Week 6: Lecture A
Defending Applications

Tuesday, September 24, 2024

Announcements

- **Project 2: AppSec** released
  - **Deadline:** Thursday, October 17th by 11:59PM

Project 2: Application Security

Deadline: Thursday, October 17 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

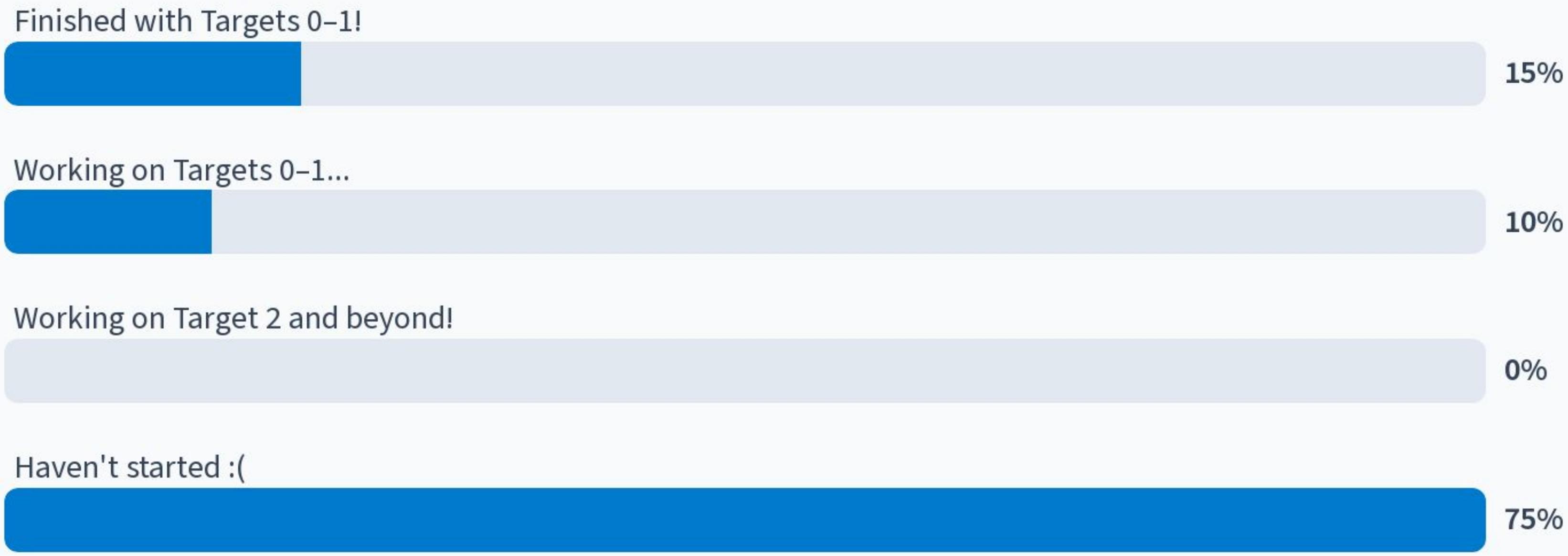
Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

Helpful Resources

- The CS 4440 Course Wiki
- VM Setup and Troubleshooting
- Terminal Cheat Sheet
- GDB Cheat Sheet
- x86 Cheat Sheet
- C Cheat Sheet

Table of Contents:

Project 2 progress?



Announcements

- **Project 1** grades are now available on **Canvas**
- **Statistics:**
  - Average score: **100%**
  - Last year's average: **85%**
- **Fantastic job!**

Announcements

- **Project 1** grades are now available on **Canvas**

- Think we made an error? Request a regrade!
  - Valid regrade requests:
    - You have verified your solution is correct (i.e., we made an error in grading)

Project 1 Regrade Requests (see **Piazza** pinned link):
Submit by **11:59 PM** on **Monday 9/30** via **Google Form**

Announcements

- Last lecture ran out of time (**sorry!**)
  - If you attended but didn't get credit (e.g., you didn't fill-in **PollEverywhere** fast enough), **please email me**

Announcements

- Last lecture ran out of time (**sorry!**)
  - If you attended but didn't get credit (e.g., you didn't fill-in **PollEverywhere** fast enough), **please email me**
- Thursday's lecture: **automated bug-finding**
  - **Guest lecture** (I will be out of town traveling)
  - TA Ethan will tackle the pre-lecture recap slides
  - Main lecture by Gabe Sherman (my PhD student)
  - **Don't miss it—one of the coolest security topics!**

Announcements

utahsec

See Discord for meeting info!

utahsec.cs.utah.edu

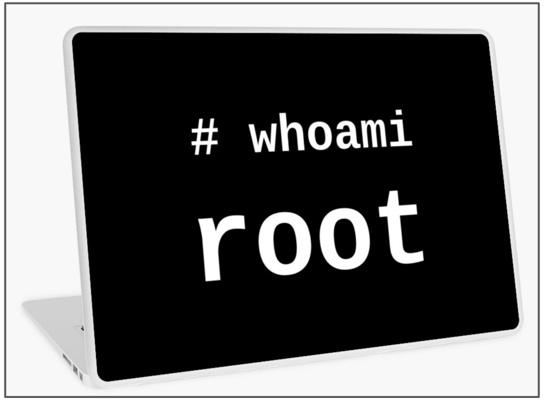
Questions?

Last time on CS 4440…

Shellcode
Constructing Exploits
Pointer Dereferences
Integer Overflows

Shellcode

- **Attacker goal:** make program open a **root shell**
  - Root-level permissions = **total system ownage**
  - **You'll do this in Project 2!**



- **Shellcode** = code to open a root shell
  - Inject this somewhere and **direct execution to it**
  - Basic structure:
    1. Call `setuid(0)` to set user ID to "`root`"
    2. Open a shell with `execve("/bin/sh")`

```
setuid(0)  +  execve("/bin/sh")
```

Where to begin?

- Mnemonic device to help guide your attack-planning thought process

D : Dive into the **source code**

E : Estimate the **stack frame**

N : **NOP-out** the entire frame

N : NOP-out the **return address**

I : **Inspect** program's memory

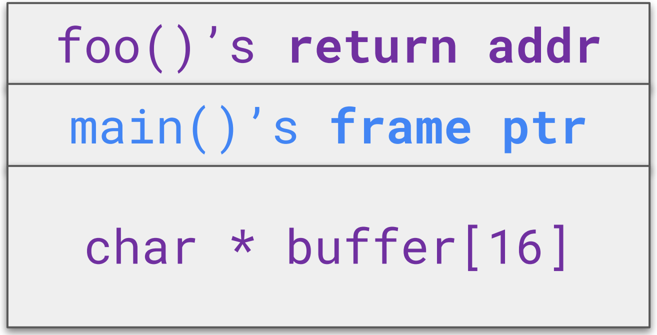
S : **Setup** and **stabilize** attack!

This acronym is silly…

But the **high-level steps** will get you a long way!

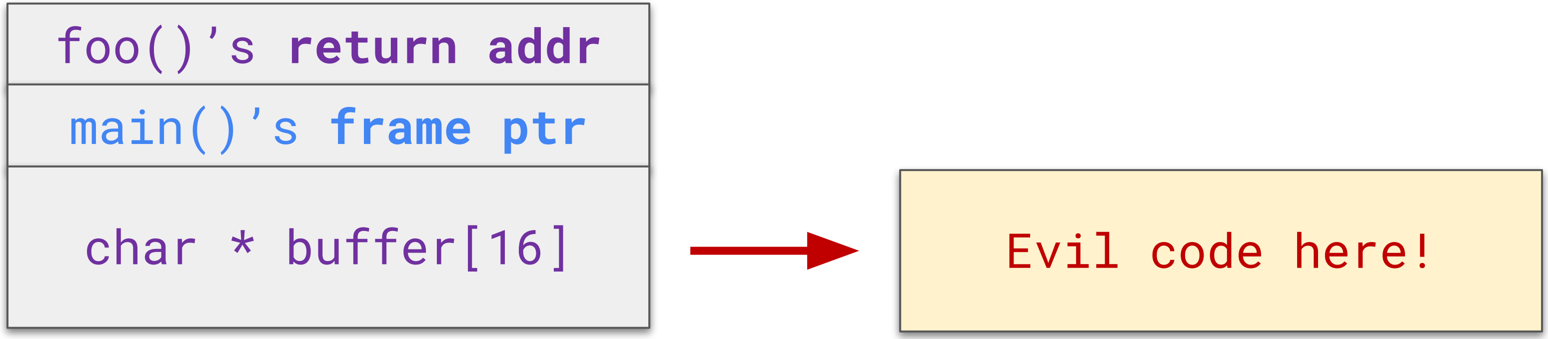
Exploiting Buffer Overflows

- **Key idea:** inject evil code inside buffer, and **redirect execution to it**



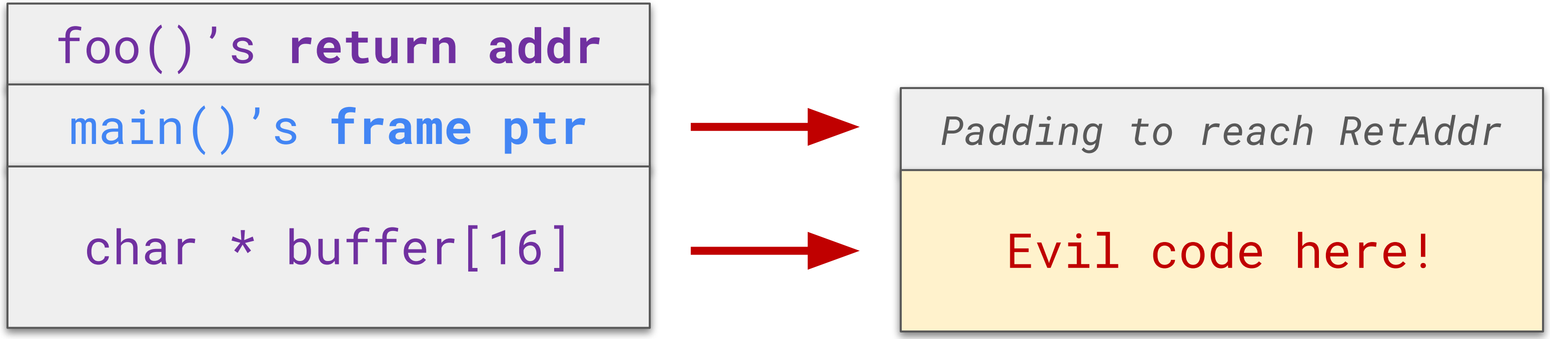
Exploiting Buffer Overflows

- **Key idea:** inject evil code inside buffer, and **redirect execution to it**



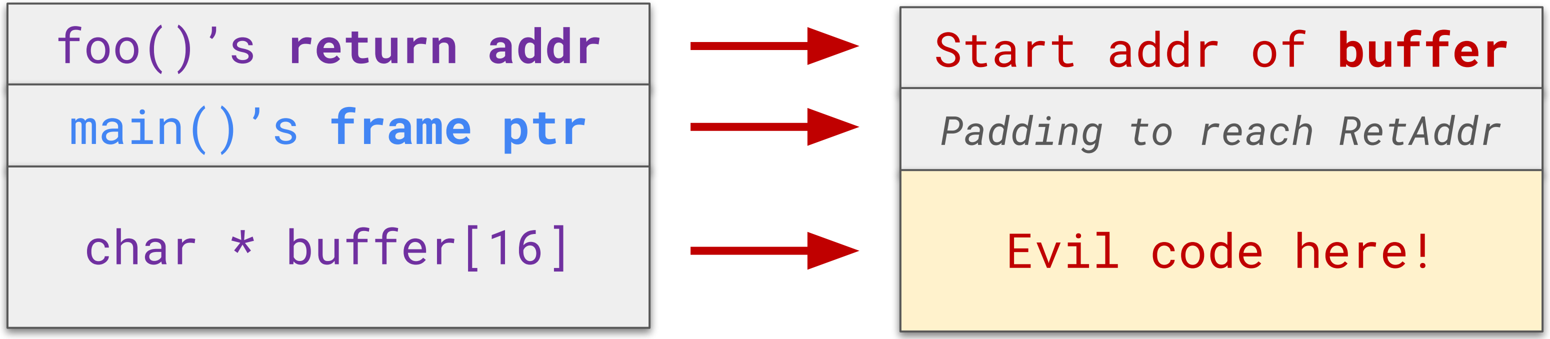
Exploiting Buffer Overflows

- **Key idea:** inject evil code inside buffer, and **redirect execution to it**



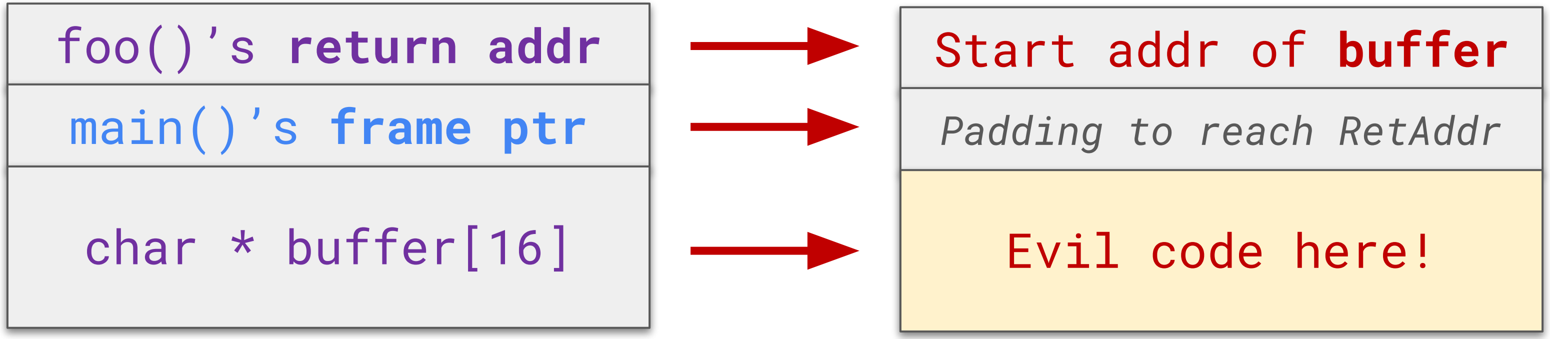
Exploiting Buffer Overflows

- **Key idea:** inject evil code inside buffer, and **redirect execution to it**



Exploiting Buffer Overflows

- **Key idea:** inject evil code inside buffer, and **redirect execution to it**



When **foo()** returns, execution will proceed to our **buffer's address**…
Thus **executing our evil code!**

Bounded vs. Unbounded Writes

- **Targets 0–2** permit **unbounded** writes
  - We can overwrite **anything** in the higher stack memory
  - Thanks to dangerous functions `gets()` and `strcpy()`
  - Definitely don't use these functions in your own code!
- **Targets 3–4** are **bounded** writes… limited reach!
  - **Target 3:** we can only write `8 + sizeof(buf)` bytes
  - **Target 4:** we can only write `count` bytes (via `fread()`)

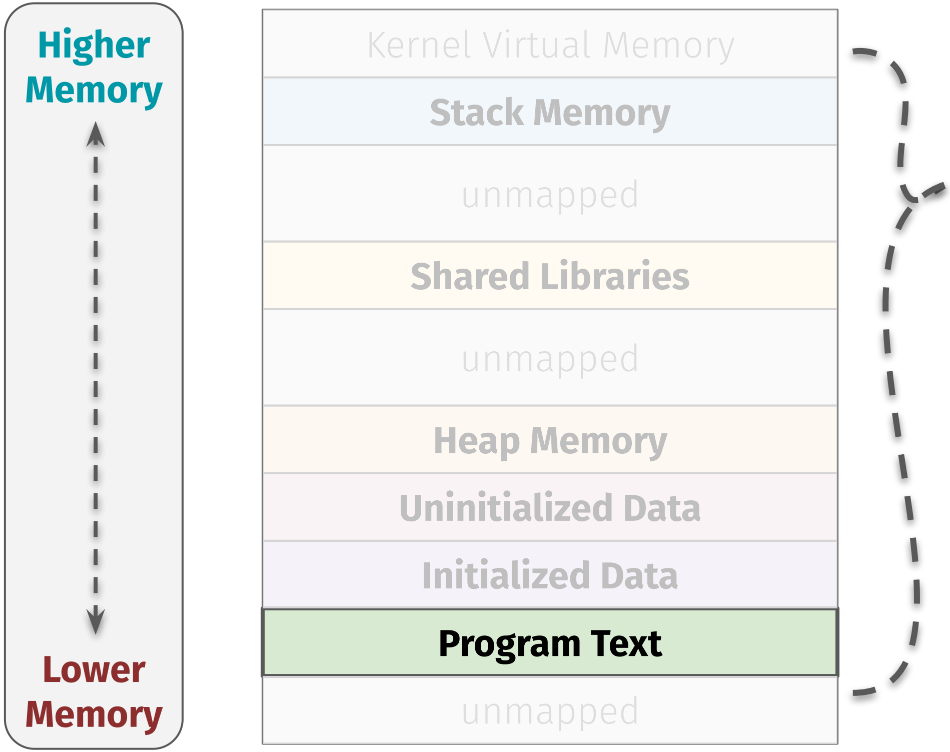
Bounded vs. Unbounded Writes

- **Targets 0–2** permit **unbounded** writes
  - We can overwrite **anything** in the higher stack memory
  - Thanks to dangerous functions `gets()` and `strcpy()`
  - Definitely don't use these functions in your own code!

- **Targets 3–4** are **bounded** writes… limited reach!
  - **Target 3:** we can only write `8 + sizeof(buf)` bytes
  - **Target 4:** we can only write `count` bytes (via `fread()`)

For **bounded** writes, we have to get creative and **find a way to overwrite** what we want!

Memory Addresses Point to Memory Slots



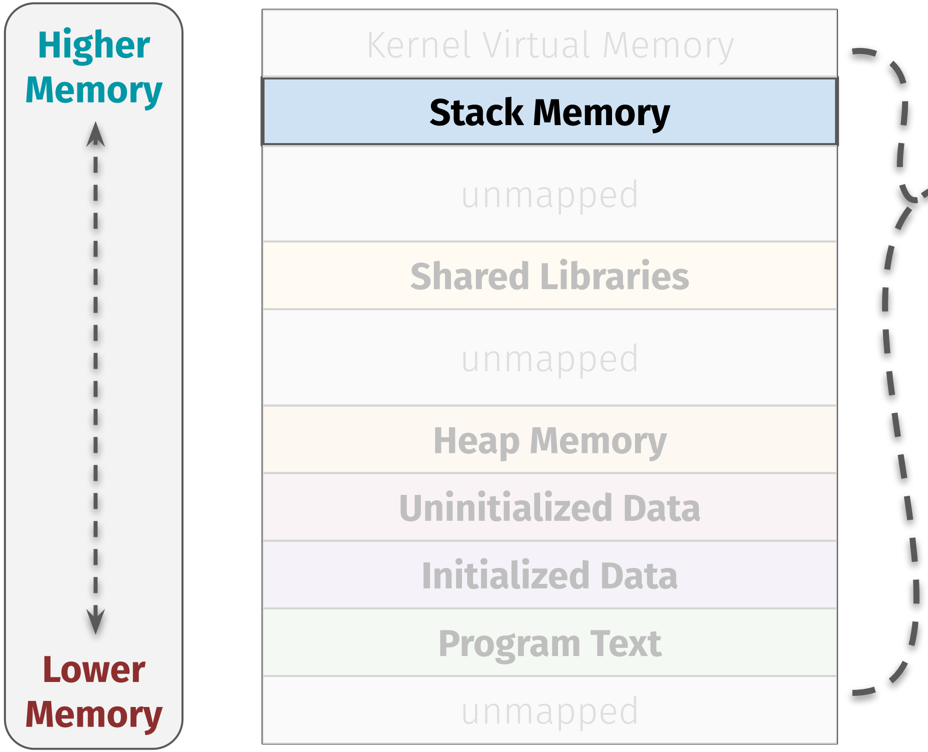
Key idea: it's all **"things" pointed to** by **addresses**

Example: instructions in the **Program Text:**

```
$ disas vulnerable:

0x0804a17b <+0>:     endbr32
0x0804a17f <+4>:     push   %ebp
0x0804a180 <+5>:     mov    %esp,%ebp
0x0804a182 <+7>:     push   %ebx
```

Memory Addresses Point to Memory Slots



Key idea: it's all **"things" pointed to** by **addresses**

Example: payload NOPs in **Stack Memory**:

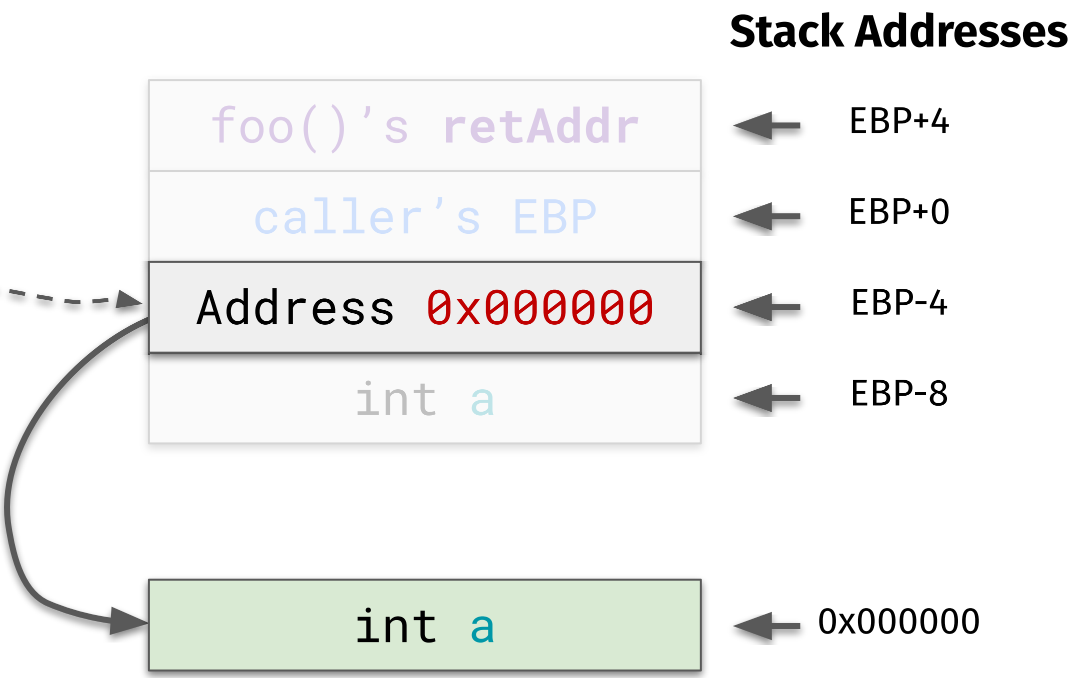
```
$ x/32xw 0xfff6d8cc

0xfff6d8cc:  0x90909090  0x90909090
0xfff6d8d4:  0x90909090  0x90909090
0xfff6d8dc:  0x90909090  0x90909090
0xfff6d8e4:  0x90909090  0x90909090
```

Indirect Memory Overwrite

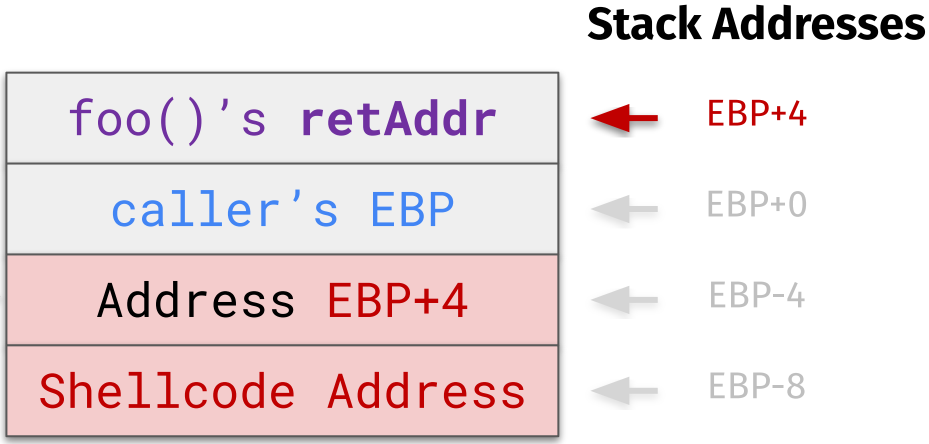
```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```

Contents of 0x000000 **updated to** a



Indirect Memory Overwrite

```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```



Indirect Memory Overwrite

```
void foo(char *str) {
    int *p;
    int  a;
    *p = a;
}
```

Stack Addresses

	Stack Addresses
Shellcode Address	EBP+4
caller's EBP	EBP+0
Address EBP+4	EBP-4
Shellcode Address	EBP-8

Contents of EBP+4 **updated to the** shellcode address**!**

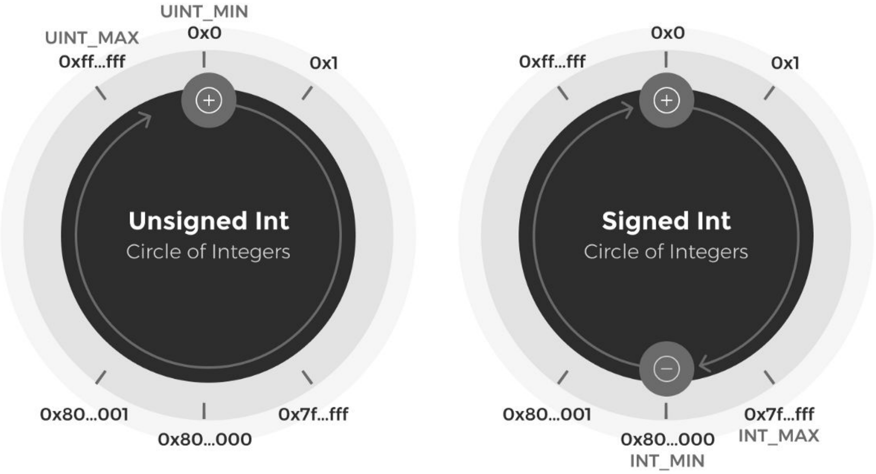
Integer Overflows

- **Integer overflows** behave differently from stack buffer overflows
  - Really just integer "**wrap-arounds**"

32-bit Integer Range:

Unsigned: [0, (2^32 - 1)]
[0, 4294967295]

Signed: [-2^31, (2^31 - 1)]
[-2147483648, 2147483647]



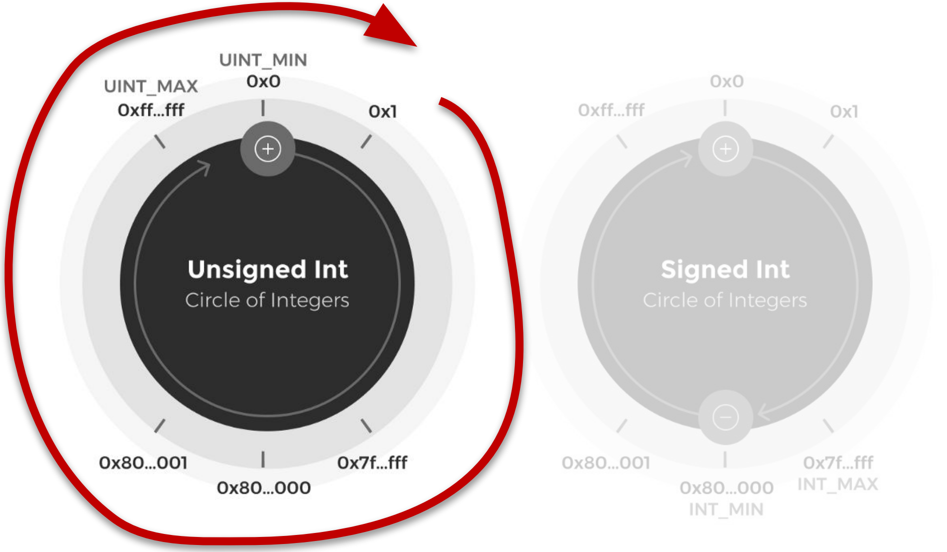
Integer Overflows

- **Integer overflows** behave differently from stack buffer overflows
  - Really just integer "**wrap-arounds**"

32-bit Integer Range:

Unsigned: `[0, (2^32 - 1)]`
`[0, 4294967295]`

Signed: `[-2^31, (2^31 - 1)]`
`[-2147483648, 2147483647]`



- **Overflowing an unsigned integer "wraps around" to a very small integer!**
  - E.g., `0xFFFFFFFF + 2 = 0x00000002`

Overcoming Bounded Writes

- **What observations can we make?**
  - Can they break the program's assumptions?

- **Target 4:** a **potential mismatch** of **buffer's size** versus the **data written to it**

```
alloca( <MAX_UINT );    // allocate our buffer
fread( &buf[i], 4, count, f ); // fill buffer
```

Range of count:

[0, ¼(MAX_UINT))

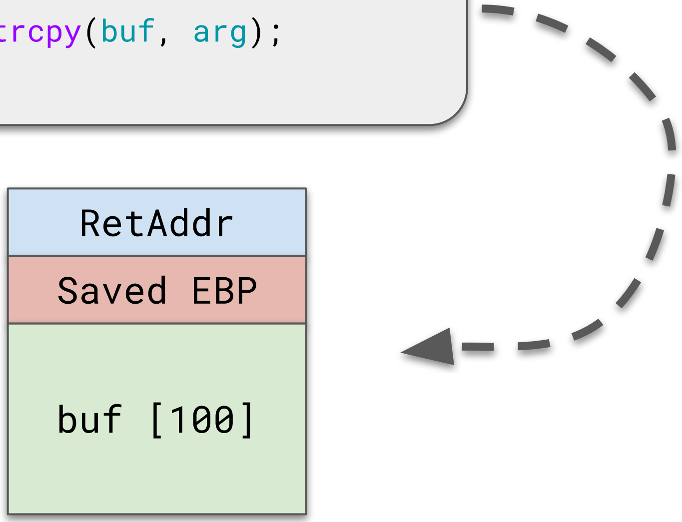
[0, MAX_UINT)

  - If we perform an **integer overflow** on count, `alloca()` creates an **artificially small** buffer
  - The resulting fill operation will **exceed the buffer's size**, resulting in a buffer overflow!

Estimating the Stack

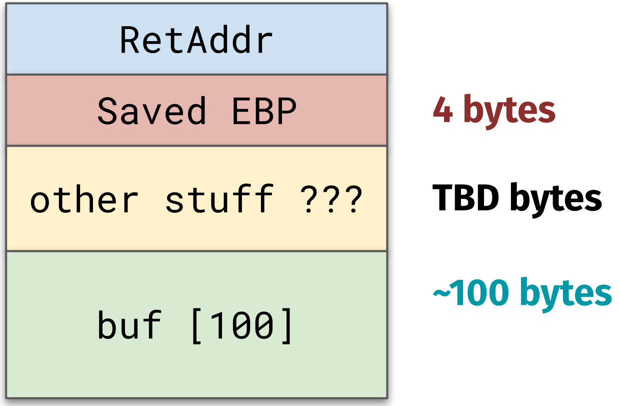
- **Identify your target function**
  - E.g., `vulnerable()` in this case
- **Each frame contains a few key things:**
  1. The function's return address
     - Address of next instruction to when the current function returns
  2. The caller's saved frame pointer
     - Where EBP will get "reset" to when the current function returns
  3. The function's local variables
     - E.g., `char buf[100]`
     - **Find these from the source code!**

```
void vulnerable(char *arg){
        char buf[100];
        strcpy(buf, arg);
}
```



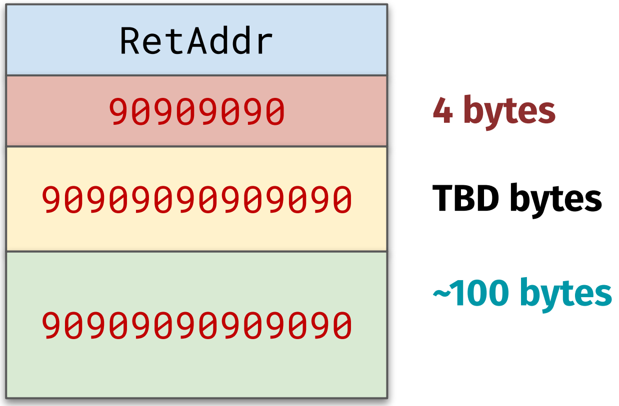
Padding Heuristics

- **How large** is our vulnerable buffer?
  - E.g., `char buf[100]`
  - Need **at least 100 bytes** to overflow!
    - Compilers may add **a few "extra" bytes** for memory alignment
- **Saved EBP** = an extra **four bytes**
- **Other things above our buffer?**
  - Other locals (e.g., `count` in Target 3)
  - Passed-by-reference function args
  - Other compiler-added artifacts



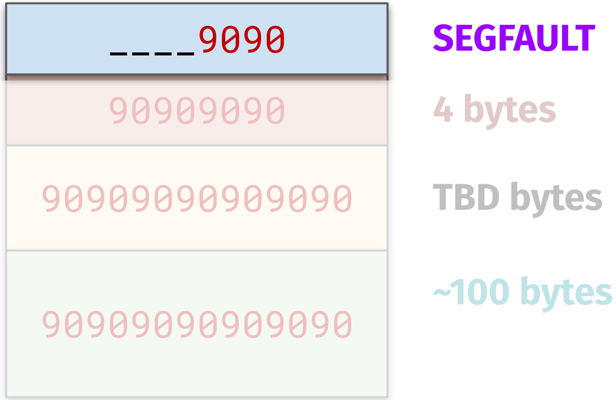
Write an Initial Payload

- Use guesstimated payload bytes as **lower bound** for an initial attempt
  - E.g., we know our payload is **104+ bytes**
- **Goal:** overwrite the return address with a **controlled**, **friendly** **payload**
  - E.g., **104 bytes** of NOP instructions
- **Did it overwrite the return address?**
  - If **yes**—**SEGFAULT** on `0x90909090`
  - If **not**—program terminates gracefully



Refine your Payload

- **Keep a table** of attempts and results
  1. `b'\x90' * 104 → normal exit`
     - **Too little!** Didn't overwrite anything
  2. `b'\x90' * 120 → SEGV on 0x90909090`
     - **Too much!** Complete RetAddr overwrite
  3. `b'\x90' * 114 → SEGV on 0x08049090`
     - **We're close—just two bytes over!**
     - Our payload should be **112 bytes**



Tweak it to figure out the **exact payload size**

Find the Buffer!

- After finding the distance to the return address, we now must **overwrite it**
  - **Recall:** the return address is our golden ticket to **controlling the program's execution**
  - Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

- **Approach:** pick a **known, friendly payload** and locate it in memory
  - Goal is to find **the start of your buffer!**

- **Helpful GDB commands:**
  - `info proc mapping`
    - Locate the stack's **boundaries**
    - E.g., `0xfff6d000` to `0xffffe000`

```
$ info proc mapping // list all memory segments
Start Addr   End Addr       Size     Offset objfile
0x8048000  0x8049000     0x1000        0x0 target2
0x8049000  0x80b8000    0x6f000     0x1000 target2
0x80b8000  0x80e8000    0x30000    0x70000 target2
0x80e8000  0x80ea000     0x2000    0x9f000 target2
0x80ea000  0x80ec000     0x2000    0xa1000 target2
0x80ec000  0x810e000    0x22000        0x0 [heap]
0xf7ff8000 0xf7ffc000     0x4000        0x0 [vvar]
0xf7ffc000 0xf7ffe000     0x2000        0x0 [vdso]
0xfff6d000 0xffffe000    0x91000        0x0 [stack]
```

Find the Buffer!

- After finding the distance to the return address, we now must **overwrite it**
  - **Recall:** the return address is our golden ticket to **controlling the program's execution**
  - Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

- **Approach:** pick a **known, friendly payload** and locate it in memory
  - Goal is to find **the start of your buffer!**

- **Helpful GDB commands:**
  - `find minAddr,maxAddr,"string"`
    - Search memory for address of `string`
    - Use **stack boundaries** from before

```
$ b *vulnerable+45 // breakpoint after buf filled
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ r "AAAA" // run program with "AAAA" as its input
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ find 0xfff6d000,0xffffe000,"AAAA"
0xfff6d8cc // this is likely where buffer begins!
0xfffed930 // when in doubt, pick the lower address
```

Find the Buffer!

- After finding the distance to the return address, we now must **overwrite it**
  - **Recall:** the return address is our golden ticket to **controlling the program's execution**
  - Instead of a normal return, we want to **redirect execution** to our **shellcode-laden buffer**

- **Approach:** pick a **known, friendly payload** and locate it in memory
  - Goal is to find **the start of your buffer!**

- **Helpful GDB commands:**
  - `x/32xw, 0xDEADBEEF`
    - Show bytes at address `0xDEADBEEF`
    - **Inspect candidates** from previous step

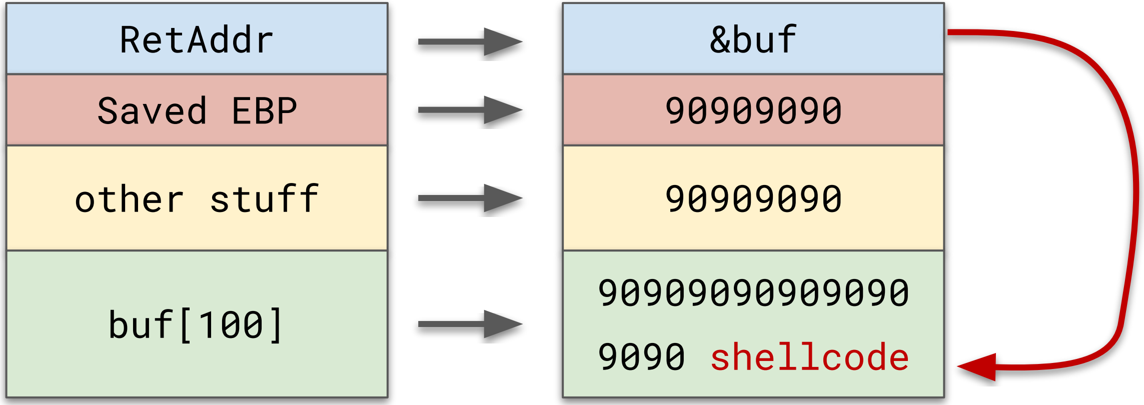
```
$ b *vulnerable+45 // breakpoint after buf filled
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ r "AAAA" // run program with "AAAA" as its input
Breakpoint 1, 0x0804a1a8 in vulnerable… target2.c:8

$ x/32xw 0xfff6d8cc // look for "AAAA" bytes here
0xfff6d8cc: 0x41414141 0x00000000 0x00000000 ...
0xfff6d8d0: 0x00000000 0x00000000 0x00000000 ...
```

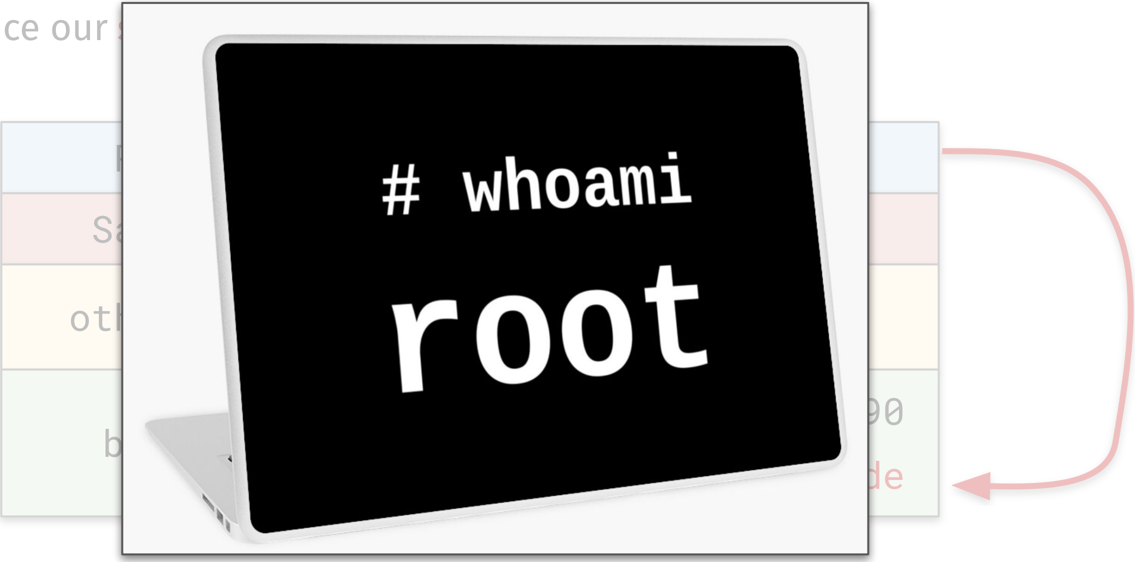
We're almost there!

- **By this point**, we've identified our **padding length** and **buffer start address**
  - Now, introduce our **shellcode** and finalize the attack payload!



We’re almost there!

- **By this point**, we’ve identified our **padding length** and **buffer start address**
  - Now, introduce our



Other Exploitation Techniques

- **Not just return addresses!**
  - Function pointers
  - Arbitrary data
  - C++ exceptions
  - C++ objects
  - Heap memory freelist
  - **Any code pointer!**

Quiz Question Recap

```
0x0804a014 <+00>:    push    %ebp
0x0804a015 <+01>:    mov     %esp, %ebp
0x0804a017 <+03>:    sub     $4, %esp
0x0804a01a <+06>:    mov     16(%ebp), %eax
```

Quiz Question Recap

```
0x0804a014 <+00>:    push    %ebp
0x0804a015 <+01>:    mov     %esp, %ebp
0x0804a017 <+03>:    sub     $4, %esp
0x0804a01a <+06>:    mov     16(%ebp), %eax
```

Registers

EIP	0x0804a014
EBP	0xbffff440
ESP	0xbffff400

Stack Diagram

0xbffff400	Return Address	← **SP**
0xbffff3fc		
0xbffff3f8		

Quiz Question Recap

```
0x0804a014 <+00>:   push   %ebp
0x0804a015 <+01>:   mov    %esp, %ebp
0x0804a017 <+03>:   sub    $4, %esp
0x0804a01a <+06>:   mov    16(%ebp), %eax
```

Registers

EIP	0x0804a014
EBP	0xbffff440
ESP	0xbffff**3fc**

Stack Diagram

0xbffff400	Return Address
0xbffff3fc	Saved EBP ← SP
0xbffff3f8	

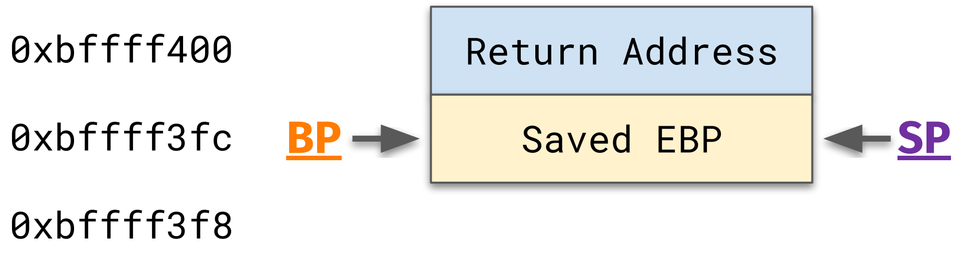
Quiz Question Recap

```
0x0804a014 <+00>:    push    %ebp
0x0804a015 <+01>:    mov     %esp, %ebp
0x0804a017 <+03>:    sub     $4, %esp
0x0804a01a <+06>:    mov     16(%ebp), %eax
```

Registers

EIP	0x0804a014
EBP	0xbffff3fc
ESP	0xbffff3fc

Stack Diagram



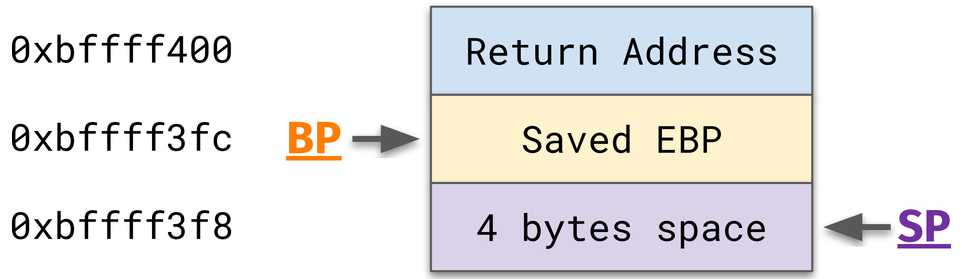
Quiz Question Recap

```
0x0804a014 <+00>:    push    %ebp
0x0804a015 <+01>:    mov     %esp, %ebp
0x0804a017 <+03>:    sub     $4, %esp
0x0804a01a <+06>:    mov     16(%ebp), %eax
```

Registers

EIP	0x0804a014
EBP	0xbffff3fc
ESP	0xbffff3f8

Stack Diagram



Quiz Question Recap

```
0x0804a014 <+00>:    push    %ebp
0x0804a015 <+01>:    mov     %esp, %ebp
0x0804a017 <+03>:    sub     $4, %esp
0x0804a01a <+06>:    mov     16(%ebp), %eax
```

Registers

EIP	0x0804a014
EBP	0xbffff3fc
ESP	0xbffff3f8

Stack Diagram

0xbffff400		Return Address	
0xbffff3fc	BP →	Saved EBP	
0xbffff3f8		4 bytes space	← SP

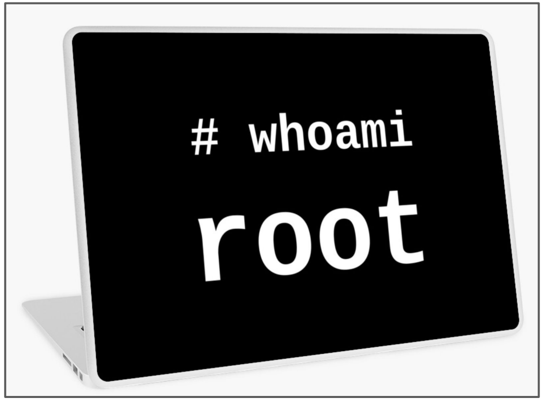
Questions?

This time on CS 4440…

Advanced Exploitation Techniques
ASLR, DEP, and Workarounds
Other Application-level Defenses

Recap: Spawning Shells

- **Attacker goal:** make program open a **root shell**
  - Root-level permissions = **total system ownage**
  - **You'll do this in Project 2!**



- **Shellcode** = code to open a root shell
  - Inject this somewhere and **direct execution to it**
  - Basic structure:
    1. Call `setuid(0)` to set user ID to "`root`"
    2. Open a shell with `execve("/bin/sh")`

`setuid(0)` + `execve("/bin/sh")`

Shell Spawning in C

```c
#include <stdio.h>


void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
```

Shell Spawning in C

```
#include <stdio.h>


void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
```

`execve()`: execute a program: the text, data, bss, and stack of calling process are **overwritten** by that of the program loaded

Shell Spawning in C

```
#include <stdio.h>


void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
```

`execve()`: execute a program: the text, data, bss, and stack of calling process are **overwritten** by that of the program loaded

`/bin/sh`: a shell program

Shell Spawning in C

```c
#include <stdio.h>


void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
```

Shell inherits same **privileges** as the original "parent" process

Shell Spawning in C

```
#include <stdio.h>


void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
```

Shell inherits same **privileges** as the original “parent” process

If the original process **run as root**, shell gives ???? access

Shell Spawning in C

```
#include <stdio.h>


void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
```

Shell inherits same **privileges** as the original “parent” process

If the original process **run as root**, shell gives **root** access

Shell Spawning in C

```
#include <s

void main()
    char *a
    argv[0]
    execve(
}
```

privileges
nt" process

ss **run as**
ot access

Shell Spawning in x86 Assembly

```
main:
     pushl   %ebp
     movl    %esp, %ebp
     pushl   $0
     pushl   $0
     pushl   $.LC0
     call    execve
     leave
     ret
```

Shell Spawning in x86 Assembly

```
main:
      pushl   %ebp
      movl    %esp, %ebp
      pushl   $0
      pushl   $0
      pushl   $.LC0
      call    execve
      leave
      ret
```

Like before, we want to call **execve("/bin/sh")**

Shell Spawning in x86 Assembly

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   $0
    pushl   $0
    pushl   $.LC0
    call    execve
    leave
    ret
```

Like before, we want to call **`execve("/bin/sh")`**

Q: How does the stack need to look for this call to work?

Invoking a Shell

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $0
        pushl   $0
        pushl   $.LC0
        call    execve
        leave
        ret
```

main()'s **locals**
??????????????????
??????????????????
??????????????????

Invoking a Shell

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $0
        pushl   $0
        pushl   $.LC0
        call    execve
        leave
        ret
```

main()'s **locals**

execve()'s **3rd arg**

???????????????????

???????????????????

Invoking a Shell

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $0
        pushl   $0
        pushl   $.LC0
        call    execve
        leave
        ret
```

main()'s **locals**
arg3 = **NULL**
????????????????????
????????????????????

Invoking a Shell

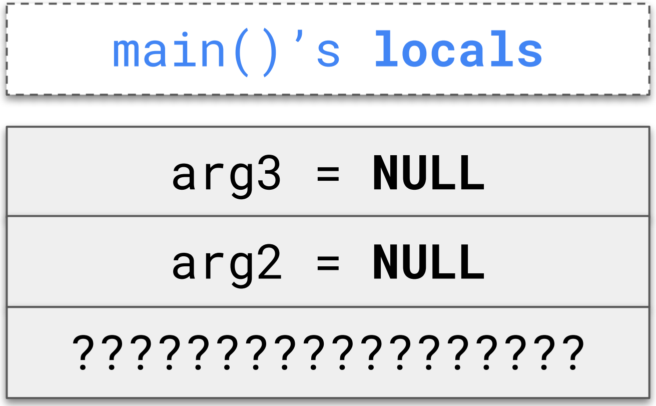
```
main:
      pushl   %ebp
      movl    %esp, %ebp
      pushl   $0
      pushl   $0
      pushl   $.LC0
      call    execve
      leave
      ret
```

main()'s **locals**

arg3 = **NULL**
execve()'s **2nd arg**
??????????????????

Invoking a Shell

```
main:
      pushl   %ebp
      movl    %esp, %ebp
      pushl   $0
      pushl   $0
      pushl   $.LC0
      call    execve
      leave
      ret
```



Invoking a Shell

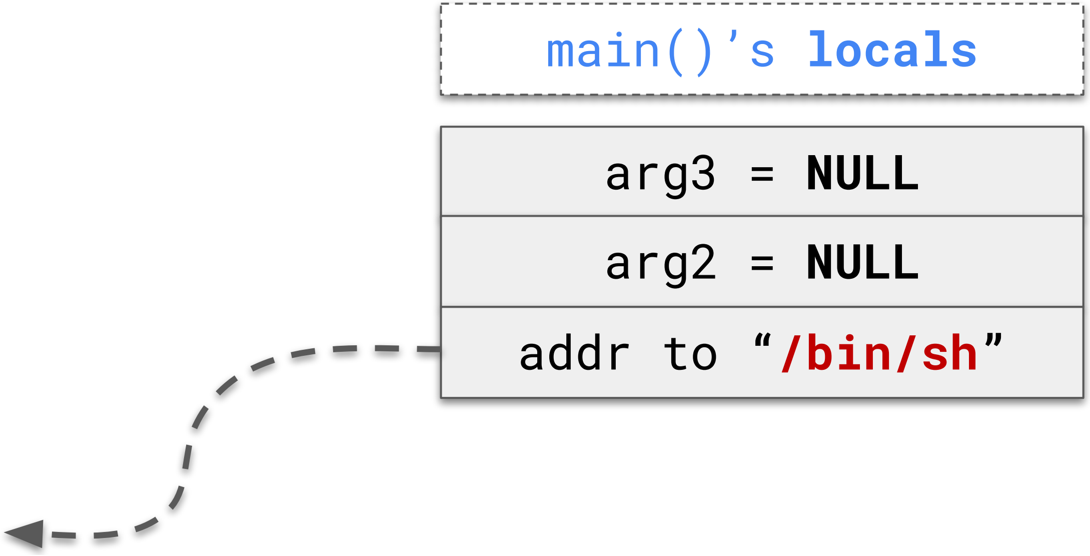
```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $0
        pushl   $0
        pushl   $.LC0
        call    execve
        leave
        ret
```

main()'s **locals**
arg3 = **NULL**
arg2 = **NULL**
execve()'s **1st arg**

Invoking a Shell

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $0
        pushl   $0
        pushl   $.LC0
        call    execve
        leave
        ret

.LC0:
        .string "/bin/sh"
```



Invoking a Shell

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $0
        pushl   $0
        pushl
        call
        leav
        ret

.LC0:
        .string "/bin/sh"
```

execve("/bin/sh", NULL, NULL);

main()'s **locals**
arg3 = **NULL**
arg2 = **NULL**
addr to "**/bin/sh**"
execve()'s **ret addr**

Invoking a Shell

```
main:
      pushl   %ebp
      movl    %esp, %ebp
      pushl   $0
      pushl   $0
      pushl
      call
      leav
      ret

.LC0:
      .string "/bin/sh"
```

main()'s locals

NULL

NULL

/bin/

execve()'s ret

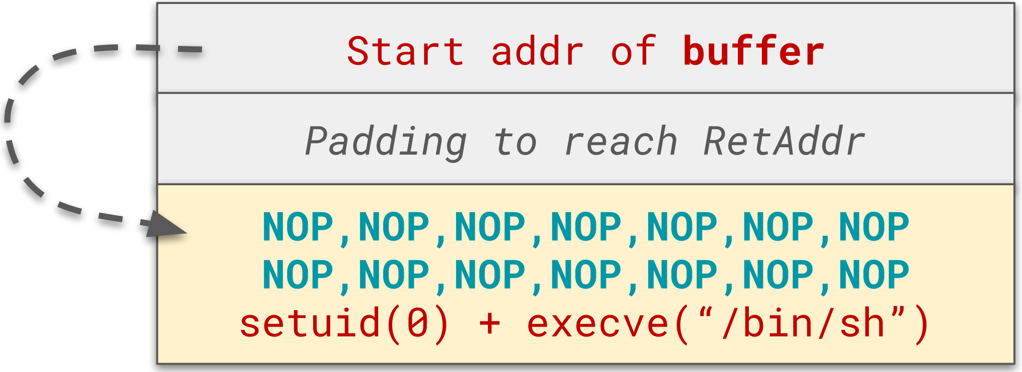
execve("/bin/s

How can we **prevent** **code injection** attacks?

Application Defense: Address Space Layout Randomization

Caveats

- Our provided shellcode requires an **executable buffer**

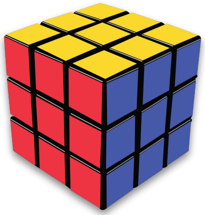


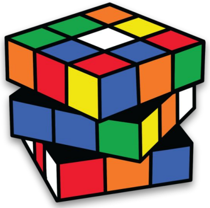
Caveats

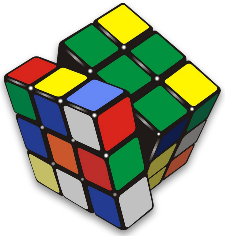
- Our provided shellcode requires an **executable buffer**
- **What if the buffer is relocated on every new run?**

Defense: ASLR

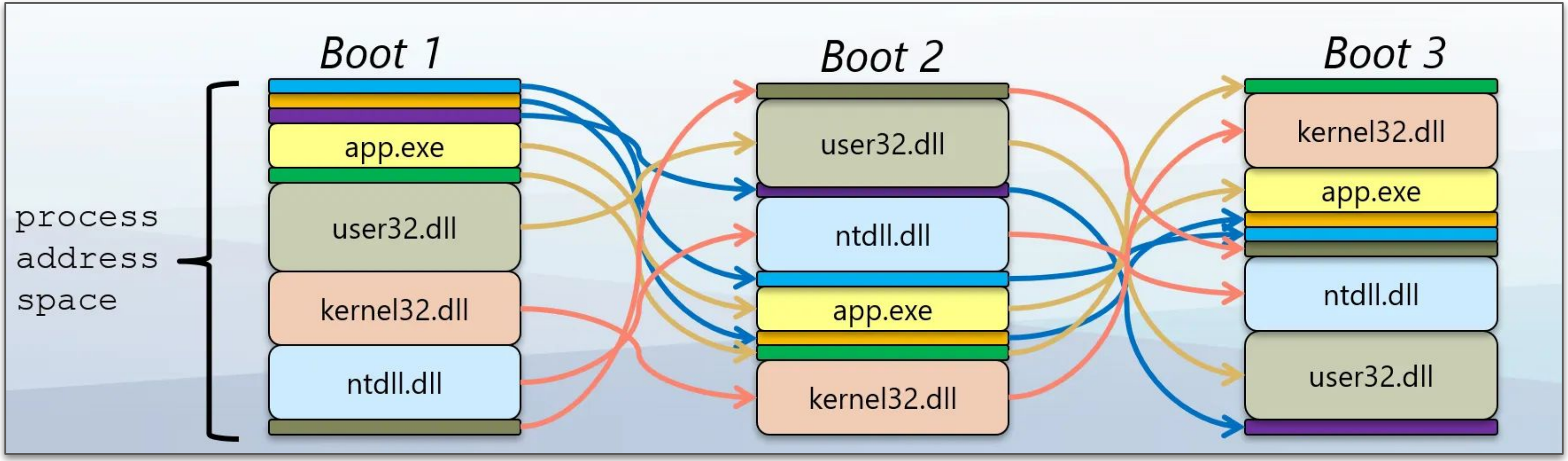
- **Address Space Layout Randomization**
  - One of the most common defenses today
- Changes **location of stack** on each execution
  - As well as other memory areas (the heap, libc, etc.)
- Makes buffer overflows significantly harder
  - Can’t “hardcode” address of buffer’s start
  - … it changes every time!





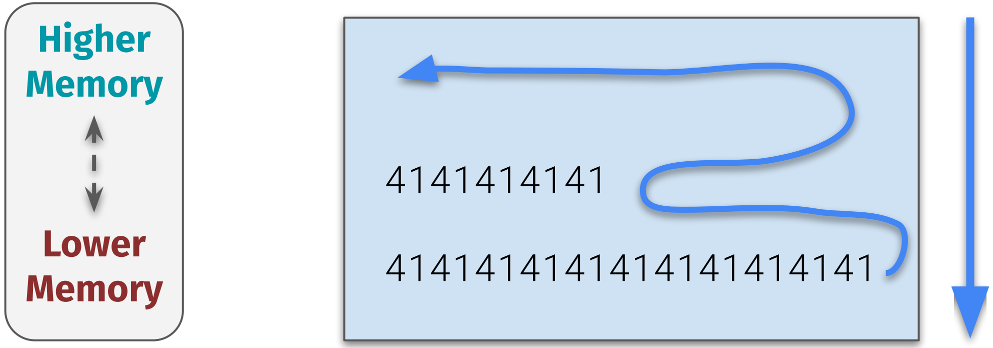


Defense: ASLR



- **How can we overcome ASLR?**

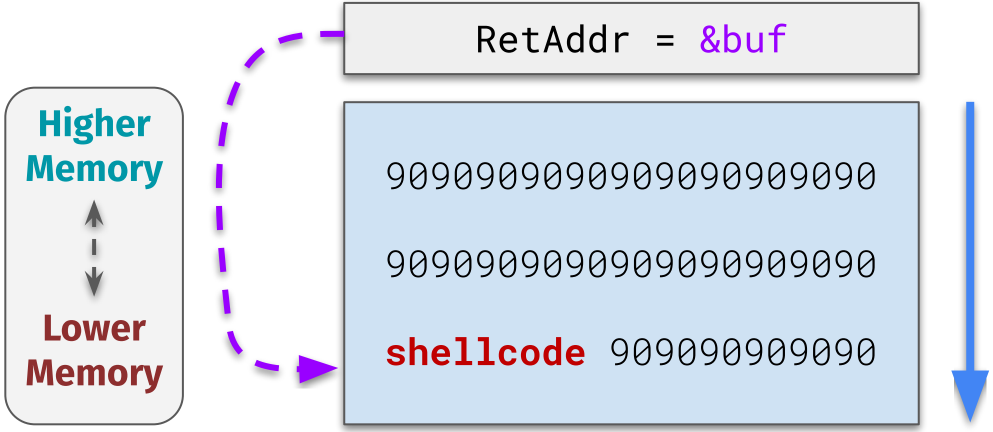
Recap: Stack Growth vs. Filling



Stack grows **downwards**

- Filled upwards

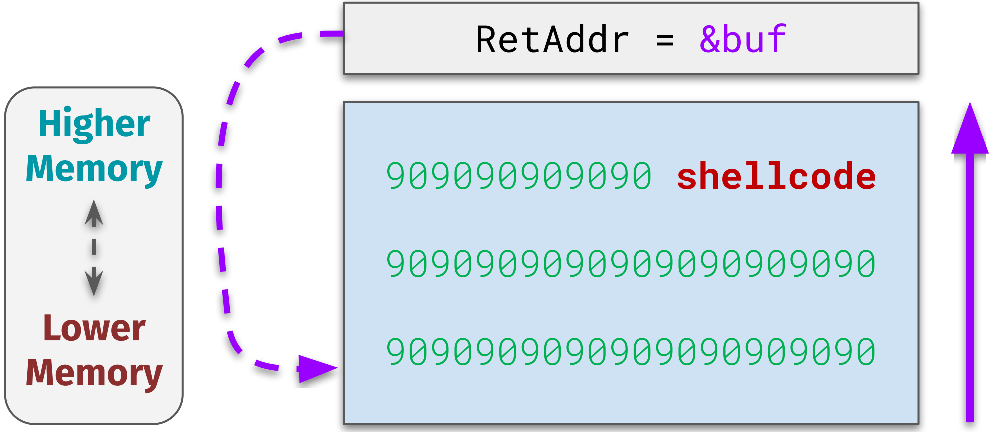
Recap: Redirection to Buffer



Stack grows **downwards**
- Filled upwards

Payload = shellcode + NOPs + &buf

Workaround: NOP Slide!

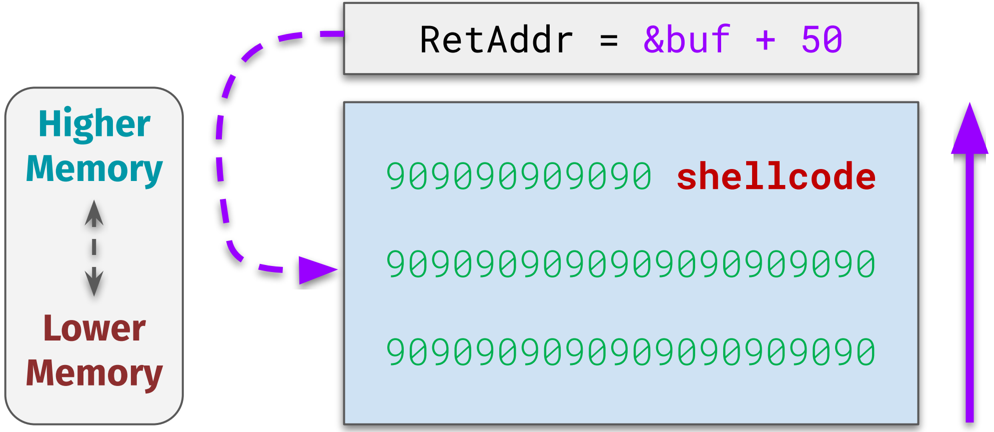


Stack grows **downwards**
- Filled upwards

Execution moves **upwards**
- Lower to higher instructions

`Payload = NOPs + shellcode + &buf`

Workaround: NOP Slide!



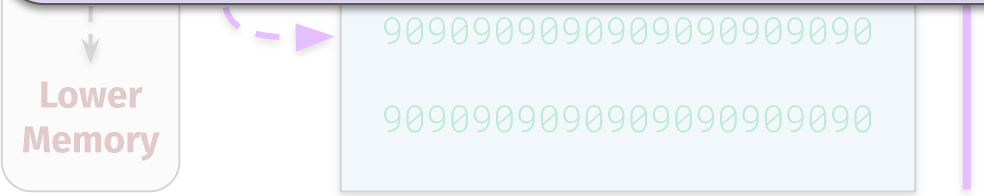
Stack grows **downwards**
- Filled upwards

Execution moves **upwards**
- Lower to higher instructions
- **NOP slide leverages this!**

```
Payload = NOPs + shellcode + (&buf + 50)
```

Workaround: NOP Slide!

We can't reliably guess the **buffer's start**—it **changes** every execution!



Execution moves **upwards**
- Lower to higher instructions
- **NOP slide leverages this!**

Payload = NOPs + shellcode + (&buf + 50)

Workaround: NOP Slide!

We can't reliably guess the **buffer's start**—it **changes** every execution!

But, if we prepended our shellcode with **a huge NOP slide**, jumping to **the middle of it** it will "slide" to our **shellcode**!

Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**

```
setuid(0) + execve("/bin/sh")
 NOP,NOP,NOP,NOP,NOP,NOP,NOP
 NOP,NOP,NOP,NOP,NOP,NOP,NOP
 NOP,NOP,NOP,NOP,NOP,NOP,NOP
 NOP,NOP,NOP,NOP,NOP,NOP,NOP
 NOP,NOP,NOP,NOP,NOP,NOP,NOP
```

Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**

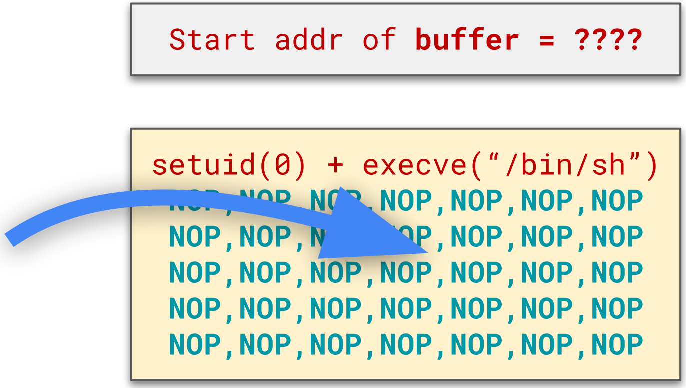
- We cannot know buffer's **exact start...**

```
Start addr of buffer = ????
```

```
setuid(0) + execve("/bin/sh")
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
```

Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**
- We cannot know buffer's **exact start…**
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all



Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**
- We cannot know buffer's **exact start…**
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all
- **Idea:** spam **"guessed" buffer addr** up the stack

Guessed addr within **buffer**
Guessed addr within **buffer**
Guessed addr within **buffer**
Guessed addr within **buffer**
Guessed addr within **buffer**

```
setuid(0) + execve("/bin/sh")
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
```

Defeating ASLR

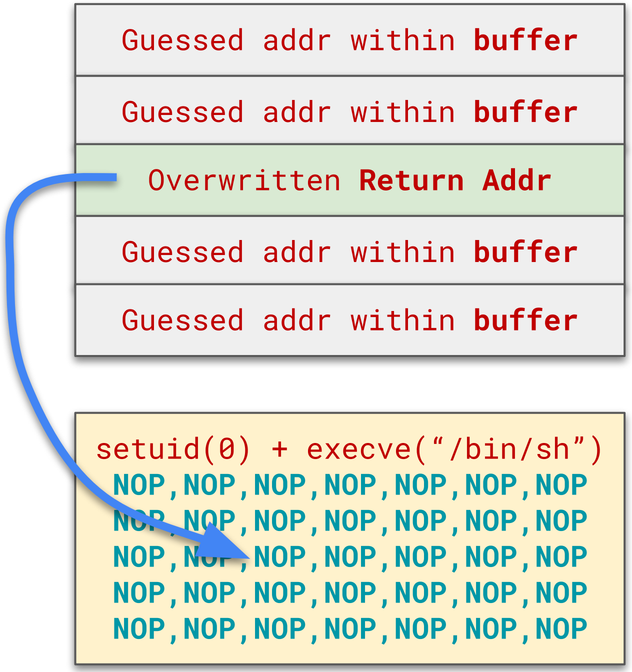
- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**
- We cannot know buffer's **exact start…**
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all
- **Idea:** spam **"guessed" buffer addr** up the stack
  - Eventually we'll overwrite some **return address**

```
Guessed addr within buffer
Guessed addr within buffer
Overwritten Return Addr
Guessed addr within buffer
Guessed addr within buffer
```

```
setuid(0) + execve("/bin/sh")
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP,NOP,NOP,NOP
```

Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**
- We cannot know buffer's **exact start…**
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all
- **Idea:** spam **"guessed" buffer addr** up the stack
  - Eventually we'll overwrite some **return address**
  - When that function returns, jump inside buffer
  - **Hit the huge NOP sled → BOOM!**

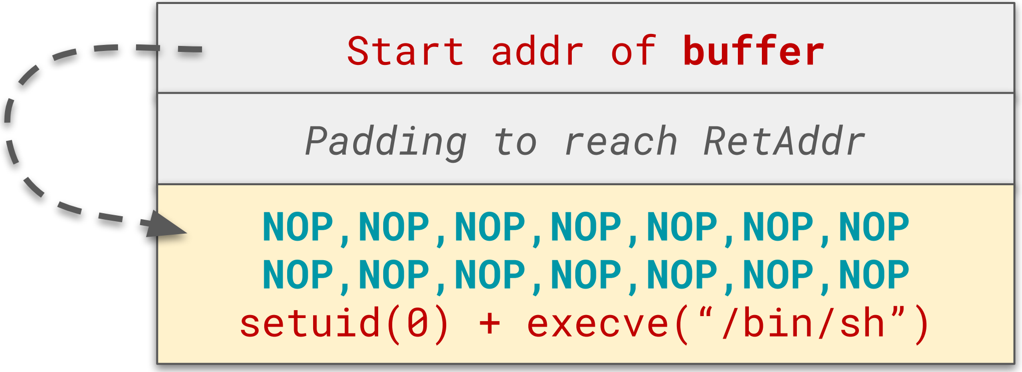


Questions?

Application Defense:
Data Execution Prevention

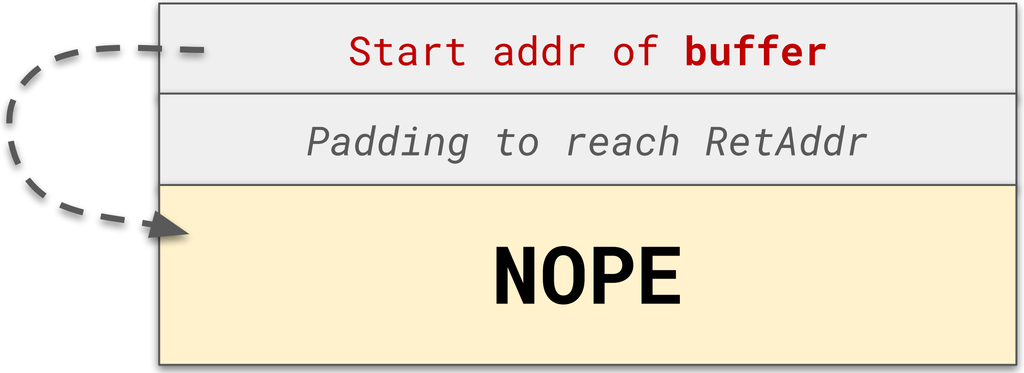
Caveats

- Our provided shellcode requires an **executable buffer**



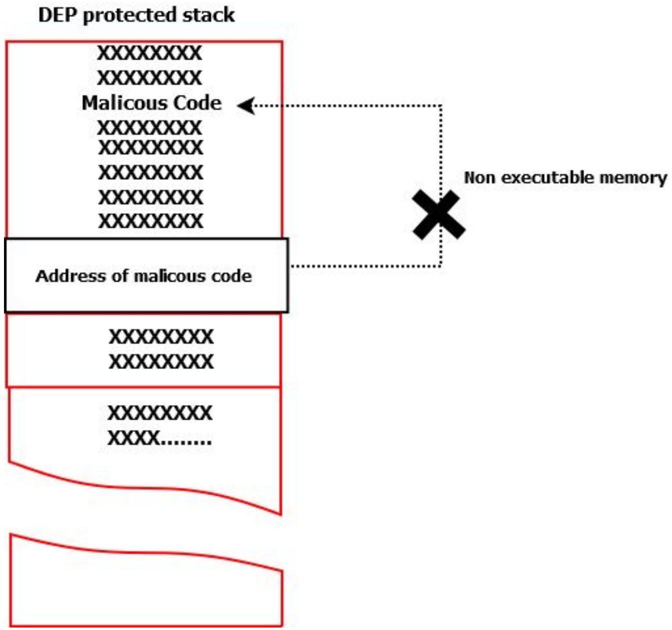
Caveats

- Our provided shellcode requires an **executable buffer**
- **What if the buffer is prohibited from being executable?**



Defense: DEP

- **Data Execution Prevention**
  - Aka **Non-eXecutable** (NX) **Stack**
  - Another common defense seen today
- Attacker can't execute **code on stack**
  - Mark pages as EITHER (never both)
  - **Read OR write** (stack/heap)
  - Executable (.text/code segments)
- **Challenges:**
  - Self-modifying code, JIT compilation
  - Requires hardware support (MMU/MPU)



Defeating DEP

- Suppose we can still overwrite buffer
  - We **cannot** place our shellcode there
  - But, we can **overwrite other stack items**

- Suppose the program calls a function that can **execute arbitrary commands**
  - execve()
  - system()

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   "/bin/ls"
    call    system
    leave
    ret
```

Dangerous Calls

- **Why are functions like `execve()` and `system()` considered dangerous?**

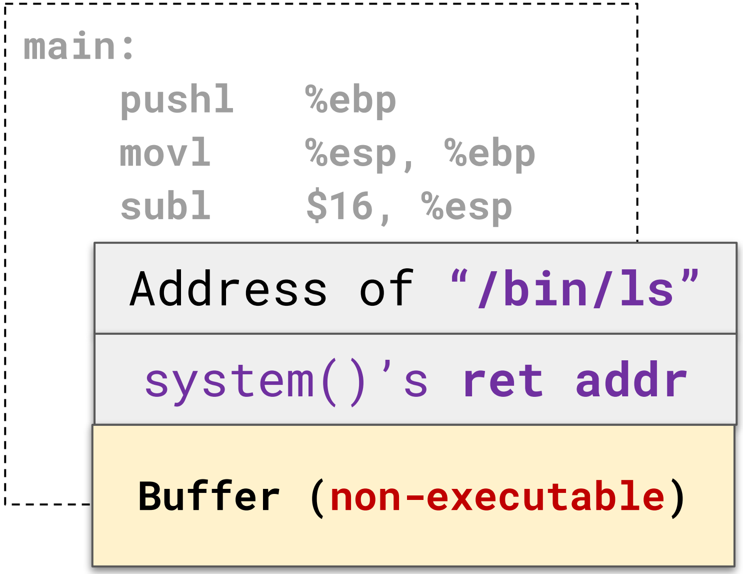
Dangerous Calls

- **Why are functions like `execve()` and `system()` considered dangerous?**

> Use of the `system()` function can result in exploitable vulnerabilities, in the worst case allowing execution of arbitrary system commands. Situations in which calls to `system()` have high risk include the following:
>
> - When passing an unsanitized or improperly sanitized command string originating from a tainted source
> - If a command is specified without a path name and the command processor path name resolution mechanism is accessible to an attacker
> - If a relative path to an executable is specified and control over the current working directory is accessible to an attacker
> - If the specified executable program can be spoofed by an attacker
>
> Do not invoke a command processor via `system()` or equivalent functions to execute a command.

Defeating DEP by Controlling Arguments

- Suppose we can still overwrite buffer
  - We **cannot** place our shellcode there
  - But, we can **overwrite other stack items**
- Suppose the program calls a function that can **execute arbitrary commands**
  - `execve()`
  - `system()`
- **Idea #1:** overwrite argument to `system()`
  - Replace it with our shell command (“`/bin/sh`”)



Defeating DEP by Controlling Arguments

- Suppose we can still overwrite buffer
  - We **cannot** place our shellcode there
  - But, we can **overwrite other stack items**
- Suppose the program calls a function that can **execute arbitrary commands**
  - `execve()`
  - `system()`
- **Idea #1:** overwrite argument to `system()`
  - Replace it with our shell command (“`/bin/sh`”)
  - Will now execute **`system(“/bin/sh”)`**!

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
```

Address of “/bin/sh”
system()’s ret addr
AAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAA

Defeating DEP by Controlling Arguments

- Suppose
  - We car
  - But, we
- Suppose
  that can e
  - execv
  - syste
- **Idea #1:** o
  - Replac
  - Will no

, %ebp
%esp
f "/bin/sh"
s ret addr
AAAAAAAAAAAA
AAAAAAAAAAAA

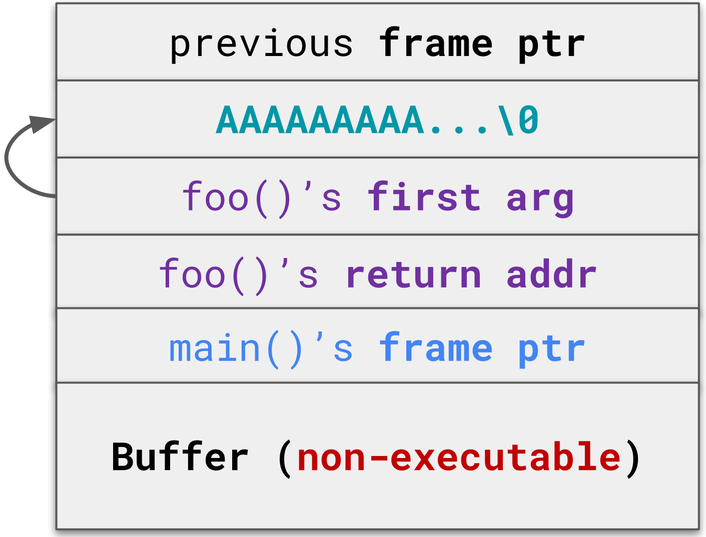
Defeating DEP via Code Reuse

- Suppose `system()` isn't executed, but **a call to it exists somewhere**
  - You can examine the **`objdump`** to look for "interesting" functions in the program

Defeating DEP via Code Reuse

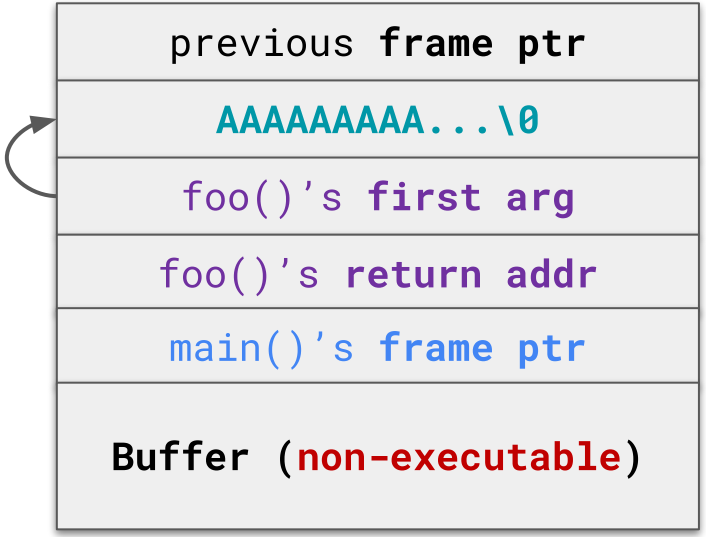
- Suppose system( ) isn't executed, but **a call to it exists somewhere**
  - You can examine the **objdump** to look for "interesting" functions in the program

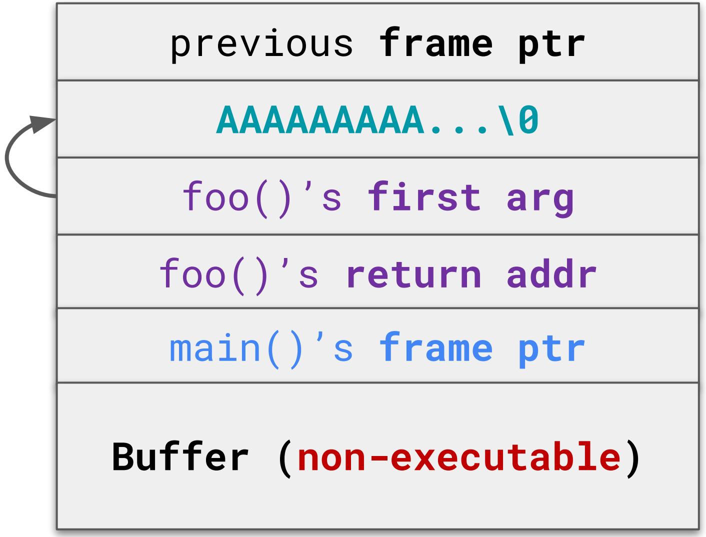
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str)
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```



Defeating DEP via Code Reuse

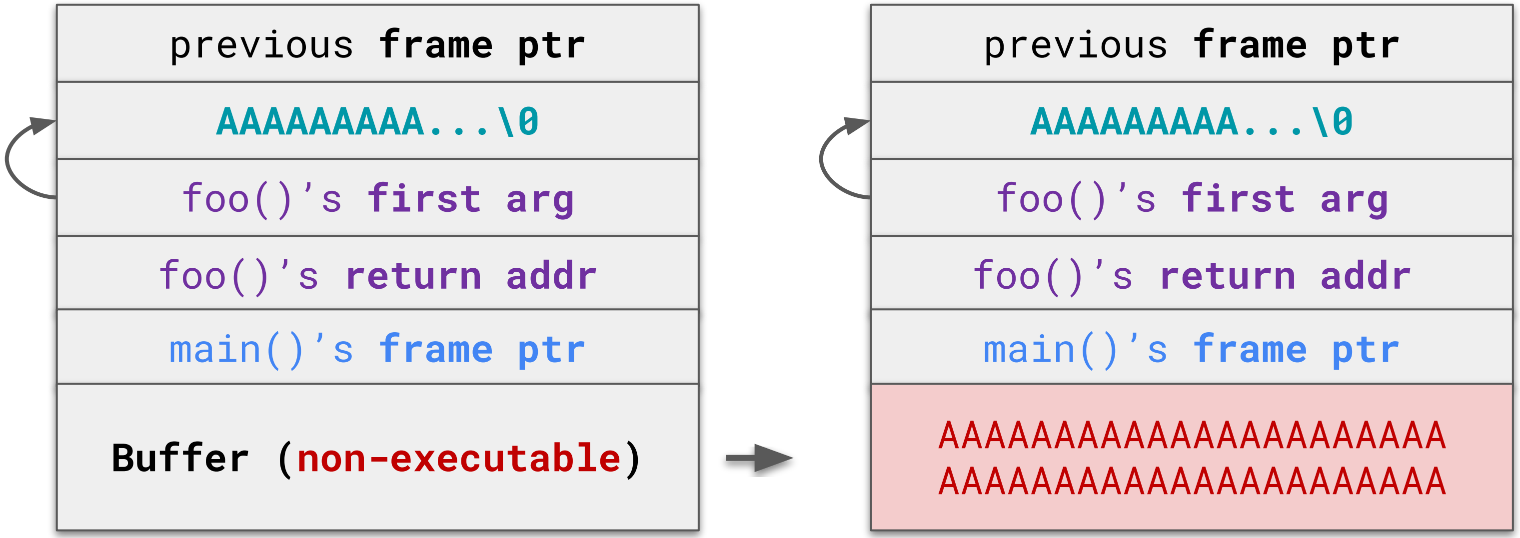
- **Idea #2:** create a **"fake" call frame for `system()`** with our desired arg





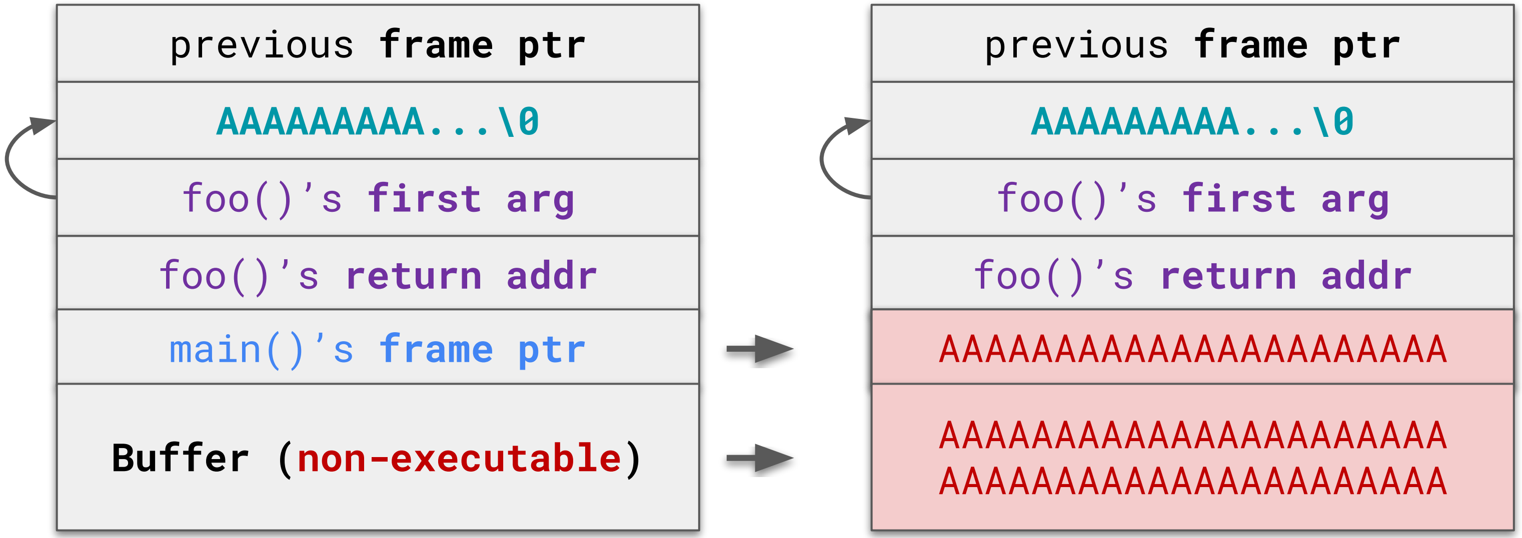
Defeating DEP via Code Reuse

- **Idea #2:** create a **"fake" call frame for `system()`** with our desired arg



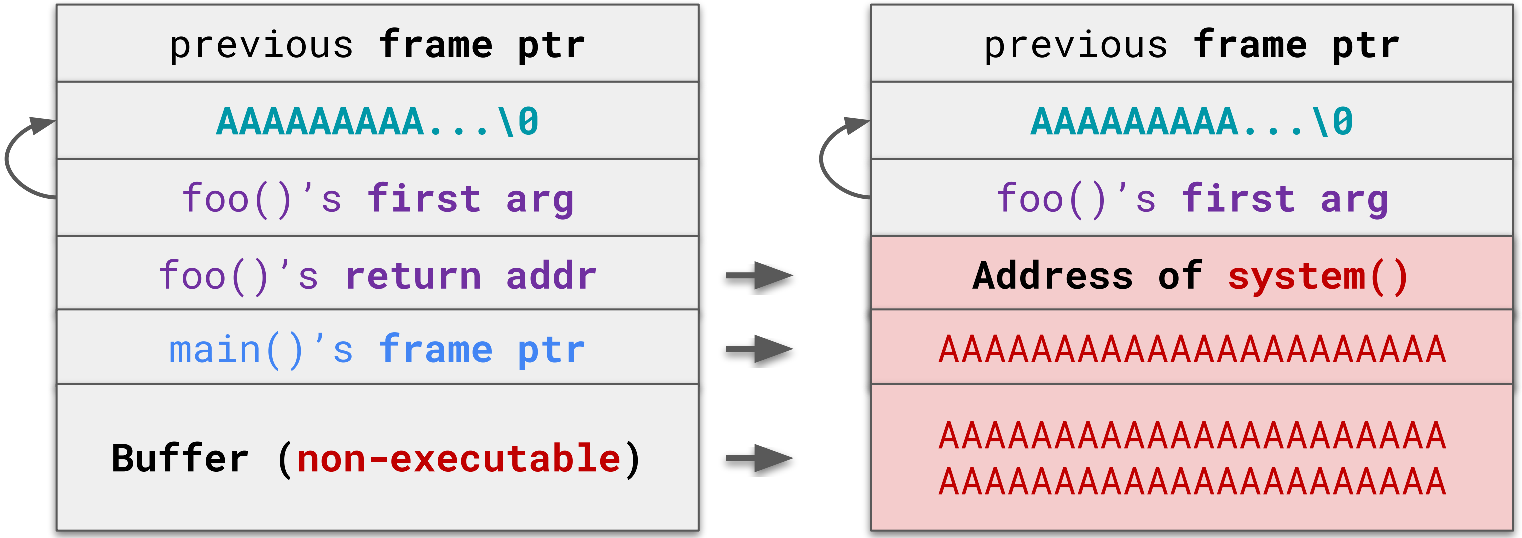
Defeating DEP via Code Reuse

- **Idea #2:** create a **"fake" call frame for `system()`** with our desired arg



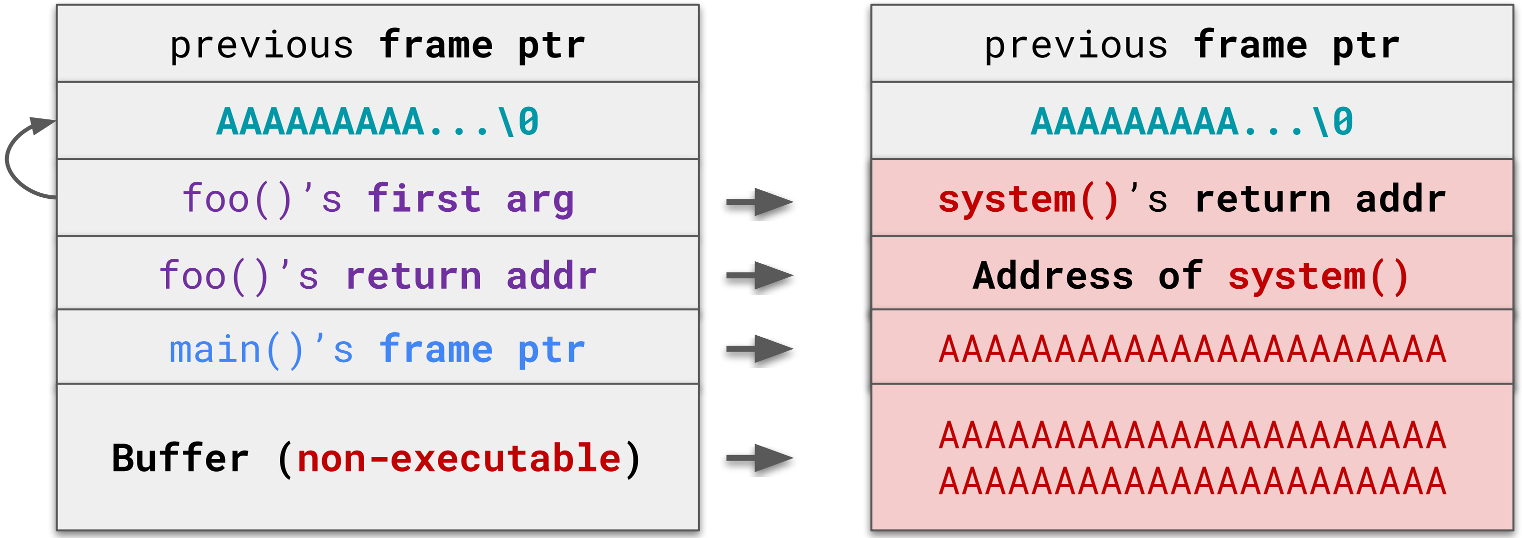
Defeating DEP via Code Reuse

- **Idea #2:** create a **"fake" call frame for `system()`** with our desired arg



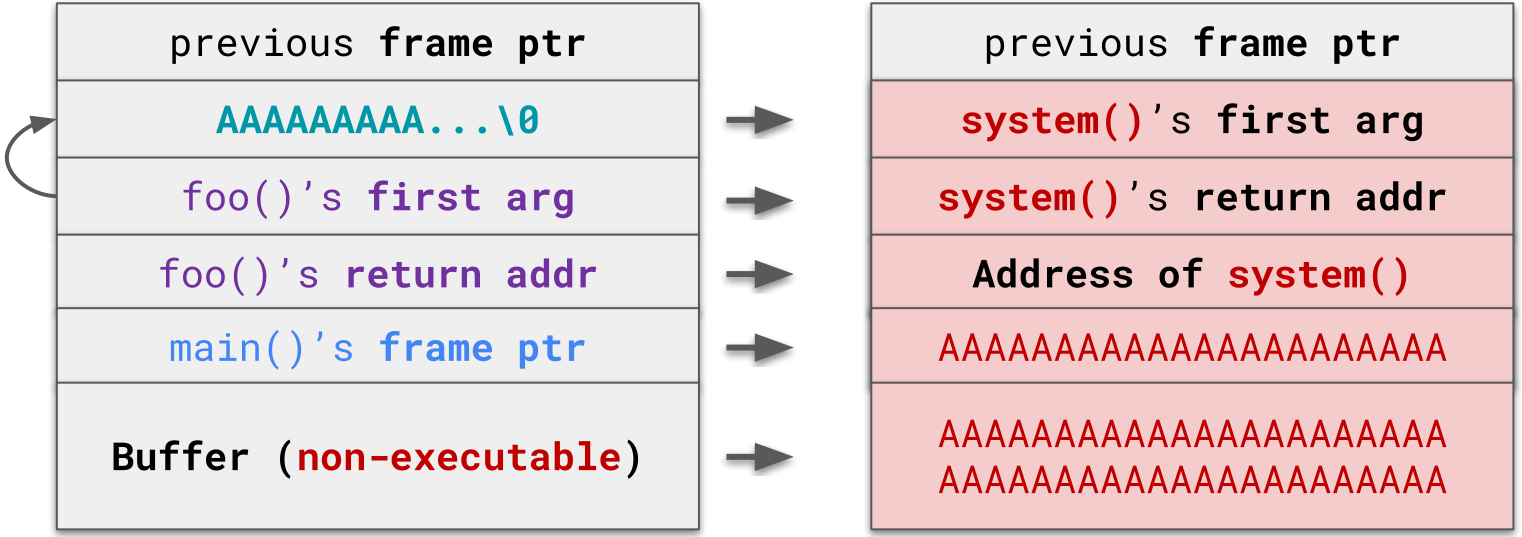
Defeating DEP via Code Reuse

- **Idea #2:** create a **"fake" call frame for `system()`** with our desired arg



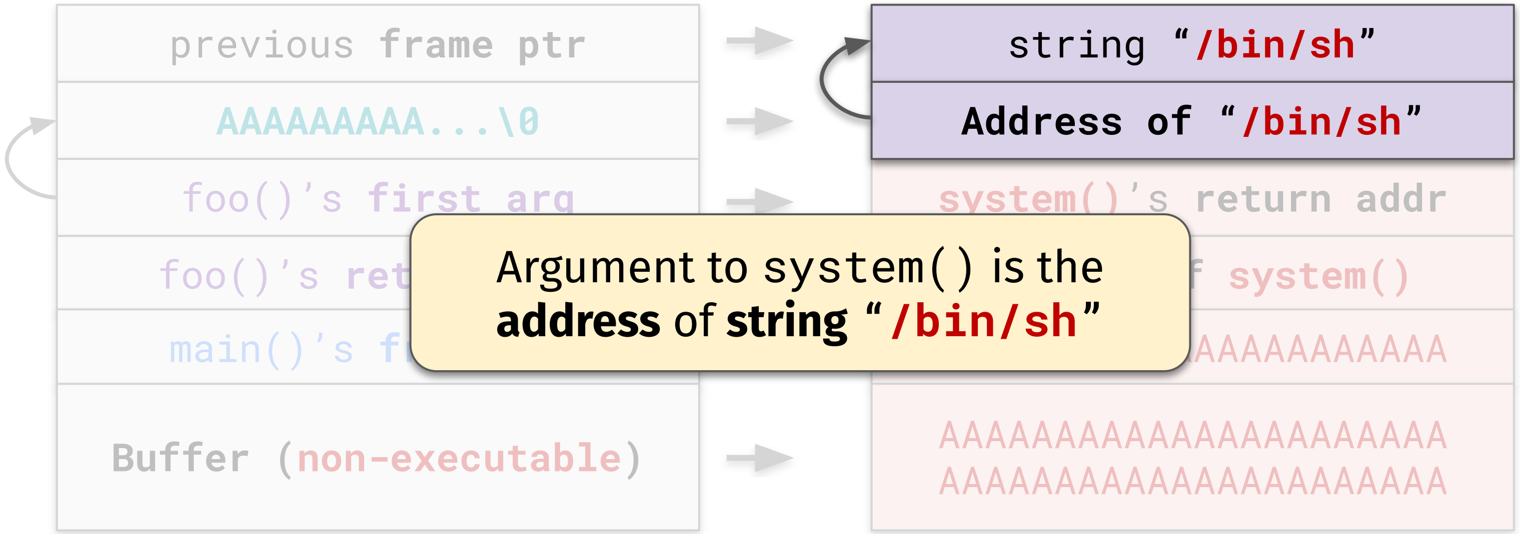
Defeating DEP via Code Reuse

- **Idea #2:** create a **“fake” call frame for `system()`** with our desired arg



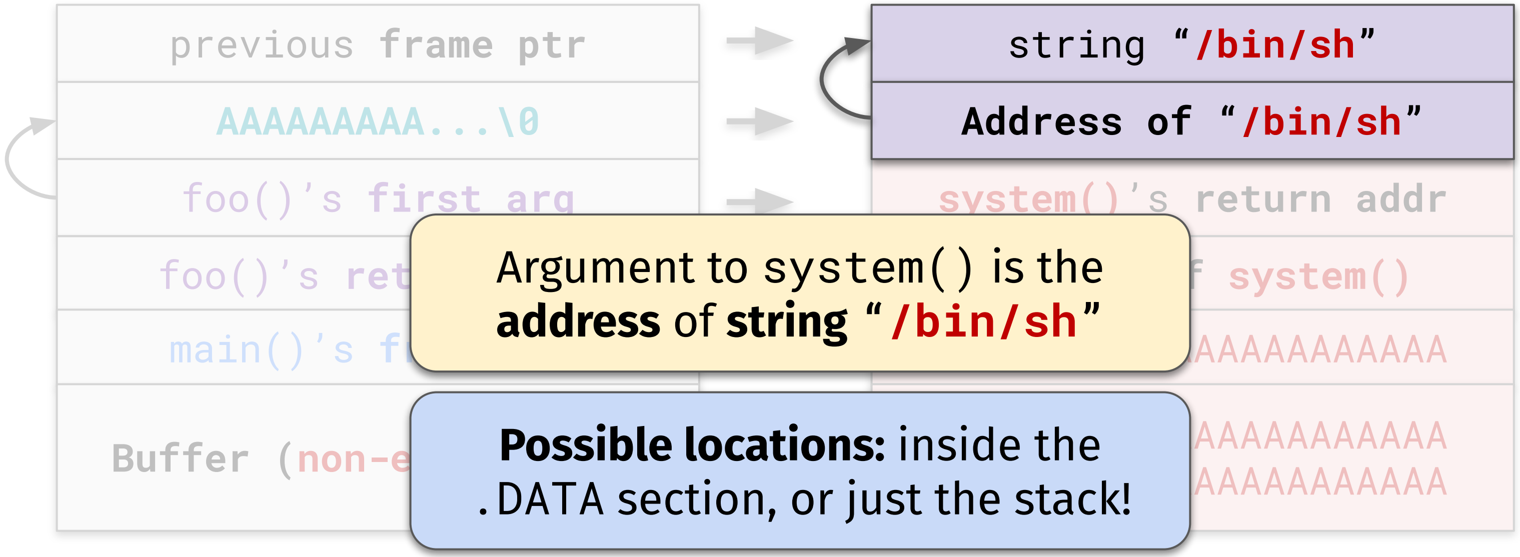
Defeating DEP via Code Reuse

- **Idea #2:** create a **"fake" call frame for `system()`** with our desired arg



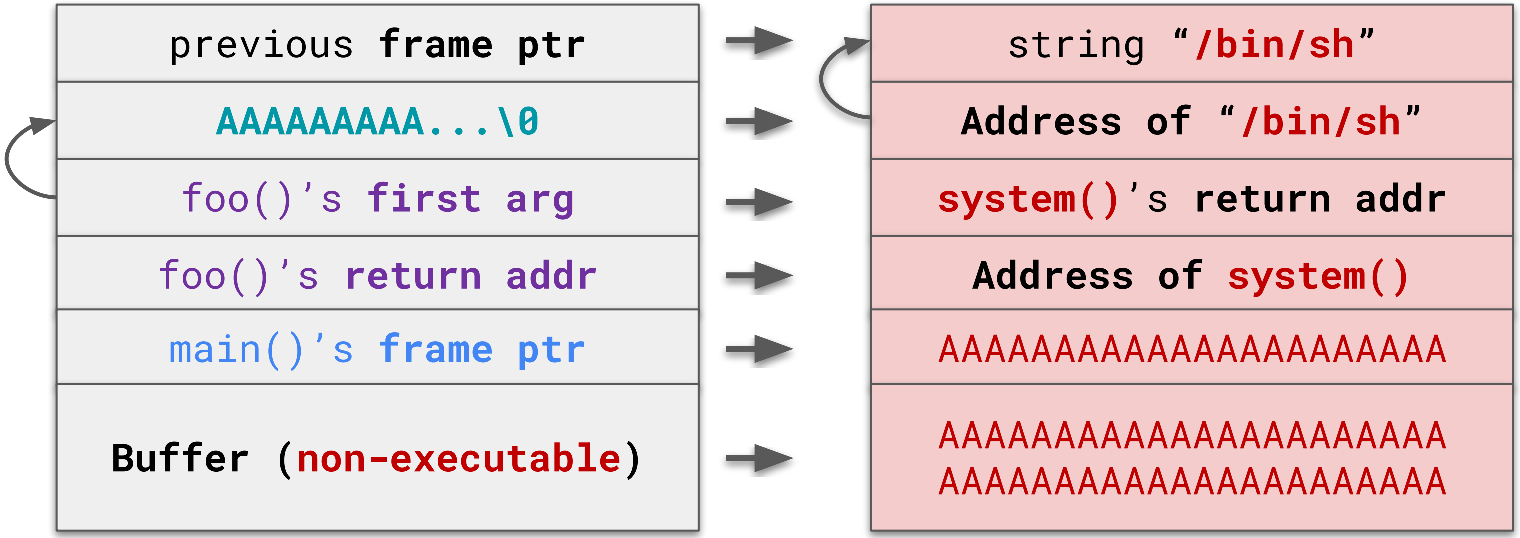
Defeating DEP via Code Reuse

- **Idea #2:** create a **"fake" call frame for `system()`** with our desired arg



Defeating DEP via Code Reuse

- **Idea #2:** create a **“fake” call frame for `system( )`** with our desired arg



Defeating DEP via Code Reuse

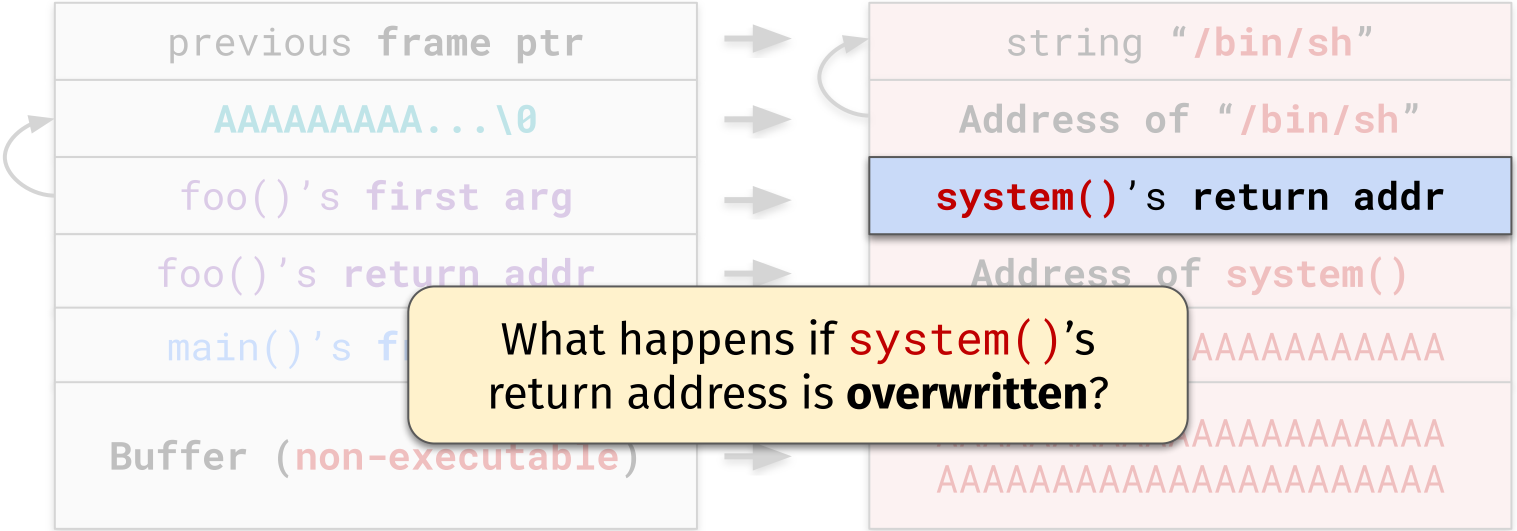
- **Idea #2:** c… …red arg

pre…
…
fo…
foo…
ma…
Buffe…

…h"
…ddr
…()
…AAA
…AAA …AAA

Defeating DEP via Code Reuse

- **Idea #2:** create a **"fake" call frame for `system()`** with our desired arg



What happens if `system()`'s return address is **overwritten**?

What happens to our exploit when system() returns?

It crashes!

0%

It executes normally...

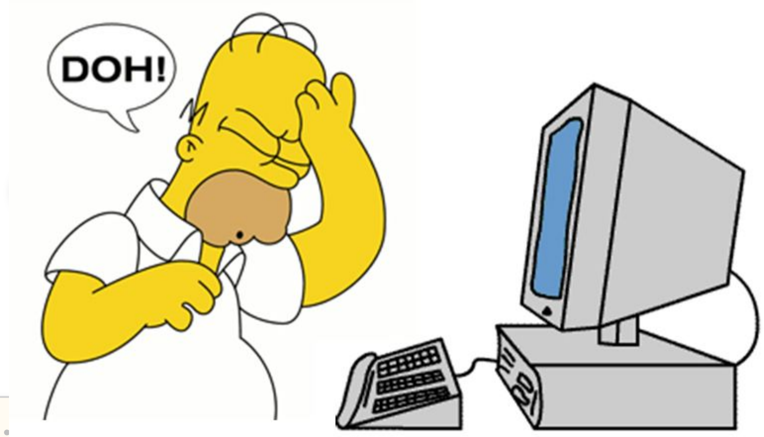
0%

None of the above

0%

Defeating DEP via Code Reuse

- **Idea #2:** create a **"fake" call frame for** **`system()`** with our desired arg



segmentation fault.
(Core dumped)

Defeating DEP via Code Reuse… stealthily!

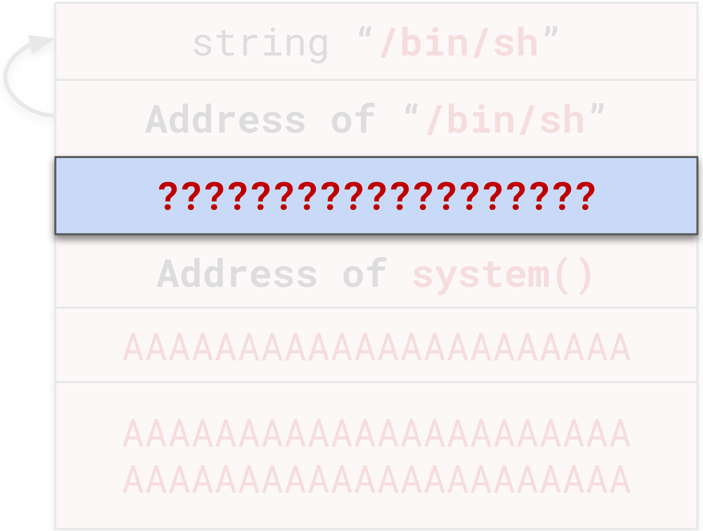
- **Idea #2:** create a **“fake” call frame for `system()`** with our desired arg

Description

The function **_exit**() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, *init*, and the process's parent is sent a **SIGCHLD** signal.

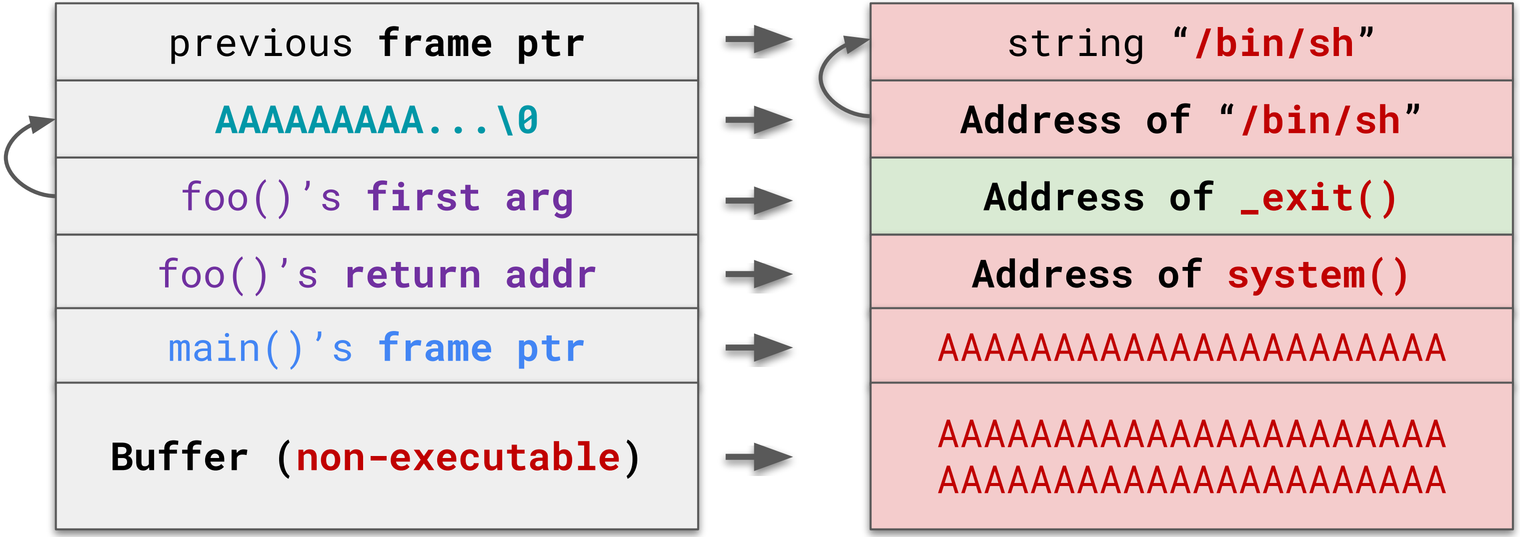
The value *status* is returned to the parent process as the process's exit status, and can be collected using one of the **wait**(2) family of calls.

The function **_Exit**() is equivalent to **_exit**().



Defeating DEP via Code Reuse… stealthily!

- **Idea #2:** create a **“fake” call frame for `system()`** with our desired arg



Defeating DEP via Code Reuse... stealthily!

- **Idea #2:** c … red arg

pre
… h"
fo …)
foo … ()
ma … AAA
Buffe … AAA AAA

Questions?

Other Attacks

Return Oriented Programming (ROP)

- Don’t have to jump only to function starts
  - Can jump in the middle of any code
  - x86 has variable instruction lengths
    - Most sequences of “bytes” can be an instruction

- **Idea:** Construct **Turing-complete set of “gadgets”** out of program’s code

- Use **Return-to-libc** like chaining to execute multiple gadgets in sequence!

- ROP is hard to master—we will not expect you to solve this
  - But you can for extra credit ;)

Other Exploitation Techniques

1997
Function ptr hijacking

1997
Ret-2-Libc attacks

1996
Stack overflows

1972
First known overflows

1998
Heap overflows

1998
StackGuard bypasses

1999
Format strings

2002
Integer overflows

2007
Heap grooming

2005
Ret oriented programming

2005
Hardware DEP bypasses

2002
ASLR bypasses

2007
Null pointer dereference

2007
Double frees

2009
Heap spraying

2010
JIT spraying

2021
Zero-click exploits

2016
Data oriented programming

2014
Call oriented programming

2011
Jmp oriented programming

What's next?

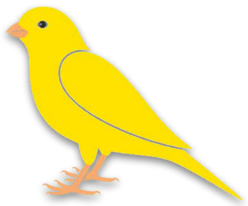
Attack Resources

- Aleph One's "Smashing the Stack for Fun and Profit"
  - http://insecure.org/stf/smashstack.html
- Paul Makowski's "Smashing the Stack in 2011"
  - http://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/
- Blexim's "Basic Integer Overflows"
  - http://www.phrack.org/issues.html?issue=60&id=10
- Return-to-libc demo:
  - http://www.securitytube.net/video/258

Other Defenses

Stack Canaries

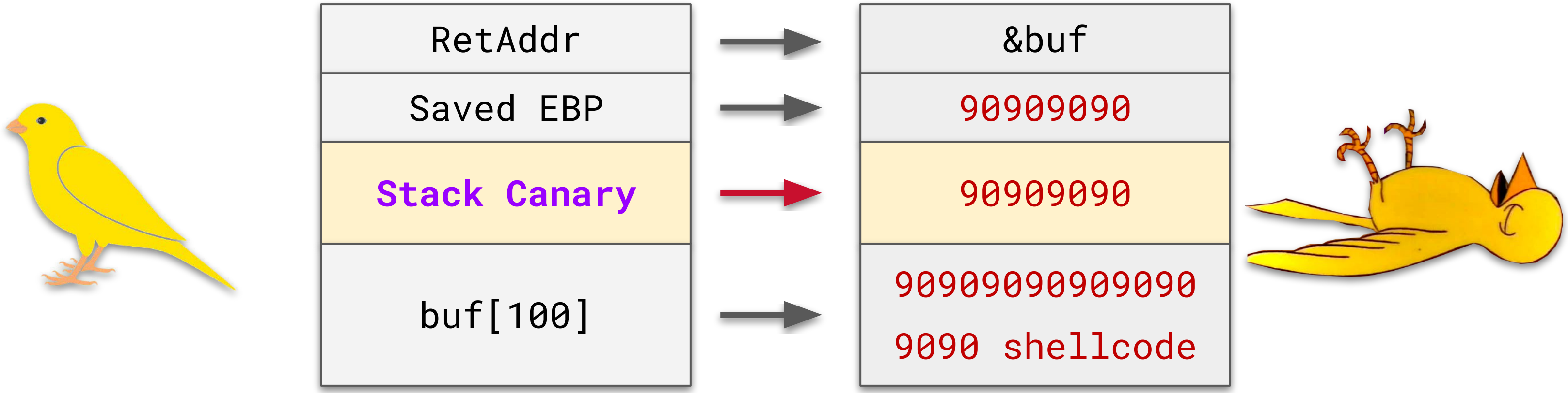
- **Basic idea:** place a **value** near the buffer, check at runtime if it's **overwritten**
  - Analogous to the real-world concept of "canary in a coalmine"



RetAddr
Saved EBP
Stack Canary
buf[100]

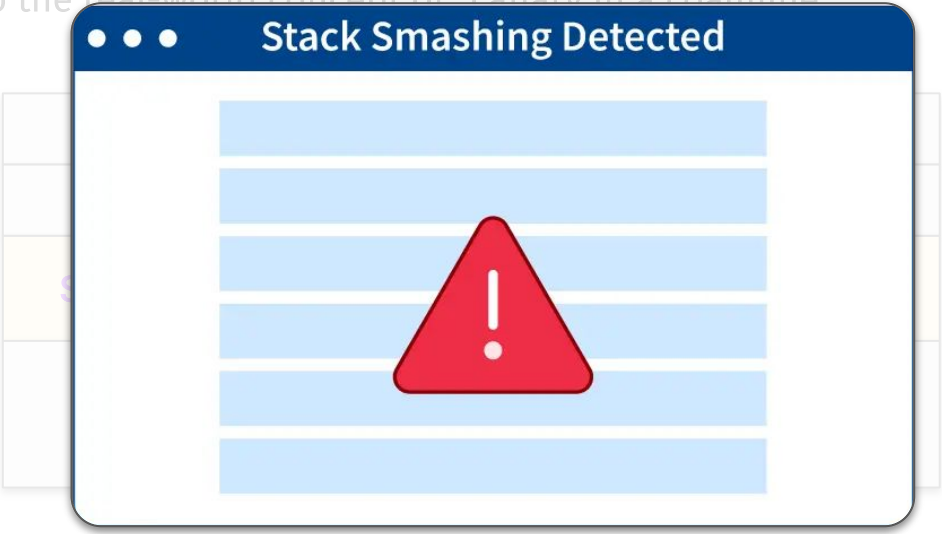
Stack Canaries

- **Basic idea:** place a **value** near the buffer, check at runtime if it’s **overwritten**
  - Analogous to the real-world concept of “canary in a coalmine”



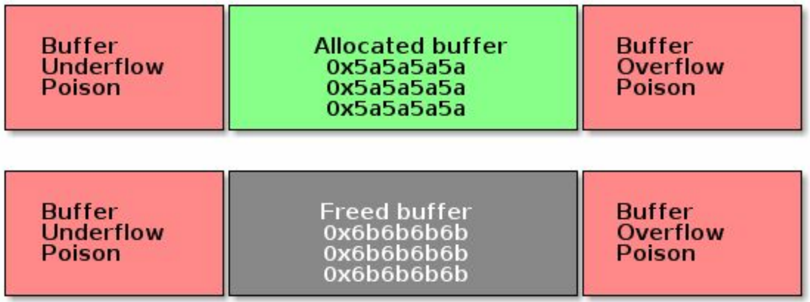
Stack Canaries

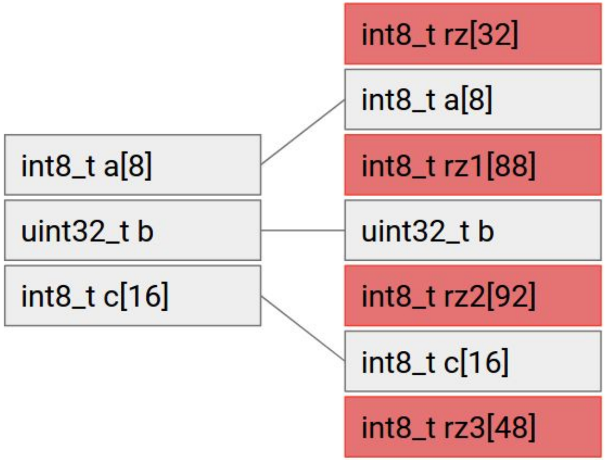
- **Basic idea:** place a **value** near the buffer, check at runtime if it's **overwritten**
  - Analogous to the real-world concept of "canary in a coalmine"



Application-level Changes

- **Memory error detectors (e.g., AddressSanitizer)**
  - **Key idea:** inject “red zones” before and after all memory objects
  - Force a crash when accessing a red zone
  - Catch all subtle (non-crashing) corruptions
  - Implement via instrumentation, custom `malloc()`
  - **Trade-off:** over **6x** execution overhead





Application-level Changes

- **Avoiding unsafe functions**

- Unsafe:
  - `strcpy` and friends (`str*`)
  - `sprintf`
  - `Gets`

- Use instead:
  - `strncpy` and friends (`strn*`)
  - `snprintf`
  - `fgets`

CWE-242: Use of Inherently Dangerous Function

Weakness ID: 242
Abstraction: Base
Structure: Simple

View customized information: Conceptual Operational Mapping-Friendly Complete

Description

The product calls a function that can never be guaranteed to work safely.

Extended Description

Certain functions behave in dangerous ways regardless of how they are used. Functions in this category were often implemented without taking security concerns into account. The gets() function is unsafe because it does not perform bounds checking on the size of its input. An attacker can easily send arbitrarily-sized input to gets() and overflow the destination buffer. Similarly, the >> operator is unsafe to use when reading into a statically-allocated character array because it does not perform bounds checking on the size of its input. An attacker can easily send arbitrarily-sized input to the >> operator and overflow the destination buffer.

Preventative Measures

- Refactoring:
  - Add bounds checking
  - “Sanitizer” user input
- Static bug detection tools:
  - C: Secure Programming Lint
  - C++: CPPCheck
- **Hire CS4440™ graduates**

Preventative Measures

- Refactoring:
  - Add bounds checking
  - “Sanitizer” user input
- Static bug detection tools:
  - C: Secure Programming Lint
  - C++: CPPCheck
- **Hire CS4440™ graduates**
- Deploy **automated testing** (next lecture’s topic)

Questions?

Next time on CS 4440…

Automated Bug Finding