CSE127, Computer Security

Control flow vulnerabilities and buffer overflows

UC San Diego

Housekeeping

General course things to know

- PA1 due Thursday by **1/15** at 11:59
  - Hopefully by now you have set up the infrastructure needed for PA1, if not, no time like the present
  - ***Do not forget to submit your AI attestation!***
- Due **1/16** at 11:59
  - #FinAid Canvas quiz: https://canvas.ucsd.edu/courses/71475/quizzes/238979, reminder to do this!
- PA2 releases Friday at midnight, due **1/27** at 11:59

Previously on CSE127…

- We talked about C.I.A., threat modeling, and the adversarial mindset
- Question: How can we apply these concepts to application areas we care about? Like software, web, etc?

Today's lecture – Control flow vulnerabilities and buffer overflows

Learning Objectives

- Understand basic software exploits in programs
- Understand how buffer overflow vulnerabilities can be exploited, with specifics of x86-32bit and C
- Identify common buffer overflow vulnerability patterns in code and assess their impact
- Learn best practices for avoiding buffer overflow vulnerabilities during implementation

Software Security

When is a program considered secure?

- Definition: When it does exactly what it should.
  - *What are some issues with this definition?*

When is a program considered secure?

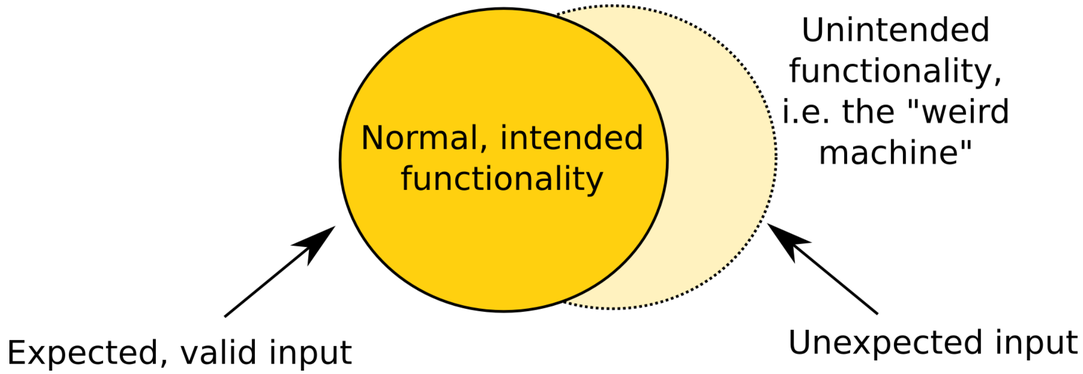
- Definition: When it does exactly what it should.
  - *What are some issues with this definition?*
- How do we know what a program is supposed to do?
  - Somebody tells us (so we have to trust them)
  - We write the code ourselves (but requires significant trust of *every aspect* of the system stack)
  - We have access to the formal specification (and we actually choose to read it)

When is a program considered secure?

- Better definition: A program is secure when it doesn't do **bad things**
- Much easier to specify a list of "bad" things, here are some examples:
  - Delete or corrupt important files
  - Crash the system
  - Send passwords over the Internet
  - Send threatening emails to Deepak about his podcast policy
- Unfortunately, software can be *mostly* good, but *occasionally* be made to do bad things — do we still consider it **secure?**

Weird machines

- Complex software almost *always* contained unintended functionality when the input changes
  - We call this artifact “weird”
- An **exploit** is a mechanism where an attacker can trigger unintended functionality in the system; e.g., “programming” the weird machine
- Security requires understanding both intended *and* unintended functionality in implementation
  - Developers could miss things
  - Attackers could be clever



https://en.wikipedia.org/wiki/Weird_machine

What is a software vulnerability?

- A bug in a software program that allows an unprivileged user capabilities that should be denied to them
  - ***What security property does this violate?***

What is a software vulnerability?

- A bug in a software program that allows an unprivileged user capabilities that should be denied to them
  - ***What security property does this violate?***
- Most classic and important vulnerabilities violate what's called "**control flow integrity**"

What is a software vulnerability?

- A bug in a software program that allows an unprivileged user capabilities that should be denied to them
  - ***What security property does this violate?***
- Most classic and important vulnerabilities violate what's called "**control flow integrity"**
  - Attacker can run **code of their choosing** on **your computer.**
- Usually involves violating *assumptions* of the programming language or underlying run-time system (including OS, hardware assumptions!)

What are the exploits we're discussing today?

- Our threat model
  - Victim code is **handling input** that comes across a security boundary. When might this happen in real life?

What are the exploits we're discussing today?

- Our threat model
  - Victim code is **handling input** that comes across a security boundary. When might this happen in real life?
    - PDF processing, word processing, web browsing… basically *all user-facing software must interact across a security boundary*

What are the exploits we're discussing today?

- Our threat model
  - Victim code is **handling input** that comes across a security boundary. When might this happen in real life?
    - PDF processing, word processing, web browsing… basically *all user-facing software must interact across a security boundary*
  - We want to protect integrity and confidentiality of internal data from being compromised by malicious users
  - Today's primary example: **buffer overflow**
    - Provide input that "overflows" the memory a program has allocated for it

Programs, Functions, Memory, Assembly

A rapid refresher

- What is a function?
- What is a stack?
- What is a call stack? (aka function stack?)
- What are CPU registers?
- What are some important, special registers used for handling control flow?

A rapid refresher

- What is a function?
  - Self contained block of code that performs a specific task… notably, **can be called by other functions**
- What is a stack?
- What is a call stack? (aka function stack?)
- What are CPU registers?
- What are some important, special registers used for handling control flow?

A rapid refresher

- What is a function?
- What is a stack?
  - Data structure that follows a LIFO strategy (last-in, first-out).
- What is a call stack? (aka function stack?)
- What are CPU registers?
- What are some important, special registers used for handling control flow?

A rapid refresher

- What is a function?
- What is a stack?
- What is a call stack? (aka function stack?)
  - Data structure that tracks active functions in a program to manage function calls, local variables, and return addresses
- What are CPU registers?
- What are some important, special registers used for handling control flow?

A rapid refresher

- What is a function?
- What is a stack?
- What is a call stack? (aka function stack?)
- What are CPU registers?
  - Processor-enabled fast storage for temporarily holding data, instructions, addresses, values, etc.
  - 6 “general-purpose” registers in x86: EAX, EBX, ECX, EDX, ESI, EDI
- What are some important, special registers used for handling control flow?

A rapid refresher

- What is a function?
- What is a stack?
- What is a call stack? (aka function stack?)
- What are CPU registers?
- What are some important, special registers used for handling control flow?
  - Program counter (EIP), Stack pointer (ESP), Frame (or base) pointer (EBP)
  - These keep track of function stacks and where the program should execute from

x86 refresher

- What do the following assembly instructions do? (Using AT&T syntax)
  - **`mov 0x34, %eax`**

x86 refresher

- What do the following assembly instructions do? (Using AT&T syntax)
  - **`mov 0x34, %eax`** (moves 0x34 into eax)
  - **`add 0x10, %eax`**

x86 refresher

- What do the following assembly instructions do? (Using AT&T syntax)
  - **`mov 0x34, %eax`** (moves 0x34 into eax)
  - **`add 0x10, %eax`** (adds 0x10 to eax)
  - **`mov %eax, %edx`**

x86 refresher

- What do the following assembly instructions do? (Using AT&T syntax)
  - **`mov 0x34, %eax`** (moves 0x34 into eax)
  - **`add 0x10, %eax`** (adds 0x10 to eax)
  - **`mov %eax, %edx`** (copies value of %eax into %edx)
  - **`push %eax`**

x86 refresher

- What do the following assembly instructions do? (Using AT&T syntax)
  - **`mov 0x34, %eax`** (moves 0x34 into eax)
  - **`add 0x10, %eax`** (adds 0x10 to eax)
  - **`mov %eax, %edx`** (copies value of %eax into %edx)
  - **`push %eax`** (pushes the value of %eax onto stack and updates %esp)
  - **`call $0x12345`**

x86 refresher

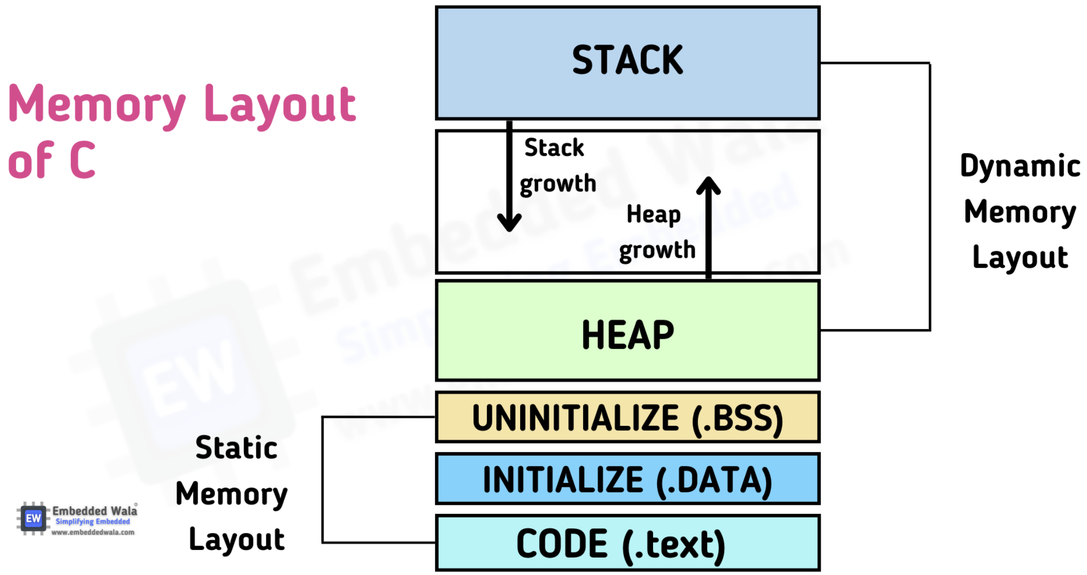
- What do the following assembly instructions do? (Using AT&T syntax)
  - **`mov 0x34, %eax`** (moves 0x34 into eax)
  - **`add 0x10, %eax`** (adds 0x10 to eax)
  - **`mov %eax, %edx`** (copies value of %eax into %edx)
  - **`push %eax`** (pushes the value of %eax onto stack and updates %esp)
  - **`call $0x12345`** (calls function at this address updating %eip)
  - **`jmp $0x12345`**

x86 refresher

- What do the following assembly instructions do? (Using AT&T syntax)
  - **`mov 0x34, %eax`** (moves 0x34 into eax)
  - **`add 0x10, %eax`** (adds 0x10 to eax)
  - **`mov %eax, %edx`** (copies value of %eax into %edx)
  - **`push %eax`** (pushes the value of %eax onto stack and updates %esp)
  - **`call $0x12345`** (calls function at this address updating %eip)
  - **`jmp $0x12345`** (moves instruction pointer to this address)

How is process memory laid out? (Linux 32-bit)

- Stack
  - Local variables, call stack
- Heap
  - Dynamically created variables; malloc, new, etc.
- Data segment (static variables, global variables
  - .data, .bss
- Text segment
  - Code has to also exist somewhere



C Arrays

- What is an array?

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

C Arrays

- What is an array?
  - Contiguous block of memory **of a fixed size.**

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

C Arrays

- What is an array?
  - Contiguous block of memory **of a fixed size.**
- How much memory is allocated on stack for these char buffers?

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

C Arrays

- What is an array?
  - Contiguous block of memory **of a fixed size.**
- How much memory is allocated on stack for these char buffers?
  - 20 bytes (8 bytes for buffer1, 12 for buffer2)

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

C Arrays

- What is an array?
  - Contiguous block of memory **of a fixed size.**
- How much memory is allocated on stack for these char buffers?
  - 20 bytes (8 bytes for buffer1, 12 for buffer2)
- Provocation: Will the program throw an error if you write beyond the buffer?

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

Understanding function calls — callers and callees

- Two functions: main (caller) and function (callee)
- Who is responsible for passing in function arguments?

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

Understanding function calls — callers and callees

- Two functions: main (caller) and function (callee)
- Who is responsible for passing in function arguments?
  - Caller

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

Understanding function calls — callers and callees

- Two functions: main (caller) and function (callee)
- Who is responsible for passing in function arguments?
  - Caller
- How does *function* know where to return to after it's done?

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

Understanding function calls — callers and callees

- Two functions: main (caller) and function (callee)
- Who is responsible for passing in function arguments?
  - Caller
- How does *function* know where to return to after it's done?
  - Caller pushes %eip as return address

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

Understanding function calls — callers and callees

- Two functions: main (caller) and function (callee)
- Who is responsible for passing in function arguments?
  - Caller
- How does *function* know where to return to after it's done?
  - Caller pushes %eip as return address
- Where is the return address stored?

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

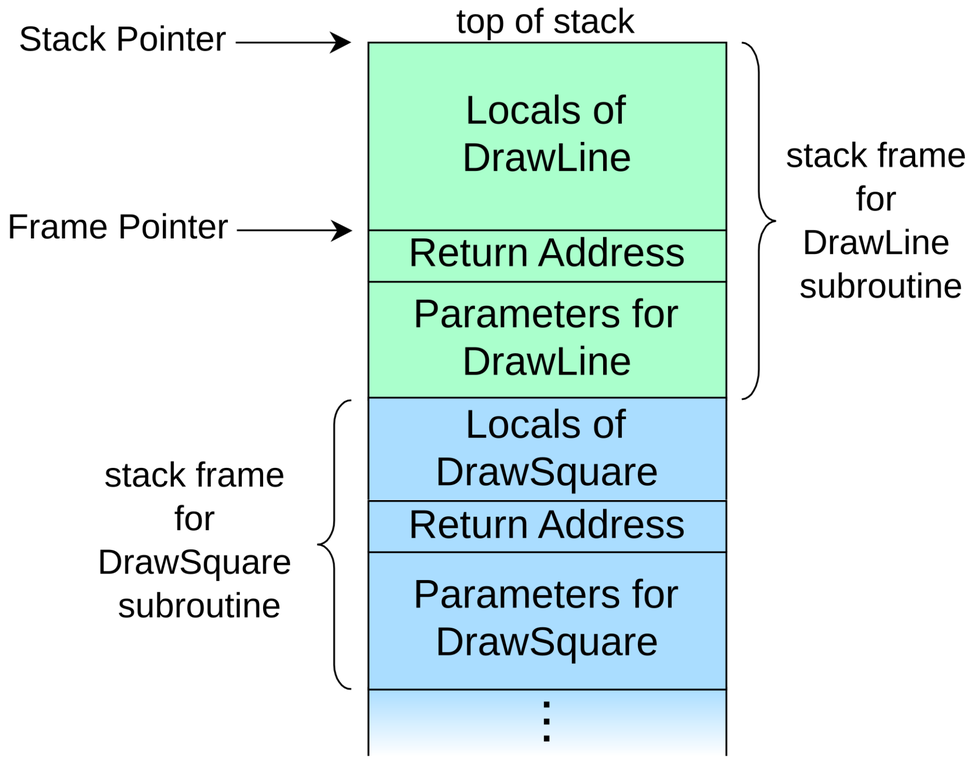
Understanding function calls — callers and callees

- Two functions: main (caller) and function (callee)
- Who is responsible for passing in function arguments?
  - Caller
- How does *function* know where to return to after it's done?
  - Caller pushes %eip as return address
- Where is the return address stored?
  - On the stack!

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

Function Stack Organization

- Stacks are divided into **frames**
  - Each *frame* stores locals + args to called functions
- **`call`** instruction will push the return address (e.g., where you were previously) onto the stack
- %esp points to the top of the stack
  - x86: stack grows down (from high to low addresses)
- %ebp points to the caller's frame on the stack (frame pointer)



https://en.wikipedia.org/wiki/Call_stack

Caller / Callee Responsibilities during `call`

- What are the responsibilities of the caller?
  - Pass arguments, save return address, call new function
- What is the responsibility of the callee?
  - Save old FP, set FP = SP, allocate stack space for local storage

Caller / Callee Responsibilities during `ret`

- What does the callee do when returning?
  - Pop local storage
  - Set SP = FP
  - Pop frame pointer
  - Pop return address and **ret**
- What does the caller do when returning?
  - Pop arguments and continue

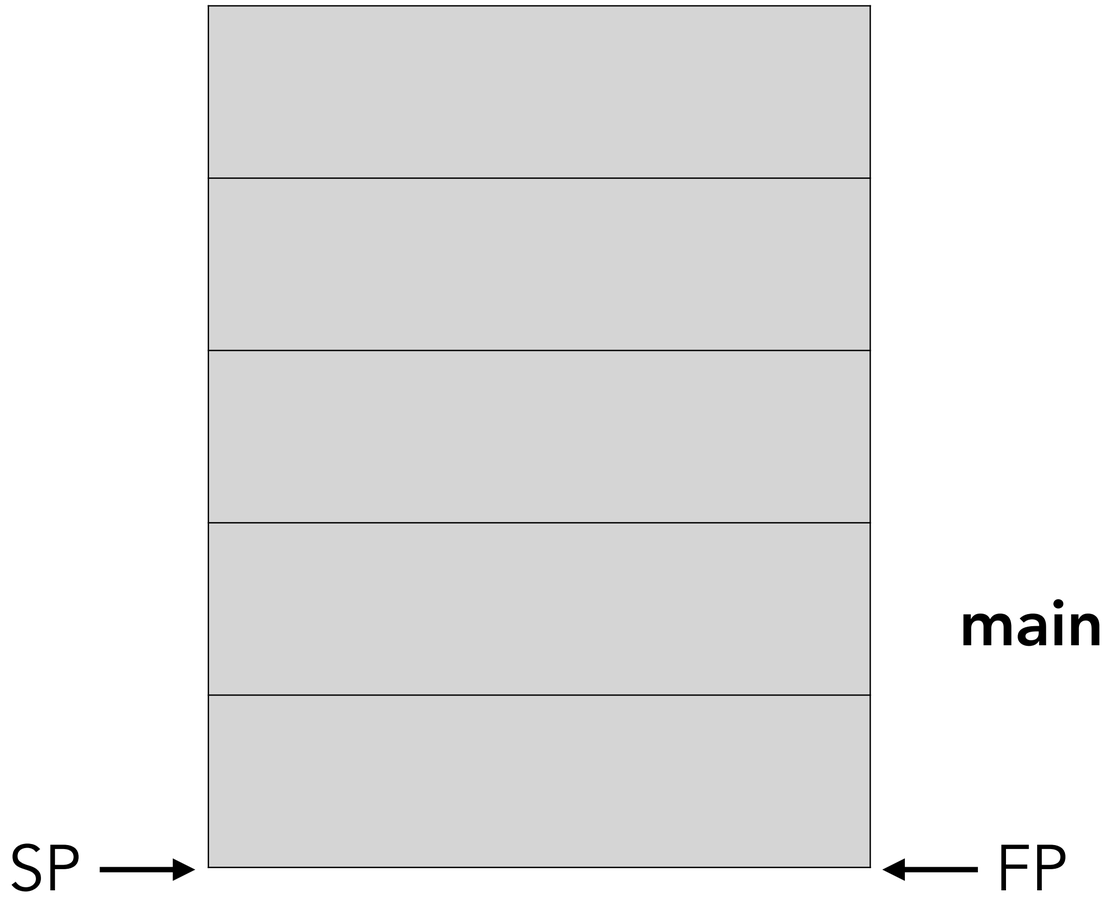
example.c – calling

```
void foo (int a, int b) {
    char buf1[16];
}


int main() {
    foo(3,6);
}
```


stolen w/ love from UMich

example.c – calling

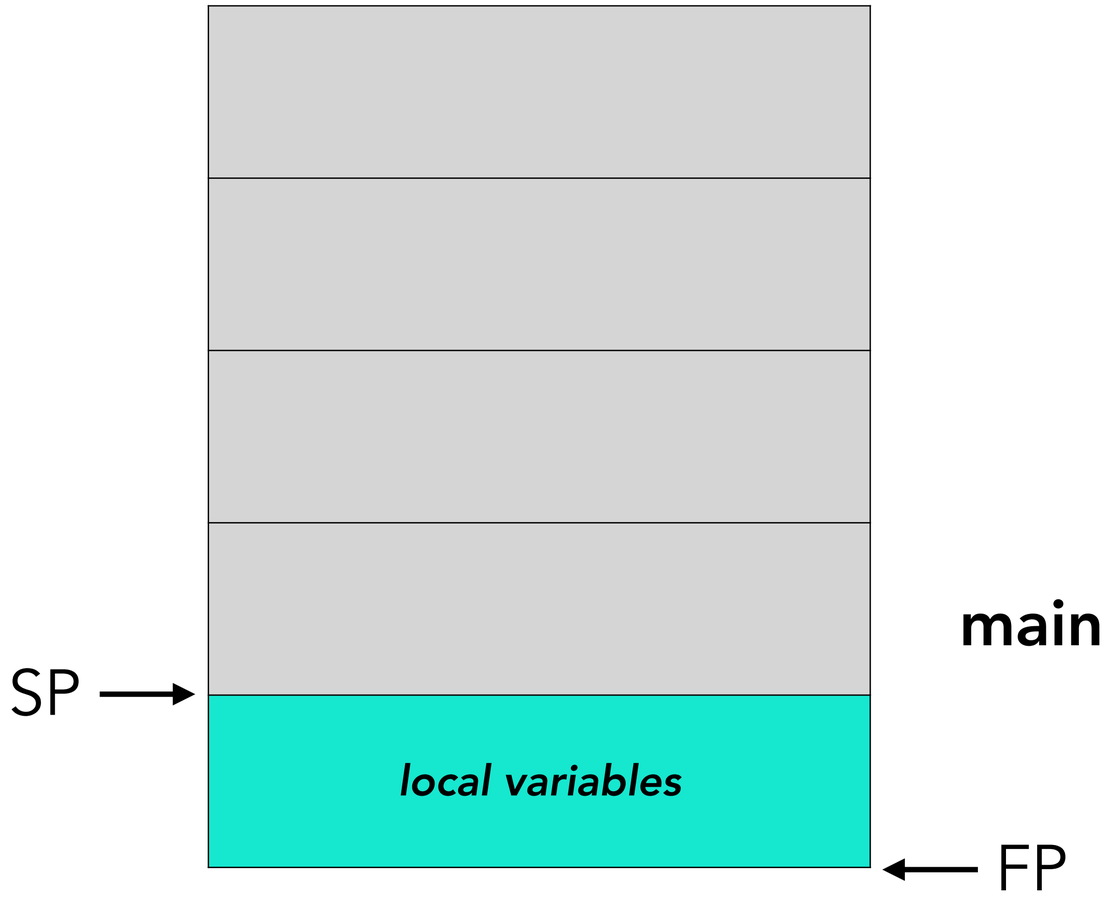
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – calling

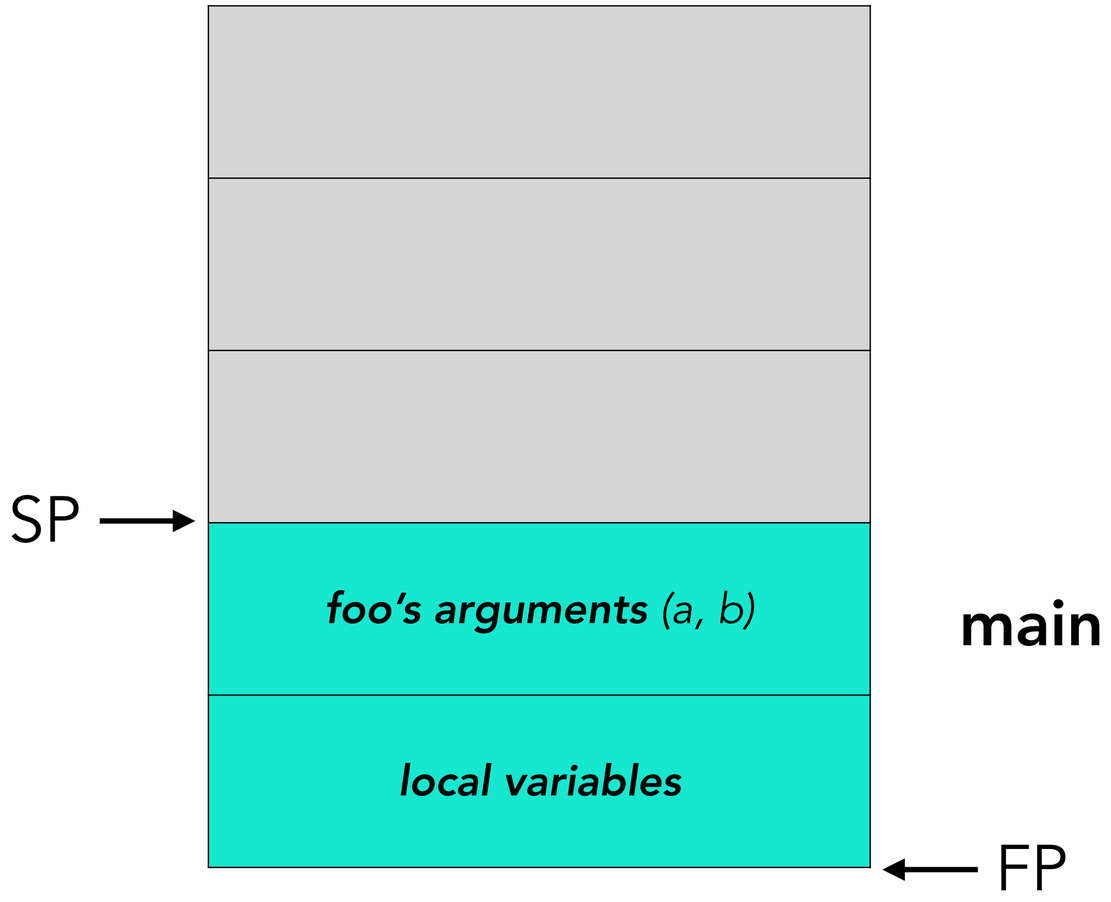
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – calling

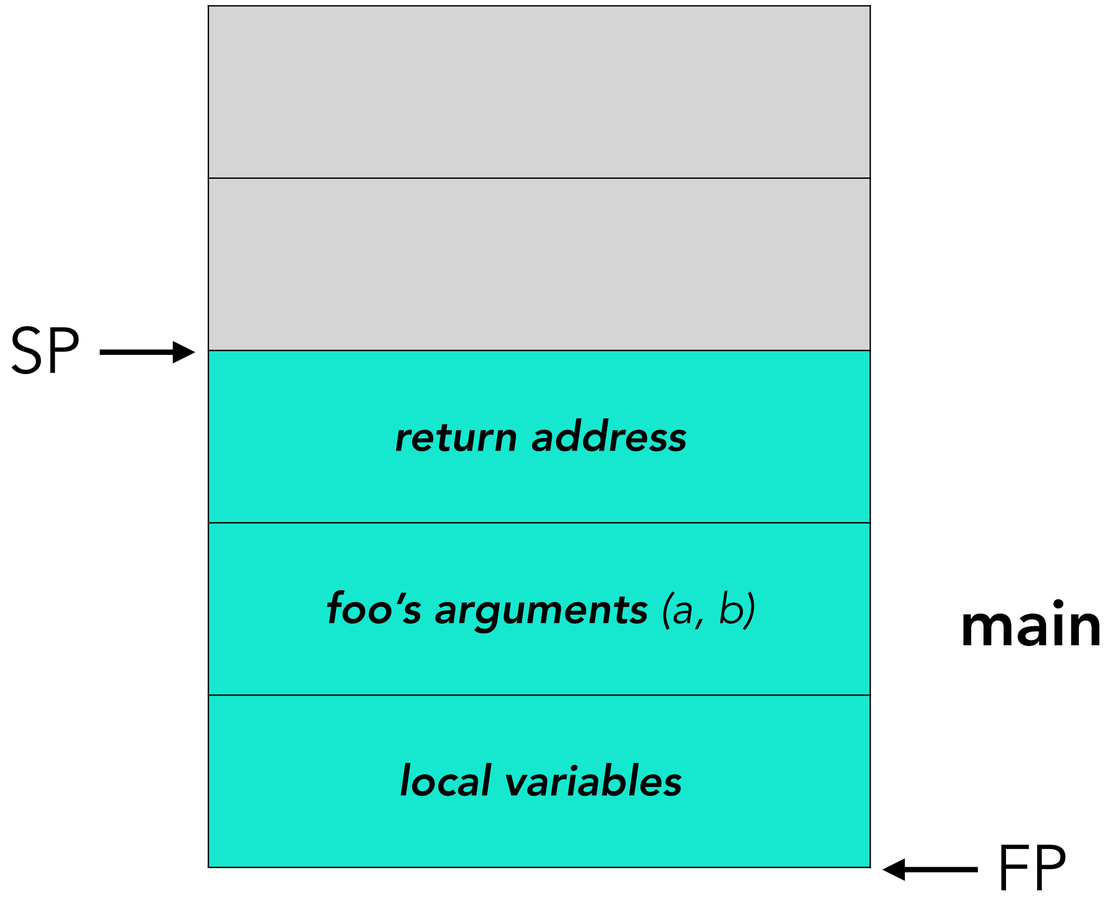
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – calling

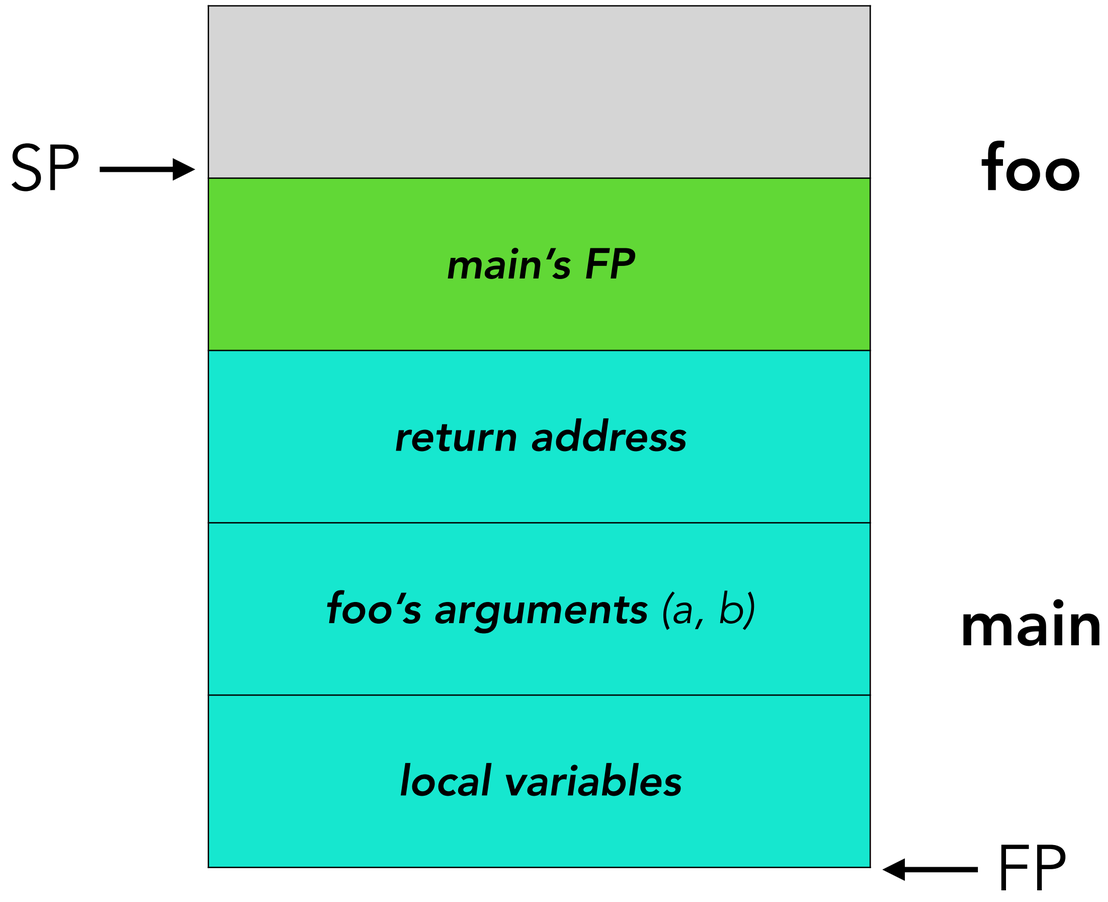
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – calling

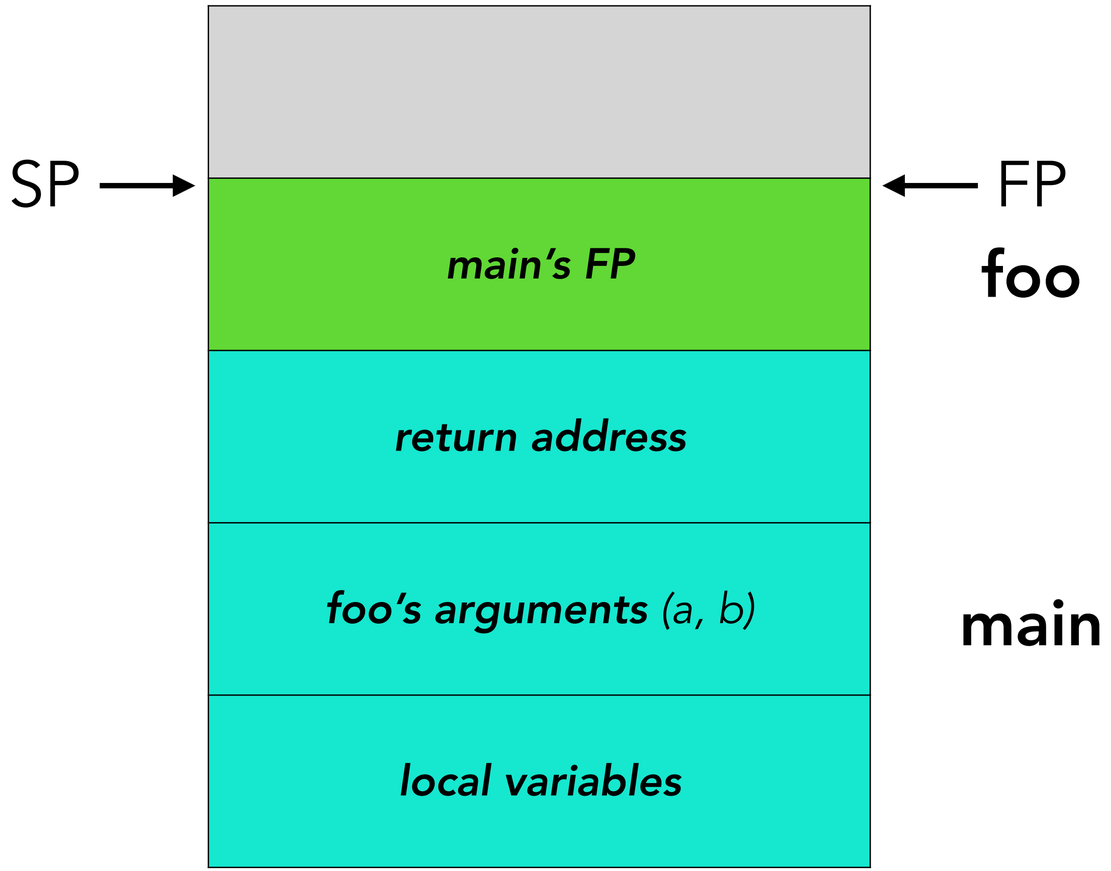
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – calling

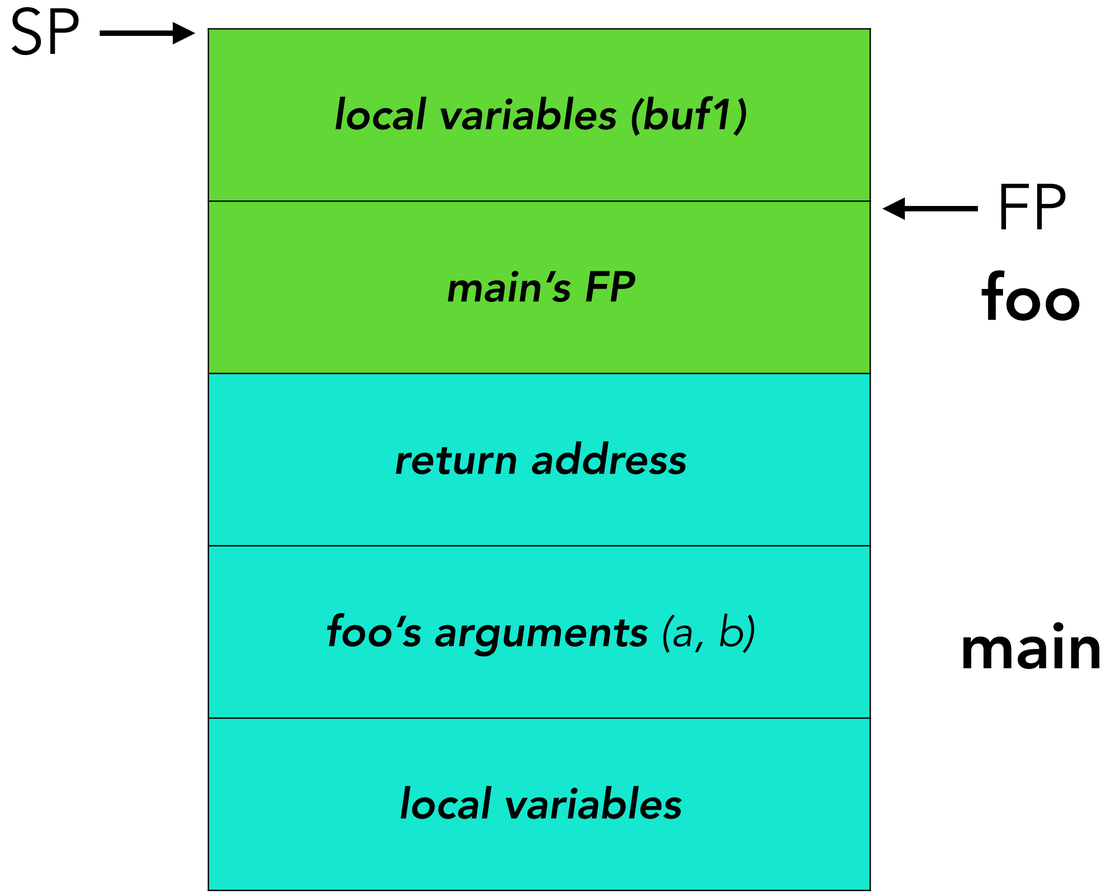
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – calling

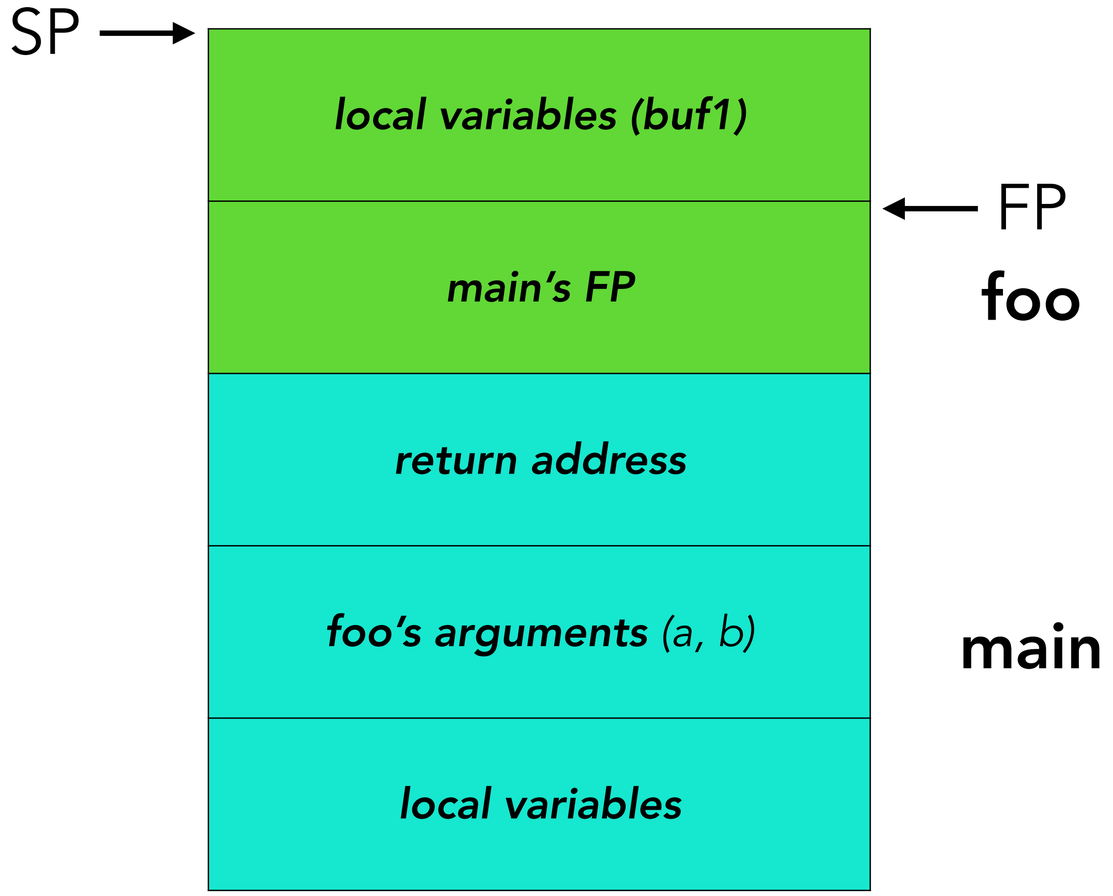
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – returning

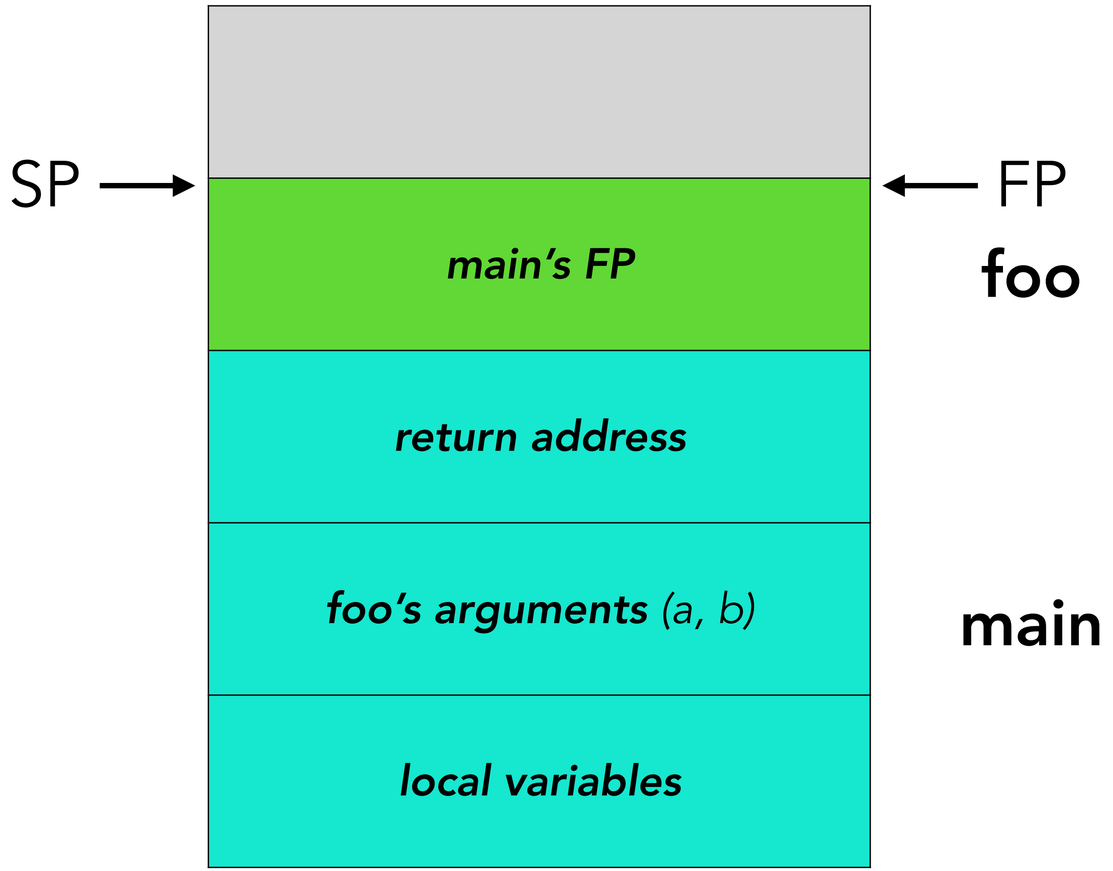
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – returning

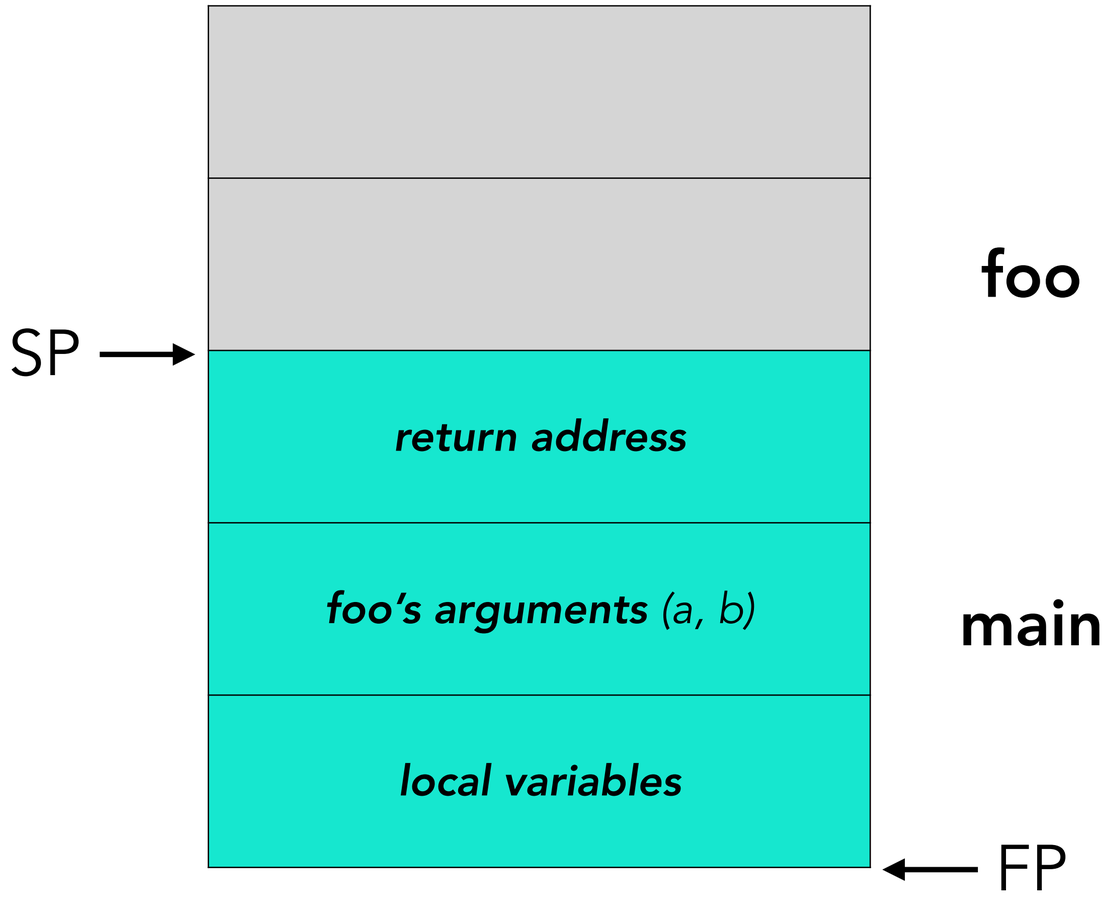
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – returning

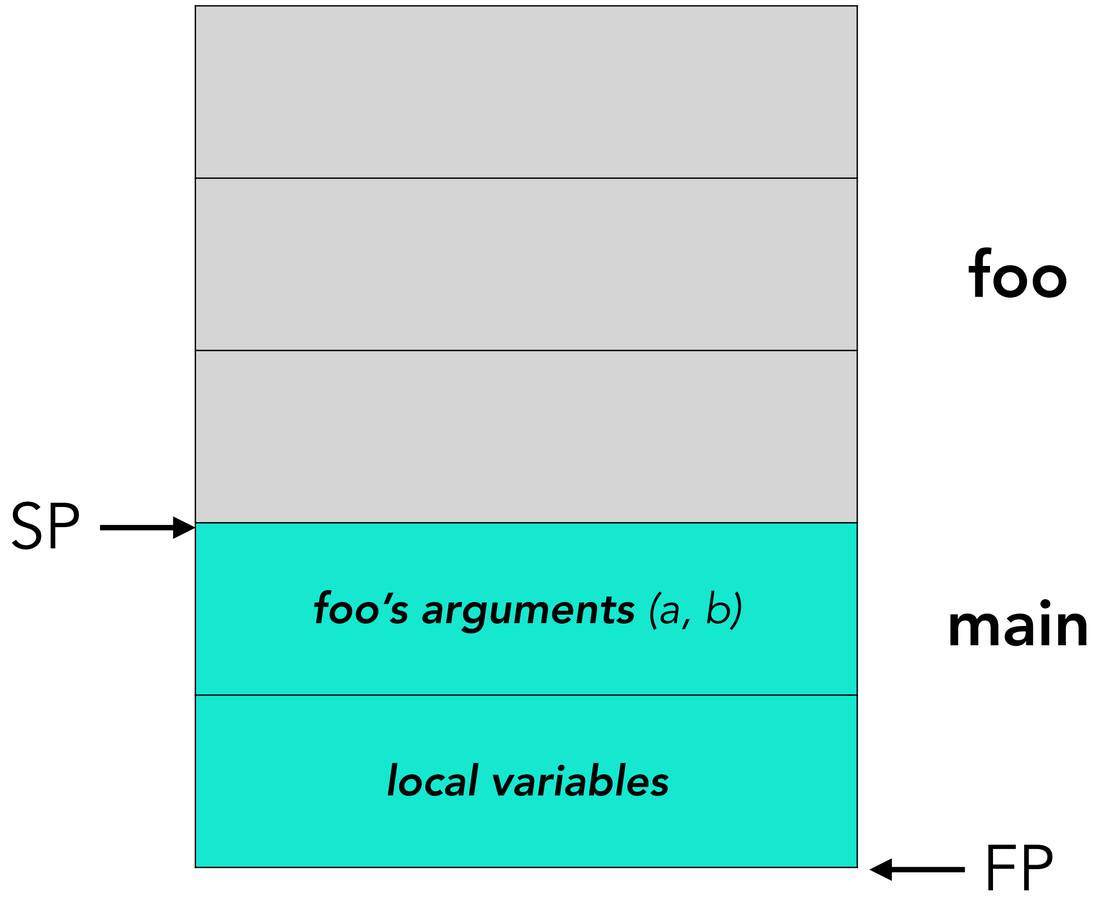
```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c – returning

```
void foo (int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



stolen w/ love from UMich

example.c (x86)

```
int main() {

    foo(3,6);

}

main:

    push %ebp
    movl %esp, %ebp
    push $0x06
    push $0x03
    call foo
    leave
    ret
```

```
void foo (int a, int b) {

    char buf1[16];

}

foo:

    push %ebp
    movl %esp, %ebp
    sub $0x10, %esp
    leave
    ret
```


stolen w/ love from UMich

example.c (x86)

```
int main() {

    foo(3,6);

}

main:

    push %ebp
    movl %esp, %ebp
    push $0x06
    push $0x03
    call foo
    leave
    ret
```

```
void foo (int a, int b) {

    char buf1[16];

}

foo:

    push %ebp
    movl %esp, %ebp
    sub $0x10, %esp
    leave
    ret
```


stolen w/ love from UMich

example.c (x86)

```
int main() {

    foo(3,6);

}

main:

    push %ebp
    movl %esp, %ebp
    push $0x06
    push $0x03
    call foo
    leave
    ret
```

```
void foo (int a, int b) {

    char buf1[16];

}

foo:

    push %ebp
    movl %esp, %ebp
    sub $0x10, %esp
    leave
    ret
```

```
mov %ebp, %esp
pop %ebp
```


stolen w/ love from UMich

example.c (x86)

```
int main() {

    foo(3,6);

}

main:

    push %ebp
    movl %esp, %ebp
    push $0x06
    push $0x03
    call foo
    leave
    ret
```

```
void foo (int a, int b) {

    char buf1[16];

}

foo:

    push %ebp
    movl %esp, %ebp
    sub $0x10, %esp
    leave
    ret
```

```
pop %eip
```


stolen w/ love from UMich

example.c (x86)

```
int main() {

    foo(3,6);

}

main:

    push %ebp
    movl %esp, %ebp
    push $0x06
    push $0x03
    call foo
    leave
    ret
```

```
void foo (int a, int b) {

    char buf1[16];

}

foo:

    push %ebp
    movl %esp, %ebp
    sub $0x10, %esp
    leave
    ret
```

stolen w/ love from UMich

Buffer Overflows

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}
```

Where is the problem with this code?

```
int main() {

    char str = “1234567890AB”;
    foo(str);

}
```


stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


int main() {

    char str = "1234567890AB";
    foo(str);

}
```

Buffer is not big enough to hold 12 bytes of input!

stolen w/ love from UMich

Buffer overflow example

```
int main() {

    char str = “1234567890AB”;
    foo(str);

}
```

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}
```

```
main:
    push %ebp
    mov  %esp, %ebp
    push str_ptr
    call foo
    leave
    ret
```

```
foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

```
str_ptr: “1234567890AB”
```


stolen w/ love from UMich

Buffer overflow example

```
int main() {

    char str = “1234567890AB”;
    foo(str);

}
```

```
main:
    push %ebp
    mov  %esp, %ebp
    push str_ptr
    call foo
    leave
    ret
```

str_ptr: “1234567890AB”

stolen w/ love from UMich

Buffer overflow example

```
int main() {

    char str = “1234567890AB”;
    foo(str);

}
```

```
main:
    push %ebp
    mov  %esp, %ebp
    push str_ptr
    call foo
    leave
    ret
```

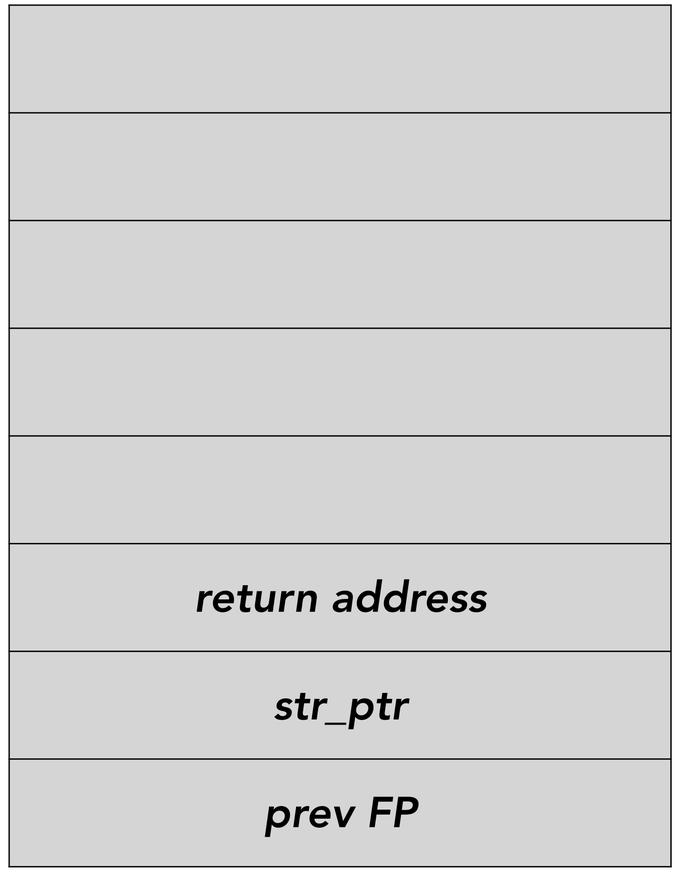
str_ptr: “1234567890AB”

stolen w/ love from UMich

Buffer overflow example

```
int main() {

    char str = “1234567890AB”;
    foo(str);

}
```

```
main:
    push %ebp
    mov  %esp, %ebp
    push str_ptr
    call foo
    leave
    ret
```



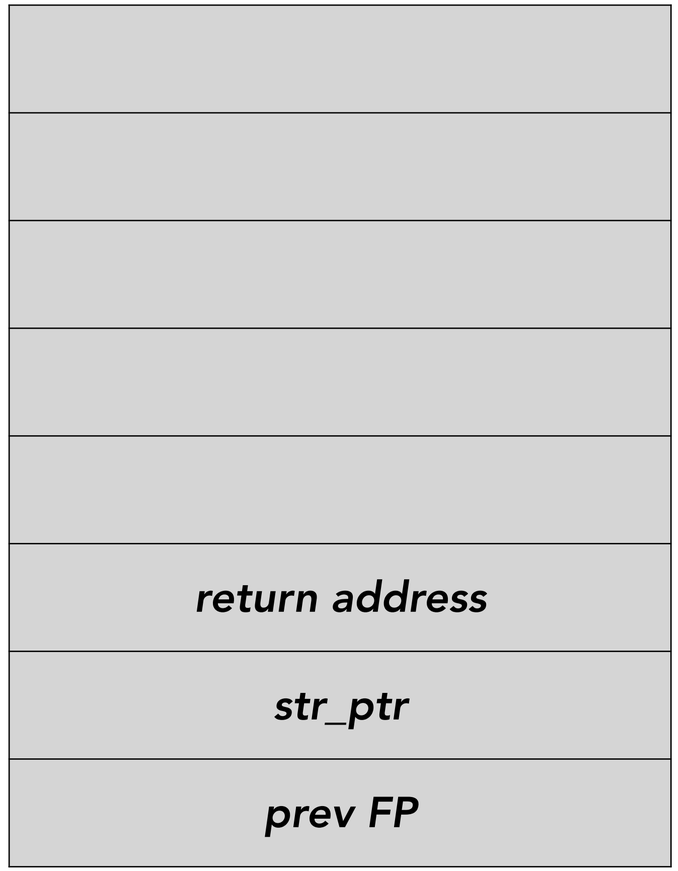
str_ptr: “1234567890AB”

stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

str_ptr: "1234567890AB"

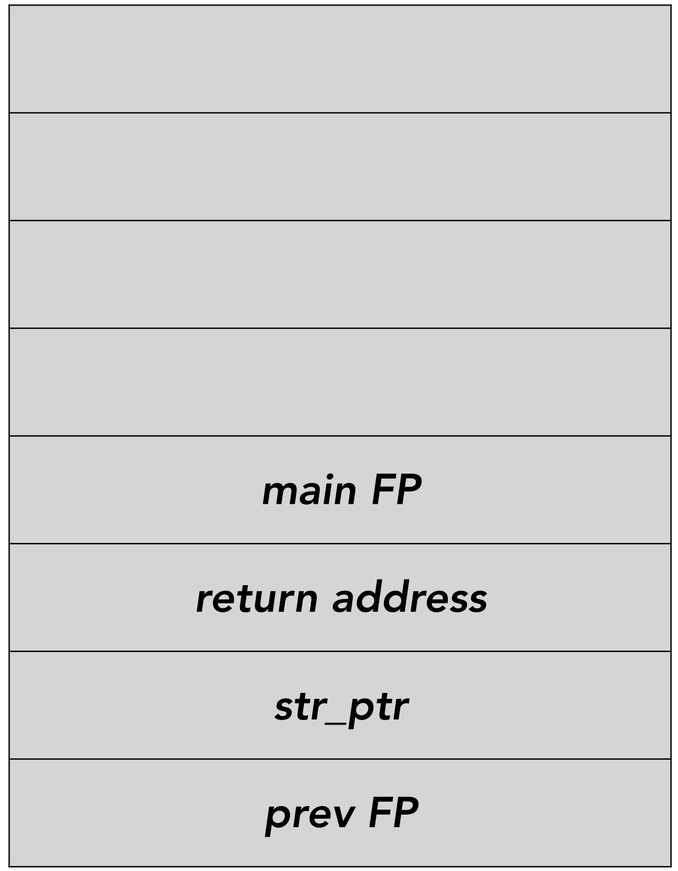


stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

str_ptr: “1234567890AB”

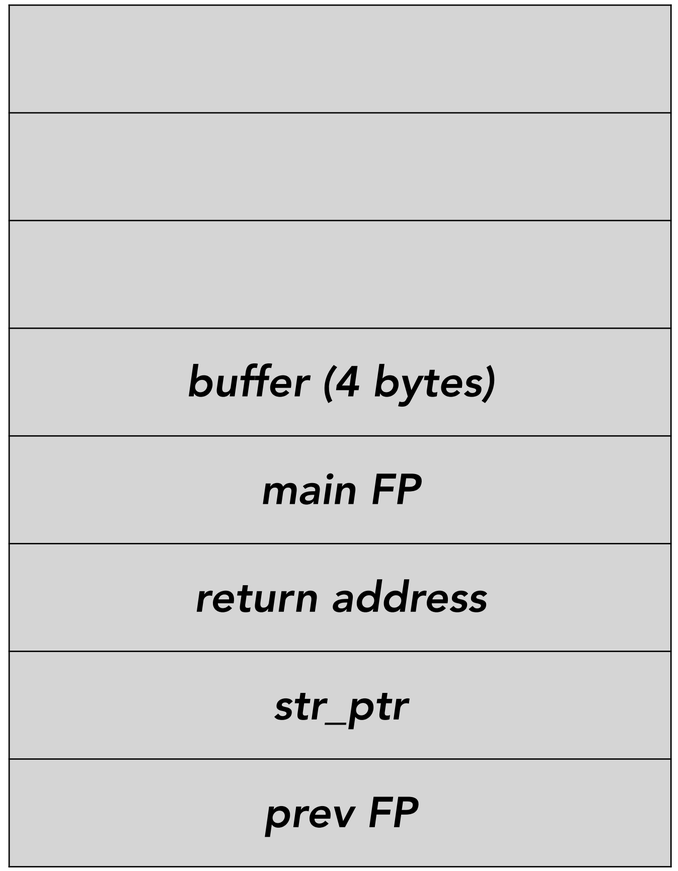


stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

str_ptr: "1234567890AB"

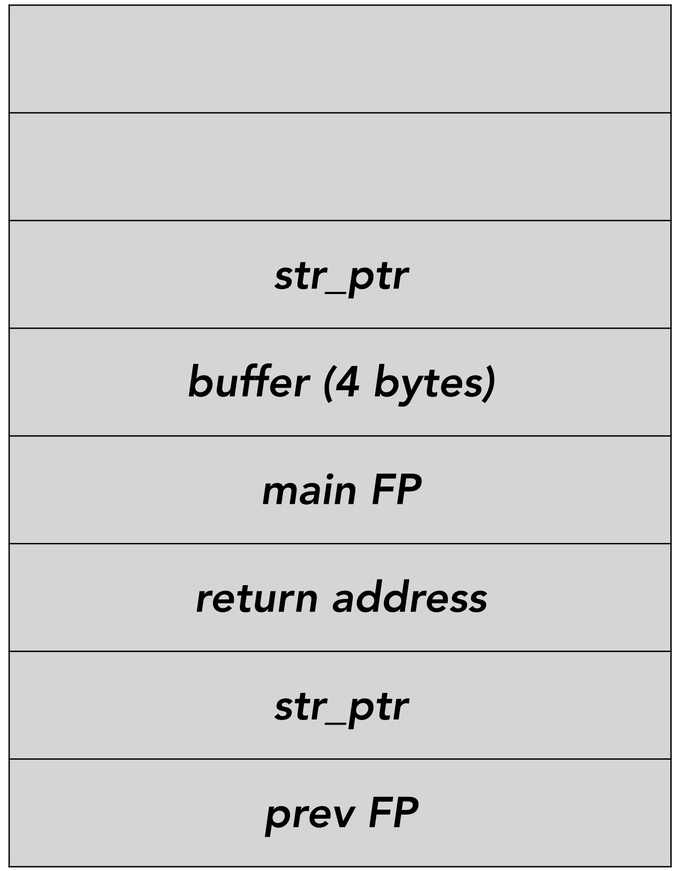


stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

str_ptr: "1234567890AB"

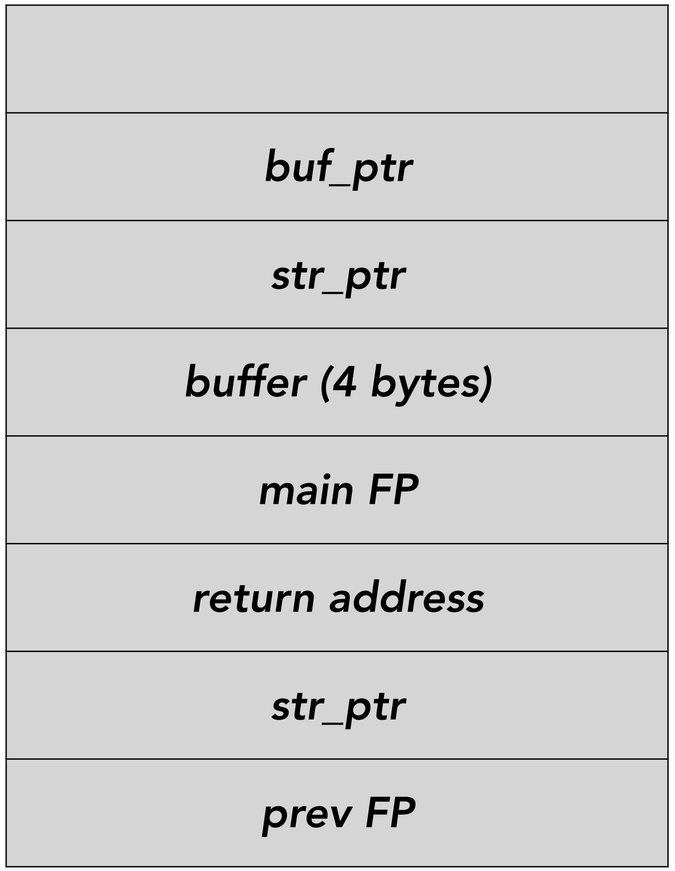


stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

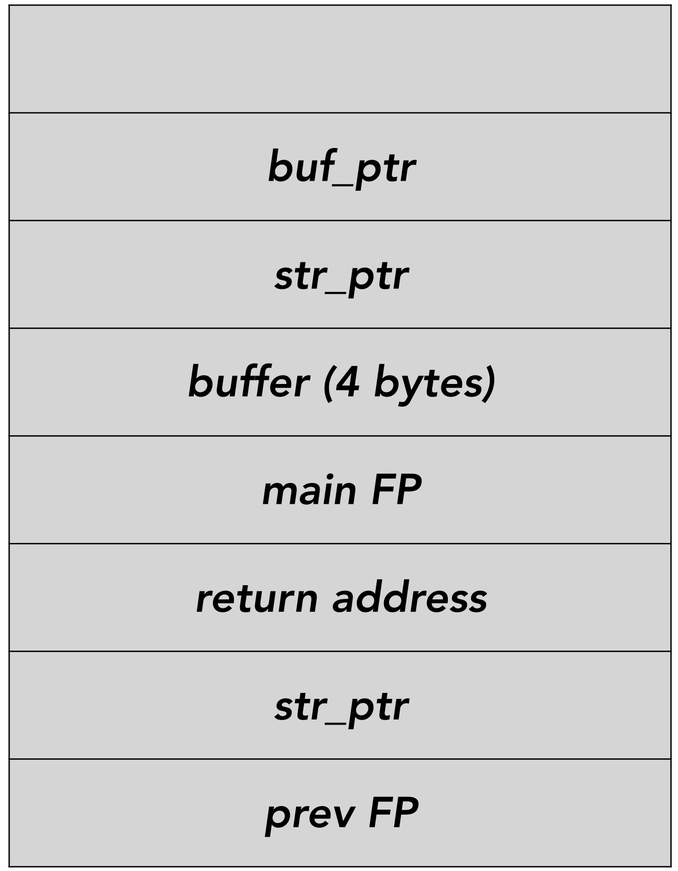
str_ptr: “1234567890AB”



stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

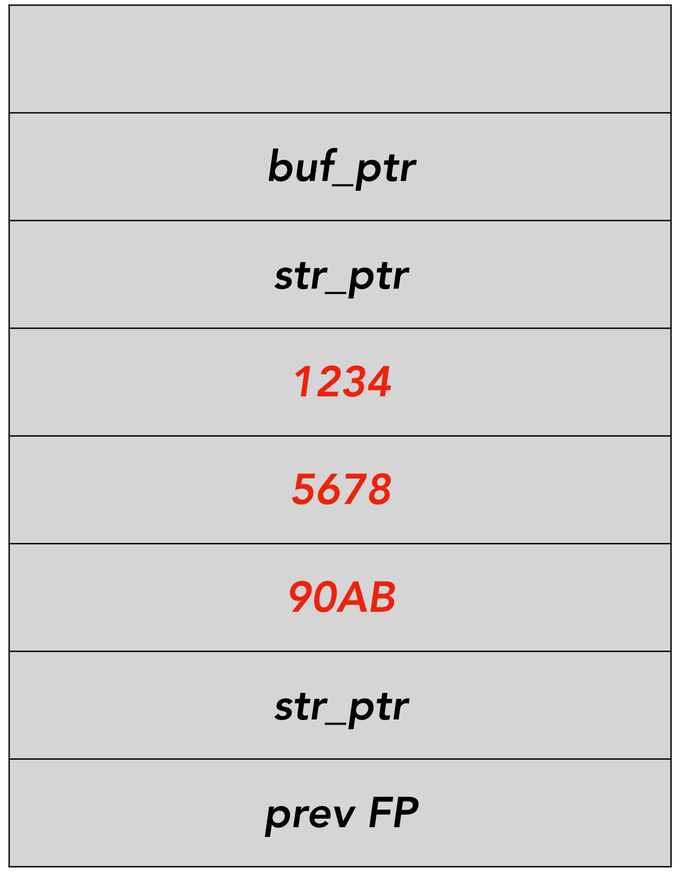


str_ptr: "1234567890AB"

stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

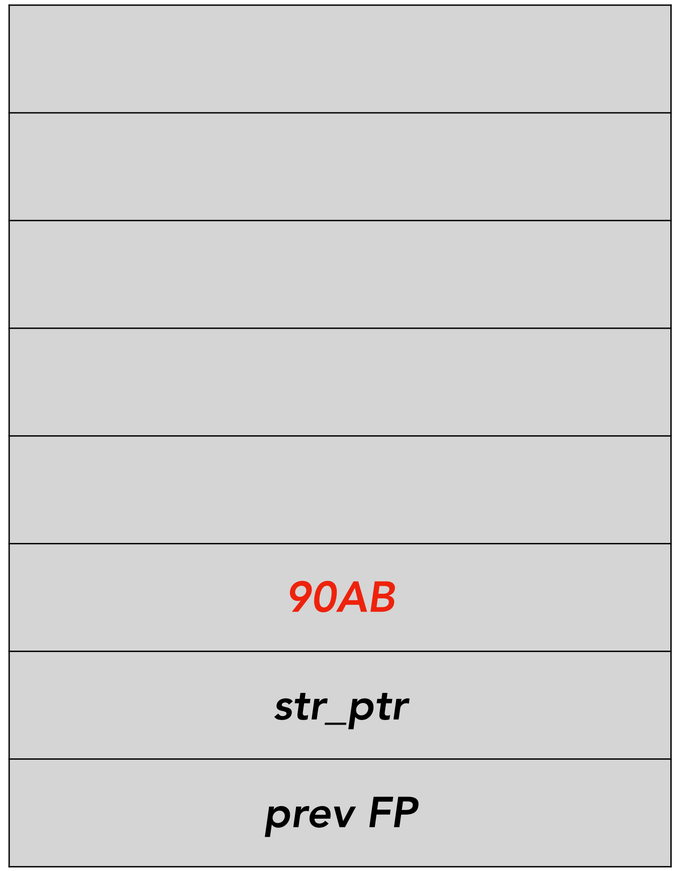


str_ptr: "1234567890AB"

stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

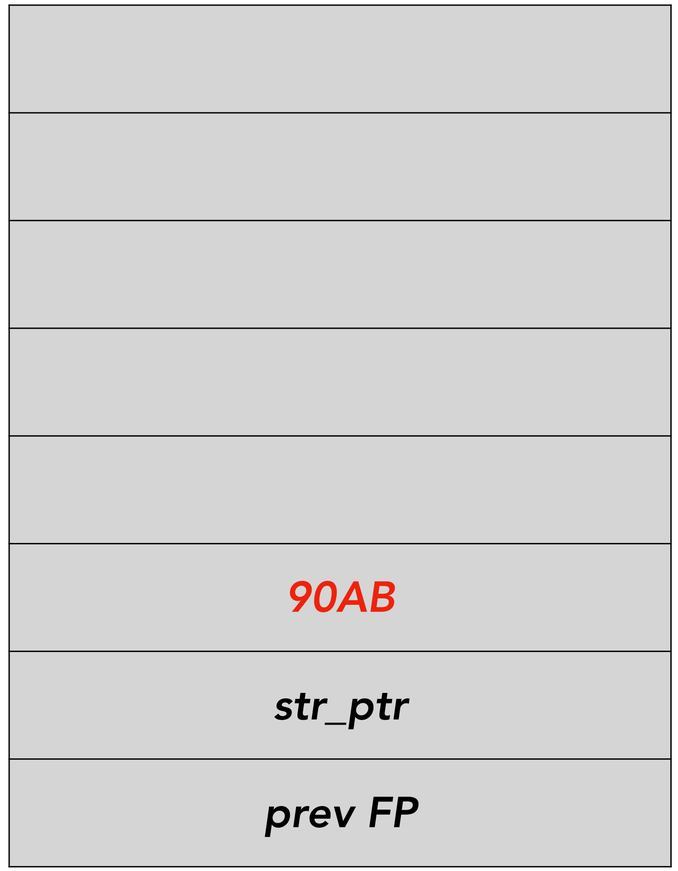


str_ptr: "1234567890AB"

stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}


foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```



str_ptr: "1234567890AB"

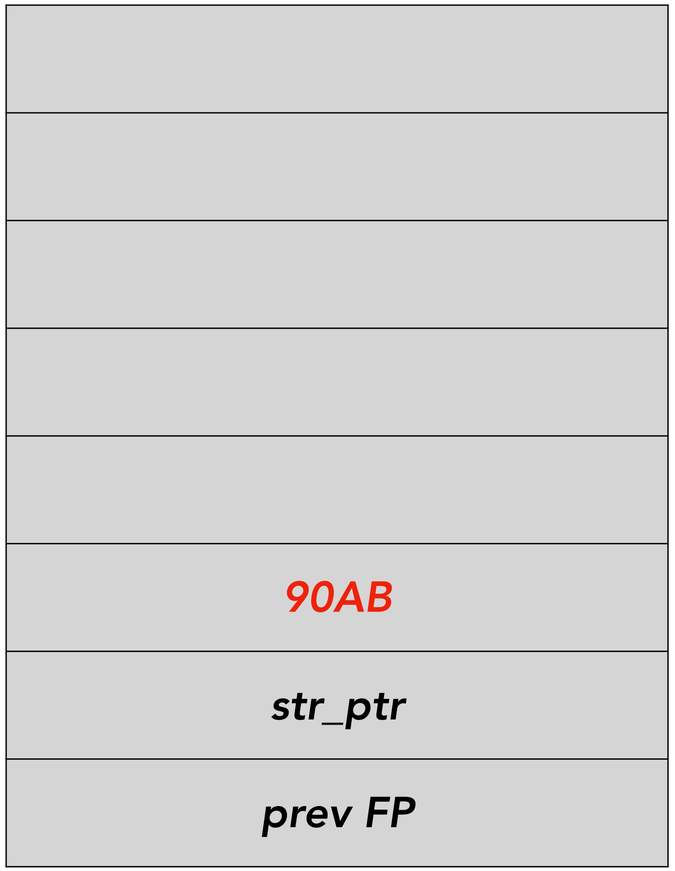
stolen w/ love from UMich

Buffer overflow example

```
void foo (char *str) {

    char buffer[4];
    strcpy(buffer, str);

}
```

What happens here?

```
foo:
    push %ebp
    mov  %esp, %ebp
    sub  $0x4, %esp
    push 0x8(%ebp)
    lea -0xc(%ebp),%edx
    push %edx
    call strcpy
    leave
    ret
```

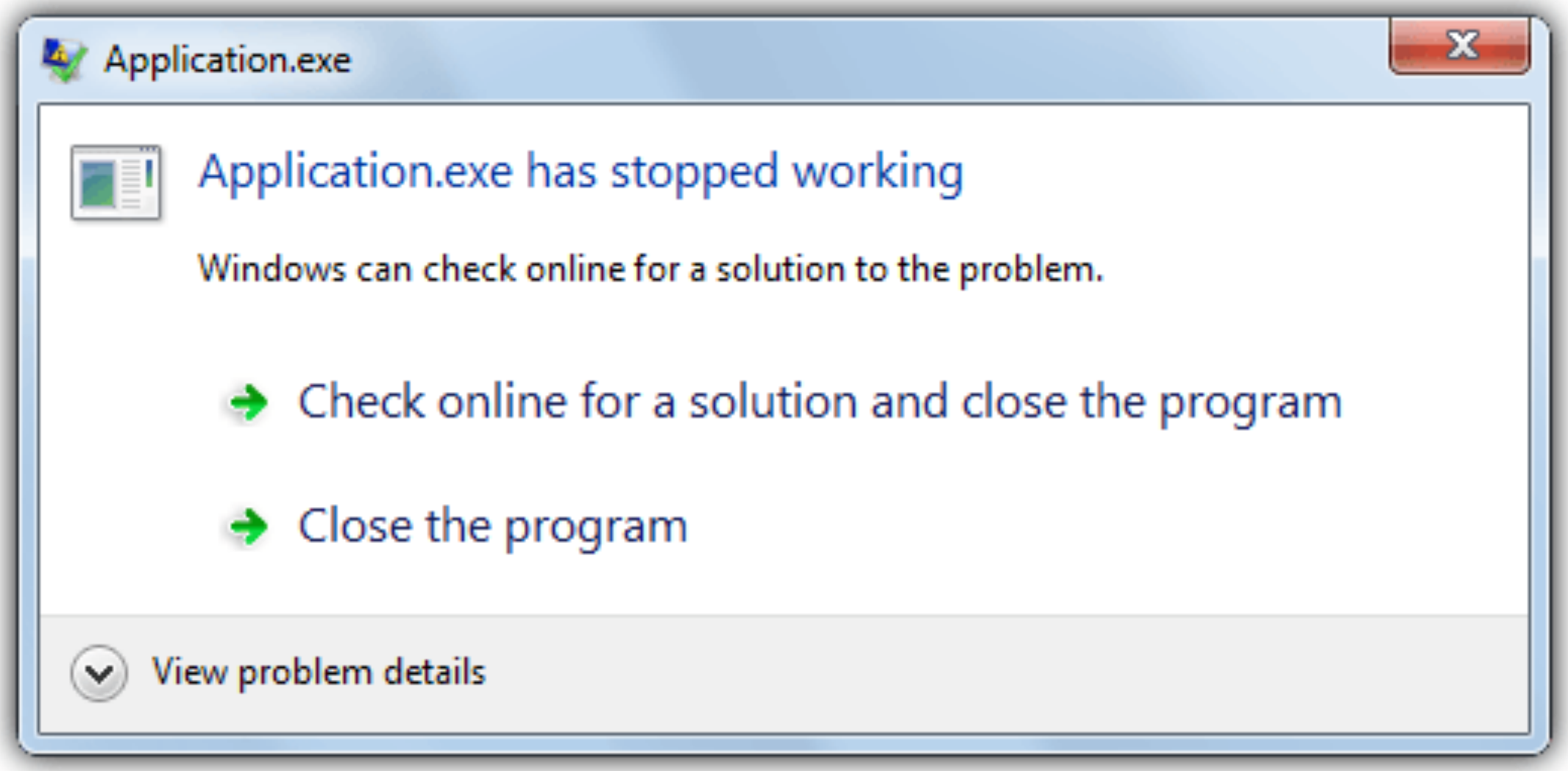


str_ptr: "1234567890AB"

stolen w/ love from UMich

Buffer overflow example

```
v
}
f
```



`str_ptr: "1234567890AB"`

stolen w/ love from UMich

So… what is a buffer overflow?

- An anomaly that occurs when a program writes data beyond the boundary of a buffer
- **Archetypal software vulnerability**
  - Ubiquitous in system software (C/C++)
    - OSes, web servers, web browsers, **embedded systems**, IoT devices…
  - If your program crashes with memory faults, you probably have a buffer overflow vulnerability
- A basic core concept that enables a broad range of possible attacks
  - Sometimes, a **single byte** is all an attacker needs

Why do buffer overflows exist?

- One reason — no automatic bounds checking in C/C++. Developers need to know what they are doing and check access bounds when necessary.
- Another reason: Default C standard library functions make it very easy to go past array bounds
  - Most standard string manipulation functions; `gets()`, `strcpy()`, `strcat()`, all **write to the destination buffer until they encounter a termination null byte in the input.**
- Deepak's version: If you hand someone a footgun, they'll probably shoot themselves in the foot

Smashing the stack

- If we have control over a buffer on the stack, we can overwrite **anything that appears above it.** Like what?

Smashing the stack

- If we have control over a buffer on the stack, we can overwrite **anything that appears above it.** Like what?
  - Other local variables
  - Saved frame point (%ebp)
  - Return address
  - Function arguments
  - Deeper stack frames (no frame by frame protection built in)
  - …Basically anything in process memory!

Smashing the stack — local variables

- Why might overwriting **local variables** or **function arguments** be bad?
  - Sort of depends on what the program is doing, but if an argument contains something sensitive… you're hosed
- Typical problem cases
  - Variables that store result of a security check
  - Variables used in checks (e.g., buffer sizes)
  - Data pointers (can corrupt writable stuff)
  - Function pointers (can direct control transfer)

```
void doValidStuff (int isValid, char*
inp)

{

    char username[8];
    strcpy(username, inp);
    if check(username, isValid) {
        // do some stuff
    }

}
```

Smashing the stack — control data

- The big one: the **return address**
  - Upon function return, control is transferred to an *attacker chosen address*
  - This enables what we call **arbitrary code execution** (the worst kind)
    - Attackers can re-direct to **their own code**, or code that already exists in the process (e.g., libc)
    - Reminder: theres ***nothing*** that distinguishes data from code, so all data (including input) will be interpreted as code if processor tries to transfer control there
- Aka: **game over, you're owned**

What's the worst thing we could put in a buffer?

- From a control flow perspective, it's **shellcode**
  - Aka, code that spawn a *shell*
- When manipulating the return address, have it return to **the beginning instruction of the shellcode**
- If you do that, you can run anything with the *same privileges* as the victim process
  - Aka; download malware, spawn new processes, read + exfiltrate any potentially privileged files… the harms are vast and many

Spawning shellcode

- How does one spawn a shell?
- Just need to call `execve` with the right arguments
  - `execve("/bin/sh", argv, NULL)`
- In "Smashing the Stack for Fun and Profit," Aleph One composes shellcode that can work with **no null bytes** (aka suitable for smashing the stack)… worth a read!

A simple buffer overflow strategy

- Identify a buffer overflow vulnerability (i.e., uses an unsafe string function)
- Place some shellcode in a buffer
- Find a way to overrun the buffer to modify the return address
- Modify return address to the buffer's address in memory (where you placed the shellcode)
- Profit!

You will get lots of experience doing this in PA2.

So let's just fix all the bad string functions!

- `char *strncpy(char *dst, const char *src, size_t len);`
- `char *strncat(char *s, const char *append, size_t count);`
- strn* family of functions attempts to remedy this problem
  - Introduce a function parameter to specify the safe amount to copy
- strncpy() copies at most len characters from src into dst
  - If src is less than len characters long, the remainder of dst is filled with null bytes. Otherwise, dst is not terminated.
- On surface, these seem good, except….

strncpy

- `char *strncpy(char *dst, const char *src, size_t len);`
- `char *strncat(char *s, const char *append, size_t count);`
- strn* family of functions attempts to remedy this problem
  - Introduce a function parameter to specify the safe amount to copy
- strncpy() copies at most len characters from src into dst
  - If src is less than len characters long, the remainder of dst is filled with null bytes. Otherwise, dst is not terminated.
- On surface, these seem good, except….

strncpy failures

- Developers don't usually know how to use it correctly
- Vulnerability in `htpasswd.c` in Apache 1.3
  - ```
    strcpy(record, user);
    strcat(record, ":");
    strcat(record,cpw);
    ```
- "Solution"
  - ```
    strncpy(record, user, MAX_STRING_LEN - 1);
    strncat(record, ":", 1);
    strncat(record, cpw, MAX_STRING_LEN - 1);
    ```
- Can write up to 2*(MAX_STRING_LEN - 1) bytes! Bad!

strncpy failures

```
void main (int argc, char **argv) {
      char program_name[256];
      strncpy(program_name, argv[0], 256);
      f(program_name);
}
```

- What's wrong with this code?

strncpy failures

```
void main (int argc, char **argv) {
      char program_name[256];
      strncpy(program_name, argv[0], 256);
      f(program_name);
}
```

- What's wrong with this code?
  - program_name may not be null terminated… and when you hand that to *another* function (maybe one you didn't write) you could be hosed
  - In other words, **strncpy** breaks the **data structure invariant of a string**
    - Extremely unrecommended in practice

Bottom line: strings in C kinda suck

- **`strncpy()/strncat()`** still problematic today
  - They do not guarantee NULL termination
  - The design forces the developer to keep track of residual buffer lengths
    - Requires performing awkward arithmetic (len(x) - 1, anyone?) which… as we all know, are very easy to get wrong
  - There is **no way** to check if the source string is truncated, leading to all kinds of annoying ambiguities
- If you **must** manipulate strings in C, the `strl*` family are **much** safer
  - Guarantees NULL termination and doesn't require complex address arithmetic

Good news: it’s not just strings

- C string functions get a bad rap… but lots of ways to overrun a buffer
  - Memcpy, pointer arithmetic, bad casts, etc.
  - Ultimately, it’s a side effect of C’s unsafe memory semantics… strings are just the most common and easiest use case to understand
  - You will get some exposure to pointer failures, integer overrun failures, etc. in PA2!

Spotting Buffer Overflows

- Three primary things to look out for:
  - Missing checks
  - Avoidable checks
  - Wrong checks

Missing checks

- No test to make sure memory writes stay within bounds
- Examples —
  - strcpy()
  - gets()
  - Most of the examples we saw today….

Avoidable check

- The check to make sure that memory stays within intended bounds can be bypassed in some way
- Example
  - libpng png_handle_tRNS()
  - 2004
- Good demonstration of how an attacker can manipulate internal state by providing “correct” input

```
356    if (png_ptr->color_type == PNG_COLOR_TYPE_PALETTE)
357    {
358       if (!(png_ptr->mode & PNG_HAVE_PLTE))
359       {
360          /* Should be an error, but we can cope with it */
361          png_warning(png_ptr, "Missing PLTE before tRNS");
362       }
363       else if (length > png_ptr->num_palette)
364       {
365          png_warning(png_ptr, "Incorrect tRNS chunk length");
366          png_crc_skip(png_ptr, length);
367          return;
368       }
```

Avoidable check

- Sometimes, the check comes too late…
  - What's the problem here?

```
1  #define BUFLEN 20
2
3  void foo(char *s)
4  {
5      char buf[BUFLEN];
6
7      strcpy(buf, s);
8      if(strlen(buf) >= BUFLEN)
9      {
10         //handle error
11     }
12 }
```

Wrong check

- The test to check if memory is in bounds is just **wrong**
- Very easy to do! Off-by-one errors in code…. not trivial
  - Especially accounting for null bytes
  - Suspicious arithmetic is usually a culprit…
- Here's an example…
  - OpenBSD
  - 2003

```
/*
 * Join the two strings together, ensuring that the right thing
 * happens if the last component is empty, or the dirname is root.
 */
if (resolved[0] == '/' && resolved[1] == '\0')
    rootd = 1;
else
    rootd = 0;

if (*wbuf) {
    if (strlen(resolved) + strlen(wbuf) + rootd + 1 > MAXPATHLEN) {
        errno = ENAMETOOLONG;
        goto err1;
    }
    if (rootd == 0)
        (void)strcat(resolved, "/");
    (void)strcat(resolved, wbuf);
}
```

Fundamental principle of buffer overflows

- **Buffer overflows exist because the runtime system mixes code and data.**
  - We'll see this time and time again in this class…

Addressing buffer overflows

- Best way to avoid these bugs is to not have them in the first place
  - If you can, avoid C/C++ for systems programming. Use a memory-safe language instead (Rust, Go)
  - Train developers to understand these bugs and their ramifications (can only really get you so far)
- Finding bugs is a must
  - Manual code review, static analysis, fuzzing, etc., — this is a **whole subfield of computer security**
- Or… we can make the bugs harder to exploit
  - More on this in two lectures (AppSec defenses)

Next time…

- We get even more in the weeds
  - Format strings, integer overflows, and return-oriented programming (one of the coolest attacks ever, IMO, the original paper is from a UCSD faculty member)
- PA1 is **due** on Thursday! See discussion, Piazza, etc. for more
  - Good luck!