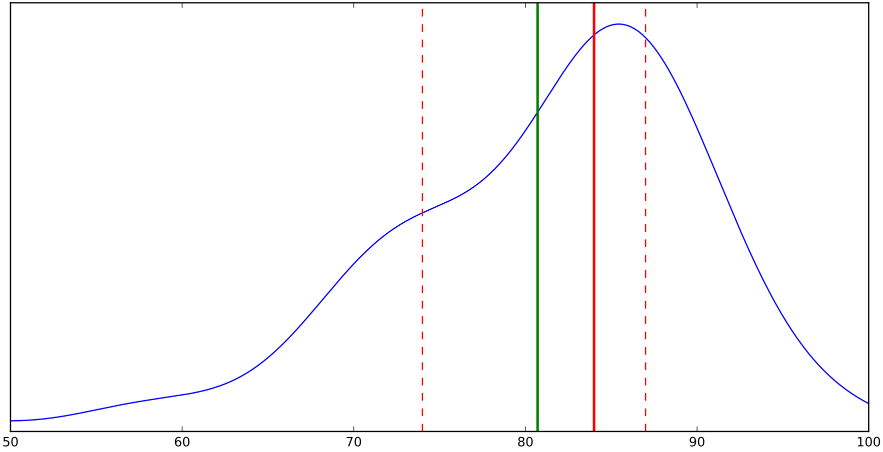
More Buffer Overflows

midterm 1



kernel density plot; red lines: 25/50/75th perecentile; green line: mean

last time

stack smashing

particular exploit technique for buffer overflows

- buffer overflow = out-of-bounds access to array

condition: buffer on the stack

two steps:

- insert machine code
- overwrite return address to point there

stack smashing: the tricky parts

construct machine code that works in any executable
- same tricks as writing relocatable virus code
- usual idea: just execute shell (command prompt)

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

stack smashing: the tricky parts

construct machine code that works in any executable

- same tricks as writing relocatable virus code
- usual idea: just execute shell (command prompt)

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

machine code that works anywhere

need relocatable machine code

relative addressing internally

absolute addressing of program

stack smashing: the tricky parts

construct machine code that works in any executable

- same tricks as writing relocatable virus code
- usual idea: just execute shell (command prompt)

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

valid input?

common restrictions: no `0` bytes, no newlines

machine code is flexible enough, but tricky

example: `mov $0x100, %rax` has 0s in encoding of `0x100`

```
xor %eax, %eax
mov $0x100, %al
```

stack smashing: the tricky parts

construct machine code that works in any executable

- same tricks as writing relocatable virus code
- usual idea: just execute shell (command prompt)

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

location of return address

easiest part, but ...

depends on what compiler does

- variable number of saved registers
- ...

read assembly?

stack smashing: the tricky parts

construct machine code that works in any executable

- same tricks as writing relocatable virus code
- usual idea: just execute shell (command prompt)

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

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
```

address space layout randomization

vary the location of things in memory

including the stack

designed to make exploiting memory errors harder

will talk more about later

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
0x7fffffffe064
$ ./stackloc.exe
0x7fffffffe064
$ ./stackloc.exe
0x7fffffffe064
```

finding stack location

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

set breakpoint at relevant location

```
b functionName
b *0x12345678
```

(by address)

output %rsp

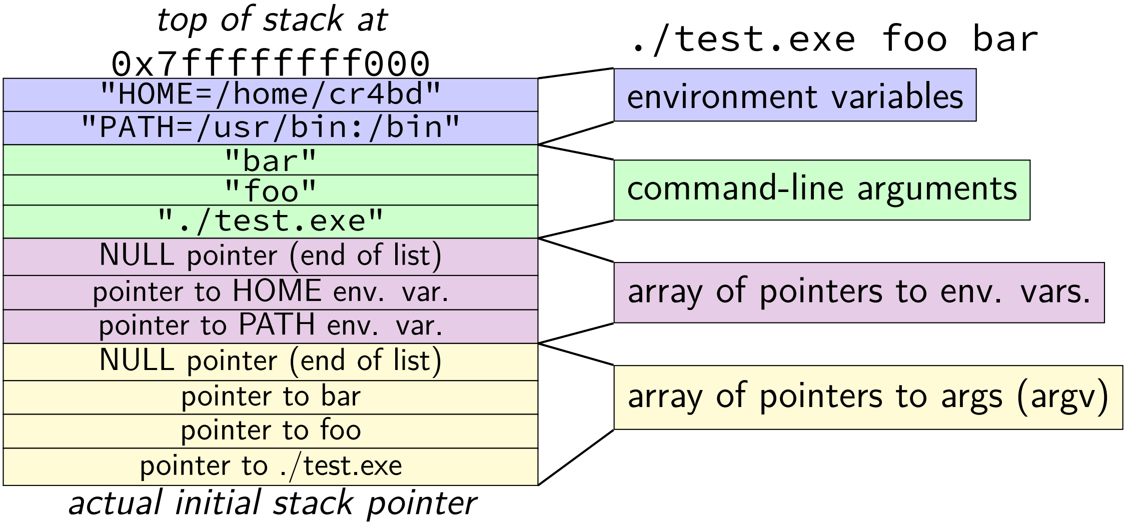
```
p $rsp
info registers
```

stack location? (take 2)

```
$ ./stackloc.exe
0x7fffffffe064
$ gdb ./stackloc.exe
...
(gdb) break main
Breakpoint 1 at 0x4005b6
(gdb) run
Starting program: /home/cr4bd/spring2017/cs4630/slides/20170307/stackloc.exe

Breakpoint 1, 0x00000000004005b6 in main ()
(gdb) p $rsp
$1 = (void *) 0x7fffffffdff8
(gdb) continue
0x7fffffffdfe4
[Inferior 1 (process 15441) exited normally]
(gdb)
```

Linux, initial stack



on using GDB

cheat sheet on website

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"

debugging demo

stopping stack smashing?

how can you stop stack smashing?

stopping stack smashing?

how can you stop stack smashing?

stop overrun — bounds-checking

stop return to attacker code

stop execution of attacker code

exploit mitigations

idea: turn vulnerablity to something less bad

e.g. crash instead of machine code execution

many of these targetted at buffer overflows

mitigation agenda

we will look briefly at one mitigation — stack canaries

then look at exploits that don't care about it

then look at more flexible mitigations

then look at more flexible exploits

mitigation priorities

effective? does it actually stop the attacker?

fast? how much does it hurt performance?

generic? does it require a recompile? rewriting software?

stopping stack smashing?

how can you stop stack smashing?

stop overrun — bounds-checking

stop return to attacker code

stop execution of attacker code

recall: RE

```
/* copy value from thread-local storage */
    mov %fs:0x28, %rax
/* ... on to stack, before return address */
    mov %rax, 0x18(%rsp)
    ...
    ...
    ...
/* copy value from stack */
    mov 0x18(%rsp), %rdi
/* xor with value in thread-local storage */
    xor %fs:0x28, %rdi
/* if result non-zero, do not return */
    jne call_stack_chk_fail
    add $0x28, %rsp
    ret
call_stack_chk_fail:
    call __stack_chk_fail
```

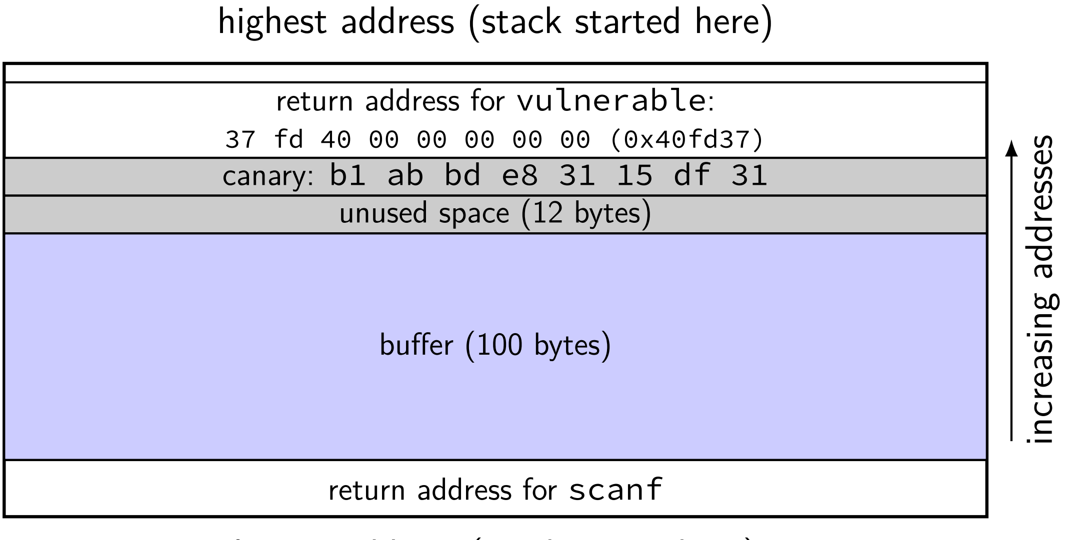
recall: RE

```
/* copy value from thread-local storage */
    mov %fs:0x28, %rax
/* ... on to stack, before return address */
    mov %rax, 0x18(%rsp)
    ...
    ...
    ...
/* copy value from stack */
    mov 0x18(%rsp), %rdi
/* xor with value in thread-local storage */
    xor %fs:0x28, %rdi
/* if result non-zero, do not return */
    jne call_stack_chk_fail
    add $0x28, %rsp
    ret
call_stack_chk_fail:
    call __stack_chk_fail
```

recall: RE

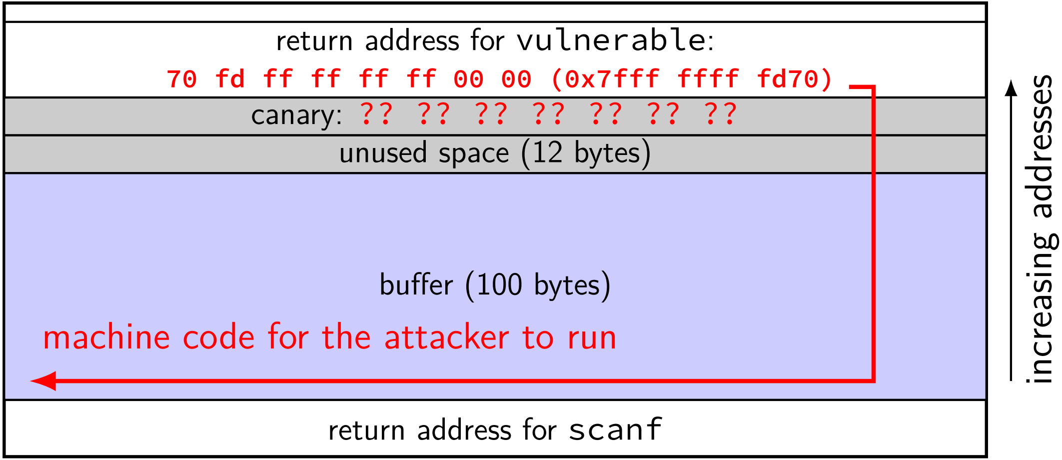
```
/* copy value from thread-local storage */
    mov %fs:0x28, %rax
/* ... on to stack, before return address */
    mov %rax, 0x18(%rsp)
    ...
    ...
    ...
/* copy value from stack */
    mov 0x18(%rsp), %rdi
/* xor with value in thread-local storage */
    xor %fs:0x28, %rdi
/* if result non-zero, do not return */
    jne call_stack_chk_fail
    add $0x28, %rsp
    ret
call_stack_chk_fail:
    call __stack_chk_fail
```

stack canary



stack canary

highest address (stack started here)



stack canary — action

```
mov %fs:0x28, %rdi //    0xb1 ab bd e8 31 15 df 31 XOR
xor %rdi, 0x112(%rsp) // 0x?? ?? ?? ?? ?? ?? ?? ??
                    // =  0x?? ?? ?? ?? ?? ?? ?? ??
jne call_stack_check_file // jump if != 0
...
call_stack_chk_fail:
  call __stack_chk_fail
  ...
__stack_chk_fail:
    /* print "*** stack smashing detected message" and exit */
```

stack canary hopes

overwrite return address $\implies$ overwrite canary

canary is secret

stack canary hopes

overwrite return address $\Longrightarrow$ overwrite canary

buffer overrun, not some other memory error

canary is secret

stack canary hopes

overwrite return address $\implies$ overwrite canary

- buffer overrun, not some other memory error

canary is secret

- chosen at random
- program doesn't output it

information disclosure (1)

```
void process() {
    char buffer[8] = "\0\0\0\0\0\0\0\0";
    char c = '␣';
    for (int i = 0; c != '\n' && i < 8; ++i) {
        c = getchar();
        buffer[i] = c;
    }
    printf("You␣input␣%s\n", buffer);
}
```

input `aaaaaaaa`

output `You input aaaaaaaa`*(whatever was on stack)*

information disclosure (2)

```
struct foo {
    char buffer[8];
    long *numbers;
};

void process(struct foo* thing) {
    ...
    scanf("%s", thing->buffer);
    ...
    printf("first␣number:␣%ld\n", thing->numbers[0]);
}
```

input: aaaaaaaa*(address of canary)*

address on stack *or* where canary is read from in thread-local storage

good choices of canary

random — guessing should not be practical

- not always — sometimes static or only 2^{15} possible

GNU libc: canary contains:

leading `\0` (string terminator)

- `printf %s` won't print it

a newline

- read line functions can't input it

`\xFF`

- hard to input?

stack canaries implementation

"StackGuard" — 1998 paper proposing strategy

GCC: command-line options

- `-fstack-protector`
- `-fstack-protector-strong`
- `-fstack-protector-all`
- one of these often default
- three differ in how many functions are 'protected'

Microsoft C/C++ compiler: `/GS`

- on by default

stack canary overheads

less than 1% runtime if added to “risky” functions

- functions with character arrays, etc.

large overhead if added to all functions

- StackGuard paper: 5–20%?

similar space overheads

(for typical applications)

- could be much worse: tons of ‘risky’ function calls

stack canaries pro/con

pro: no change to calling convention

pro: recompile only — no extra work

con: can't protect existing executable/library files (without recompile)

con: doesn't protect against many ways of exploiting buffer overflows

con: vulnerable to information leak

stack canary summary

stack canary — simplest of many mitigations

key idea: detect corruption of return address

assumption: if return address changed, so is adjacent token

assumption: attacker can't learn true value of token

- often possible with memory bug

later: workarounds to break these assumptions

more migitations?

in future lectures

after we talk about other ways of exploiting buffer overflows

beyond return addresses

overwriting return address to point to code

- “stack smashing”

not the only thing on the stack

- easier to overwrite something else?

some buffers are not be on the stack

- is something “interesting” next to them in memory?

mitigation priorities

effective? does it actually stop the attacker?

fast? how much does it hurt performance?

generic? does it r … e?

recurring theme: stop stack smashing,
but not other buffer overflows

recall: simpler overflow

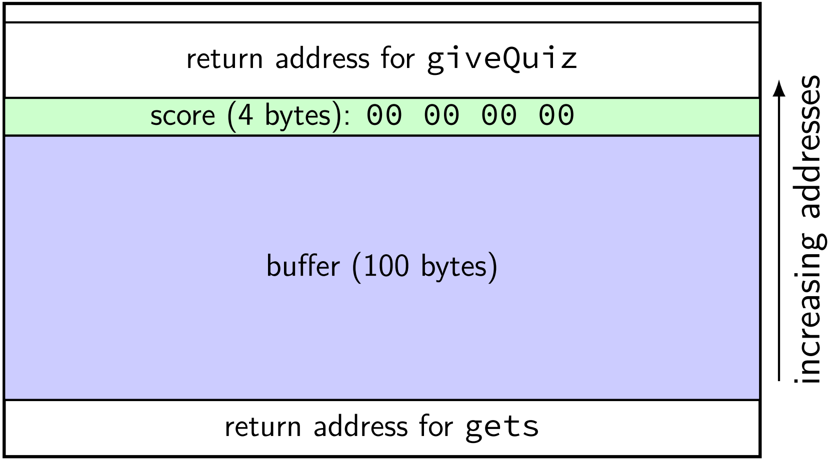
```
struct QuizQuestion questions[NUM_QUESTIONS];
int giveQuiz() {
    int score = 0;
    char buffer[100];
    for (int i = 0; i < NUM_QUESTIONS; ++i) {
        gets(buffer);
        if (checkAnswer(buffer, &questions[i])) {
            score += 1;
        }
    }
    return score;
}
```

recall: simpler overflow

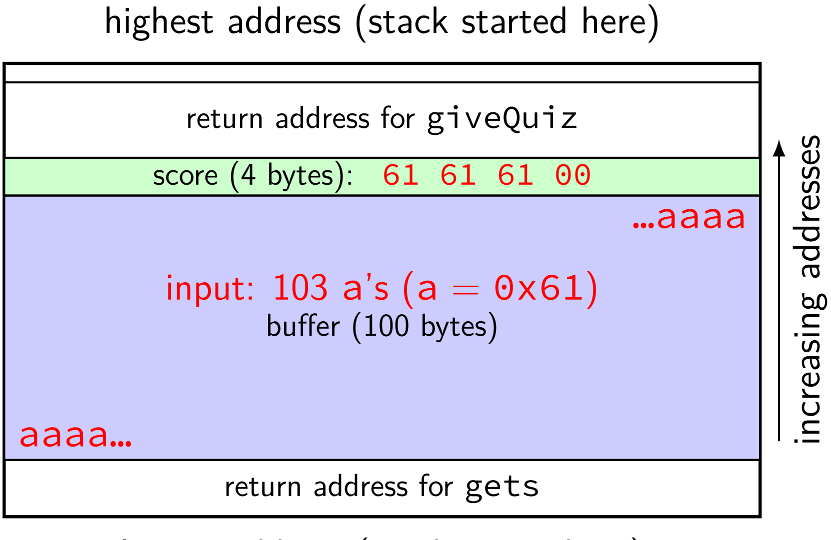
```
struct QuizQuestion questions[NUM_QUESTIONS];
int giveQuiz() {
    int score = 0;
    char buffer[100];
    for (int i = 0; i < NUM_QUESTIONS; ++i) {
        gets(buffer);
        if (checkAnswer(buffer, &questions[i])) {
            score += 1;
        }
    }
    return score;
}
```

recall: simpler overflow: stack

highest address (stack started here)



recall: simpler overflow: stack



but don’t you have to get lucky?

simple overflow seems contrived

stack smashing had big advantages:

- every buffer on the stack was a problem
- easy to adapt exploit — recall debugger exercise

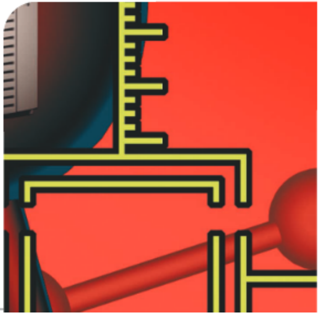
some more exploit techniques

- not as generic as stack smashing
- but collectively close

article on topic

Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns

This article describes three powerful general-purpose families of exploits for buffer overruns: arc injection, pointer subterfuge, and heap smashing. These new techniques go beyond the traditional "stack smashing" attack and invalidate traditional assumptions about buffer overruns.



techniques from Pincus and Baker

arc injection AKA return-oriented programming

    more detail (+ assignment) later in semester

overwriting data pointers

overwriting function pointers

overwriting pointers to function pointers

(on heap) overwriting `malloc`'s data structures

other buffer overflows?

old example: data on stack

techniques from Pincus and Baker

arc injection AKA return-oriented programming

more detail (+ assignment) later in semester

overwriting data pointers

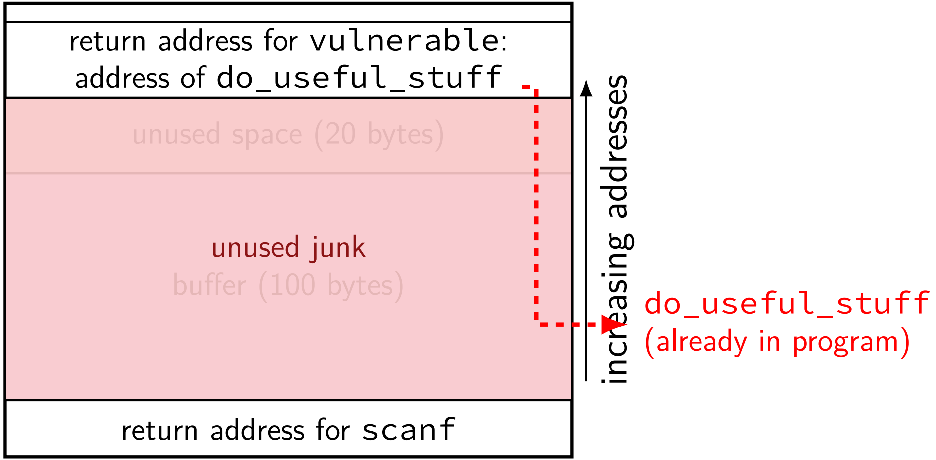
overwriting function pointers

overwriting pointers to function pointers

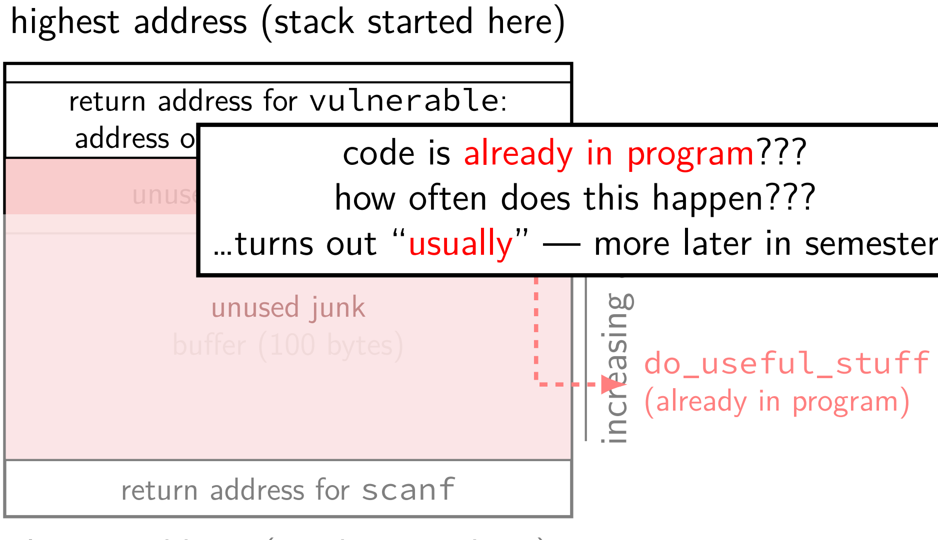
(on heap) overwriting `malloc`'s data structures

return-to-somewhere

highest address (stack started here)



return-to-somewhere



code is already in program???
how often does this happen???
...turns out "usually" — more later in semester

techniques from Pincus and Baker

arc injection AKA return-oriented programming

more detail (+ assignment) later in semester

overwriting data pointers

overwriting function pointers

overwriting pointers to function pointers

(on heap) overwriting `malloc`'s data structures

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

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

how can attacker exploit this?

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

how can attacker exploit this?

overwrite return address directly

overwrite other function pointer?

overwrite existing machine code (insert jump?)

overwrite another data pointer — copy more?

arbitrary memory write

bunch of scenarios that lead to single arbitrary memory write

how can attacker exploit this?

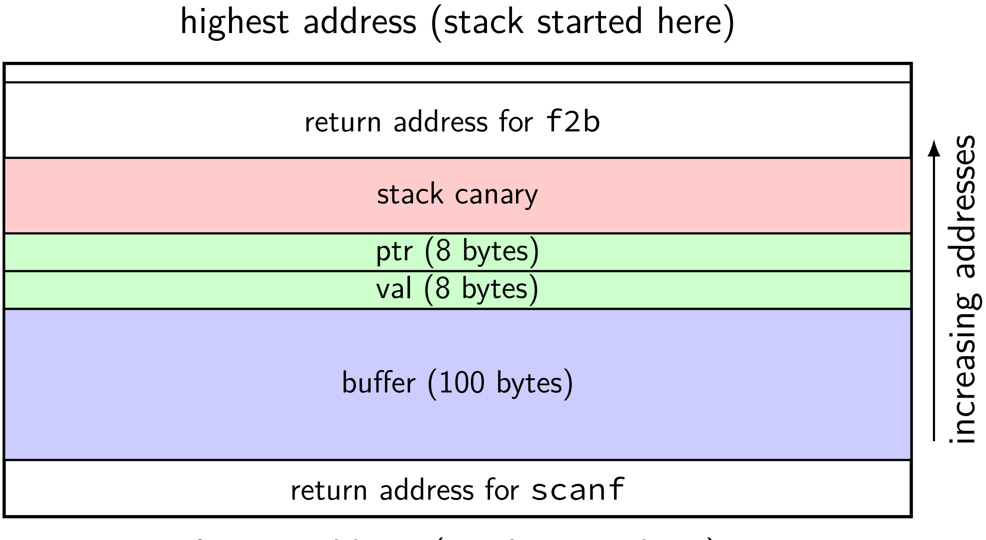
overwrite return address directly

overwrite other function pointer?

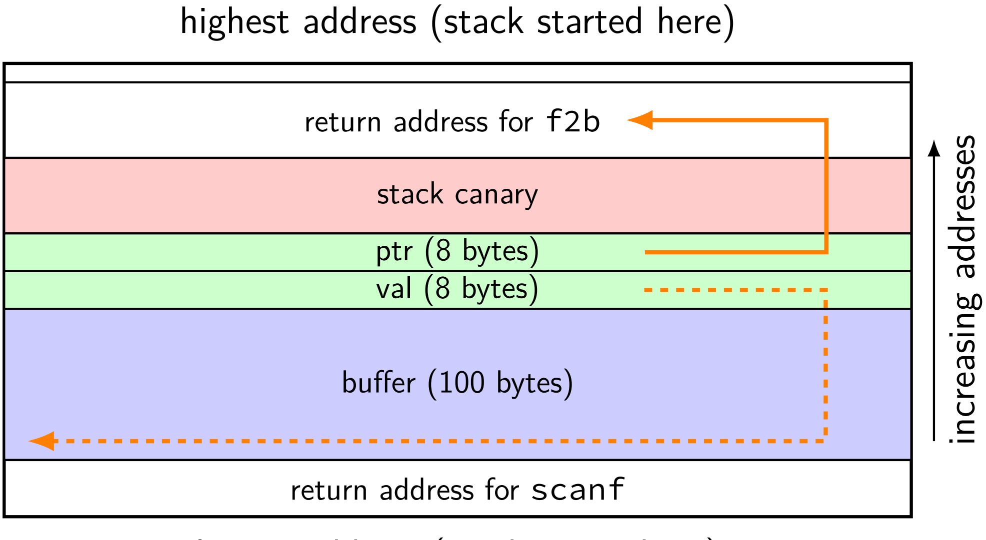
overwrite existing machine code (insert jump?)

overwrite another data pointer — copy more?

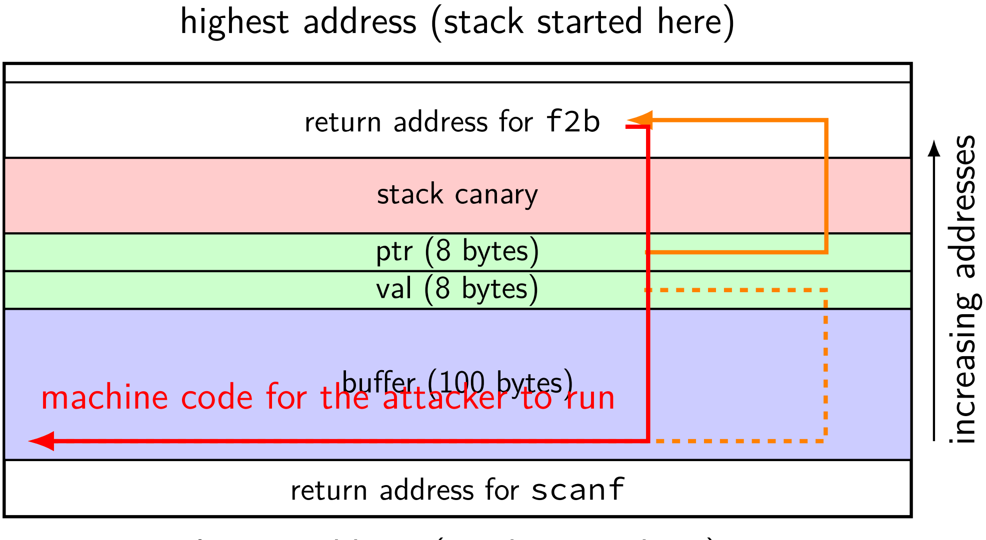
skipping the canary



skipping the canary



skipping the canary



fragility

problem: need to know exact address of return address

discussed how stack location varies — this is tricky/unreliable