Week 5: Lecture B
Attacking Applications

Thursday, September 21, 2023
Announcements

- **Project 1: Crypto**
  - **Deadline:** **tonight** by 11:59PM
Announcements

- **Project 2: AppSec** released
  - **Deadline:** Thursday, October 19th by 11:59PM

---

**Project 2: Application Security**

**Deadline:** Thursday, October 19 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

**Table of Contents:**
- Helpful Resources
- Introduction
- Objectives
- Start by reading this!
  - Setup Instructions
  - Important Guidelines
- Part 1: Beginner Exploits
  - Target 0: Variable Overwrite
  - Target 1: Execution Redirect
  - What to Submit
- Part 2: Intermediate Exploits
  - Target 2: Shellcode Redirect
  - Target 3: Indirect Overwrite
  - Target 4: Beyond Strings
  - What to Submit
- Part 3: Advanced Exploits
  - Target 5: Bypassing DEP
  - Target 6: Bypassing ASLR
  - What to Submit
- Part 4: Super L337 Pwnage
  - Extra Credit: Target 7
  - Extra Credit: Target 8
  - What to Submit
- Submission Instructions
CS 4440 Wiki: All Things CS 4440

This Wiki is here to help you with all things CS 4440: from setting up your VM to introducing the languages and tools that you’ll use. Check back here throughout the semester for future updates.

Have ideas for other pages? Let us know on Piazza!

## Tutorials and Cheat Sheets

<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM Setup &amp; Troubleshooting</td>
<td>Instructions for setting up your CS 4440 Virtual Machine (VM).</td>
</tr>
<tr>
<td>Terminal Cheat Sheet</td>
<td>Navigating the terminal, manipulating files, and other helpful tricks.</td>
</tr>
<tr>
<td>Python 3 Cheat Sheet</td>
<td>A gentle introduction to Python 3 programming.</td>
</tr>
<tr>
<td>x86 Assembly Cheat Sheet</td>
<td>Common x86 instructions and instruction procedures.</td>
</tr>
<tr>
<td>C Cheat Sheet</td>
<td>Information on C functions, and storing and reading data.</td>
</tr>
<tr>
<td>GDB Cheat Sheet</td>
<td>A quick reference for useful GNU Debugger (GDB) commands.</td>
</tr>
<tr>
<td>JavaScript Cheat Sheet</td>
<td>A gentle introduction to relevant JavaScript commands.</td>
</tr>
</tbody>
</table>
See Discord for meeting info!

www.utahsec.com
Last time on CS 4440...

Program Execution
Virtual Memory
The Stack
Stack Corruption
Software bugs lead to **unintended behavior**

```c
int main(void) {
    char buffer[40];
    gets(buffer);
    // Saves user input into the buffer
    // into the buffer
}
```

**CWE-242: Use of Inherently Dangerous Function**

**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.
**Problem:** attacker can’t load their own code on to the system

**Opportunity:** the attacker can interact with *existing programs*

**Challenge:** make the system do *what you want*... using only the existing programs on the system that you can interact with
Software Exploitation

- **Goal:** take over a system by exploiting an application on it

- **Exploit technique 1:** code injection
  - Insert your own code (as an input)
  - Redirect the program to execute it

- **Exploit technique 2:** code reuse
  - Leverage the program’s existing code
  - Execute it in a way it wasn’t intended to

- **Attack vector:** memory corruption
Virtual Memory

The "Break"

Higher Memory

0xFFFFFFFF

Kernel Virtual Memory

Stack Memory

unmapped

Shared Libraries

unmapped

Heap Memory

Uninitialized Data

Initialized Data

Lower Memory

0x08048000

Program Text

unmapped

0x00000000

0xC0000000

Program Text

0xFFFFFFFF

Kernel Virtual Memory

Stack Memory

unmapped

Shared Libraries

unmapped

Heap Memory

Uninitialized Data

Initialized Data
Virtual Memory

Higher Memory

0xFFFF_FFFF
0xC0000000

The “Break”

Kernel Virtual Memory

Stack Memory

unmapped

Shared Libraries

Uninitialized Data

Initialized Data

Program Text

unmapped

0x08048000
0x00000000

Lower Memory

Stack grows downwards

Heap grows upwards
Virtual Memory

Kernel Virtual Memory

Unmapped

Shared Libraries

Unmapped

Uninitialized Data

Initialized Data

Program Text

unmapped

The “Break”

Stack grows downwards
- Filled upwards

Heap grows upwards
- Filled downwards

Higher Memory

0xFFFFF000
0xC0000000

Lower Memory

0x08048000
0x00000000
Stack Operation

1. Push 0xA
2. Push 0x6C
3. **Push 0xFF**

```
<table>
<thead>
<tr>
<th></th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A</td>
<td></td>
</tr>
<tr>
<td>6C</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td></td>
</tr>
</tbody>
</table>
```

Stack grows → move SP down!
Push and Pop

1. Push 0x0A
2. Push 0x6C
3. Push 0xFF
4. Pop R1

Pop sends data at top of stack to a register
1. Push 0x0A
2. Push 0x6C
3. Push 0xFF
4. **Pop R1**

Stack clears → **move SP up!**
Stack Frames

- Assume `main()` calls `foo()`

**Call-er** (`main`) Stack Frame

- `main()`’s **local vars**
- `foo()`’s **arguments**
- `foo()`’s **return addr**
- `main()`’s **frame ptr**
- `foo()`’s **local vars**
  - .......

**Call-ee** (`foo`) Stack Frame
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

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

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

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

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

void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}

mov %ebp, %esp
pop %ebp
pop %eip

Execution will return to a garbage address!
“AAAA” = 0x41414141
Questions?
This time on CS 4440...

- Shellcode
- Constructing Exploits
- Pointer Dereferences
- Integer Overflows
What goals would an attacker have?

- **Controlling a local variable**
  - E.g., setting variable grade to an A+

- **Redirect execution to some function**
  - E.g., calling function `print_good_grade()`
What goals would an attacker have?

- **Controlling a local variable**
  - E.g., setting variable grade to an A+

- **Redirect execution to some function**
  - E.g., calling function print_good_grade()

- **Make the program execute evil code**
  - **Ideal goal:** gain root access to the system
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
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")
Executing Shellcode

- **Problem**: how can we construct our attack to *execute* our shellcode?

```
RetAddr              9090909090909090
Saved EBP           9090909090909090
other stuff         9090909090909090
buf[100]             9090909090909090
                   9090 shellcode
```
Executing Shellcode

**Problem:** how can we construct our attack to **execute** our shellcode?

**Solution:** overwrite **RetAddr** with the address of where our shellcode is!
- We put our shellcode in the **buffer**—so its **starting address** is the buffer’s location!

![Diagram showing the relationship between RetAddr, Saved EBP, other stuff, buf[100], &buf, and the shellcode address.](image-url)
**Problem:** how can we construct our attack to execute our shellcode?

**Solution:** overwrite `RetAddr` with the address of where our shellcode is!

We put our shellcode in the buffer—so its starting address is the buffer's location!
Questions?
Constructing Exploits
Project 2 Overview

- **We give you some binaries to exploit**
  - Limited to some rudimentary attacks
    - These don’t exist anymore in practice
    - See Targets 7–8 for more “realistic” ones

- **Various obstacles and defenses to beat**
  - Targets 0–2: None... **unbounded** overflow!
  - Target 3: **Bounded** overflow (strcpy())
  - Target 4: Requires a **two-step** exploit
  - Target 5: **DEP** (non-executable stack)
  - Target 6: **ASLR** (randomized stack location)
These challenges seem **daunting**
- We are covering C, x86, GDB, etc.

**Common questions that I’m seeing:**
- “I have absolutely zero experience with **C programming**!”
- “I’m trying to draw the stack but I don’t know **assembly**!”
- “How do I calculate the **exact number of padding** bytes?”
- “I don’t know **where to look** to find this thing in memory!”
- “My attack should be working, **but it SEGFAULTS**... why?!?!”
Project 2 Overview

- These challenges seem daunting
  - We are covering C, x86, GDB, etc.

- Common questions that I’m seeing:
  - “I have absolutely zero experience with C programming!”
  - “I’m trying to draw the stack but I don’t know assembly!”
  - “How do I calculate the exact number of padding bytes?”
  - “I don’t know where to look to find this thing in memory!”
  - “My attack should be working, but it SEGFAULTS... why?!”

No expertise necessary!
You’ll use just a few skills...
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!
D.E.N.N.I.S.

Dive into the source code
Dive into the Source Code

- **Objective:** understanding the program
- **Challenge:** understanding C programming

```c
int main(int argc, char *argv[]) {
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("%s,%s", name, grade);
}
```
<table>
<thead>
<tr>
<th>Experience with C?</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (that's totally okay!)</td>
<td>0%</td>
</tr>
<tr>
<td>Some</td>
<td>0%</td>
</tr>
<tr>
<td>Lots!</td>
<td>0%</td>
</tr>
</tbody>
</table>
Dive into the Source Code

- **Objective:** understanding the program

- **Challenge:** understanding C programming
  - Don’t sweat it—we don’t expect you to master C!

```c
int main(int argc, char *argv[]) {
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("%s,%s", name, grade);
}
```
Dive into the Source Code

- **Objective:** understanding the program

- **Challenge:** understanding **C programming**
  - Don’t sweat it—we **don’t** expect you to master C!

- Ideas from other **OOP languages** carry over
  - Functions
  - Local variables
  - Function arguments
  - Same building blocks as Java, Python, C++, etc.
  - **Finding the “best” order of teaching you these remains an unsolved problem in CS education!**

```c
int main(int argc, char *argv[])
{
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("%s,%s", name, grade);
}
```
Objective: understanding the program

Challenge: understanding C programming
  - Don’t sweat it—we don’t expect you to master C!

Need more info about a function?
  - Answer: locate and read its manpage
    - Short for “manual page”
  - E.g., “How is `strcpy` different from `strncpy`?”
    - [https://linux.die.net/man/3/strcpy](https://linux.die.net/man/3/strcpy)
    - Many other helpful resources on the web

**Strcpy(3) - Linux man page**

**Name**
`strcpy`, `strncpy` - copy a string

**Synopsis**
```c
#include <string.h>
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

**Description**
The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte (`'\0'`), to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)

The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied. *Warning:* If there is no null byte among the first `n` bytes of `src`, the string placed in `dest` will not be null-terminated.

If the length of `src` is less than `n`, `strncpy()` writes additional null bytes to `dest` to ensure that a total of `n` bytes are written.
Dive into the Source Code

- **Objective:** understanding the program

- **Challenge:** understanding C programming
  - Don’t sweat it—we don’t expect you to master C!

- See the C Cheat Sheet on the CS 4440 Wiki

---

C seems daunting, but you don’t need to master it—just understand the basics, and keep a link or two bookmarked for the rest!
Objective: understanding the program

Fundamental questions to consider:

1. What is my target function?
2. What variables does it have?
3. How is data written to stack?
4. How far can data be written?
5. What is the goal of my attack?
Objective: understanding the program

Fundamental questions to consider:

1. What is my target function?
2. What variables does it have?
3. How is data written to stack?
4. How far can data be written?
5. What is the goal of my attack?

```c
int main(int argc, char *argv[])
{
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("%s,%s", name, grade);
}
```
Example: Target 0

- **Objective:** understanding the program

- **Fundamental questions to consider:**
  1. What is my target function?
      - main()
  2. What variables does it have?
      - char grade[5], char name[10]
  3. How is data written to stack?
      - gets(name)
  4. How far can data be written?
      - As far as we want!
  5. What is the goal of my attack?
      - To overwrite char grade[5]!

```c
int main(int argc, char *argv[])
{
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("%s,%s", name, grade);
}
```
## Target Reconnaissance

<table>
<thead>
<tr>
<th>Target</th>
<th>What is our attack’s goal?</th>
<th>How to write up the stack?</th>
<th>How far can we write?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Overwrite <strong>Variable</strong></td>
<td><code>gets()</code></td>
<td><strong>Unbounded</strong></td>
</tr>
<tr>
<td>1</td>
<td>Redirect to <strong>Function</strong></td>
<td><code>strcpy()</code></td>
<td><strong>Unbounded</strong></td>
</tr>
<tr>
<td>2</td>
<td>Redirect to <strong>Shellcode</strong></td>
<td><code>strcpy()</code></td>
<td><strong>Unbounded</strong></td>
</tr>
</tbody>
</table>
### Target Reconnaissance

<table>
<thead>
<tr>
<th>Target</th>
<th>What is our attack’s goal?</th>
<th>How to write up the stack?</th>
<th>How far can we write?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Overwrite <strong>Variable</strong></td>
<td><code>gets()</code></td>
<td>Unbounded</td>
</tr>
<tr>
<td>1</td>
<td>Redirect to <strong>Function</strong></td>
<td><code>strcpy()</code></td>
<td>Unbounded</td>
</tr>
<tr>
<td>2</td>
<td>Redirect to <strong>Shellcode</strong></td>
<td><code>strcpy()</code></td>
<td>Unbounded</td>
</tr>
<tr>
<td>3</td>
<td>Redirect to <strong>Shellcode</strong></td>
<td><code>strncpy()</code></td>
<td>Bounded</td>
</tr>
<tr>
<td>4</td>
<td>Redirect to <strong>Shellcode</strong></td>
<td><code>fread()</code></td>
<td>Bounded</td>
</tr>
</tbody>
</table>
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!
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 $8 + \text{sizeof(buf)}$ 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 $8 + \text{sizeof(buf)}$ bytes
  - **Target 4**: we can only write $\text{count}$ bytes (via `fread()`)

For **bounded** writes, we have to get creative and **find a way to overwrite** what we want!
Questions?
Overcoming Bounded Writes: Pointer Dereferencing
Overcoming Bounded Writes

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

- **Target 3: ???**

```c
int *p;
int a;
*p = a;
```
Overcoming Bounded Writes

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

- **Target 3: a pointer dereference**

  *p* = 5, whatever *p* points to will be updated to 5
Overcoming Bounded Writes

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

- **Target 3: a pointer dereference**

  ```c
  int *p;
  int a;
  *p = a;
  ```

  - If we set \( *p = 5 \), **whatever p points to** will be updated to 5
  - If we take control over both \( a \) and \( p \), we can **change arbitrary objects** in memory
Recap: Process Virtual Memory

- Kernel Virtual Memory
- Stack Memory
- unmapped
- Shared Libraries
- unmapped
- Heap Memory
- Uninitialized Data
- Initialized Data
- Program Text
- unmapped
Recap: Process Virtual Memory

<table>
<thead>
<tr>
<th>Higher Memory</th>
<th>Kernel Virtual Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stack Memory</td>
</tr>
<tr>
<td></td>
<td>unmapped</td>
</tr>
<tr>
<td></td>
<td>Shared Libraries</td>
</tr>
<tr>
<td></td>
<td>unmapped</td>
</tr>
<tr>
<td></td>
<td>Heap Memory</td>
</tr>
<tr>
<td></td>
<td>Uninitialized Data</td>
</tr>
<tr>
<td></td>
<td>Initialized Data</td>
</tr>
<tr>
<td></td>
<td>Program Text</td>
</tr>
<tr>
<td></td>
<td>unmapped</td>
</tr>
</tbody>
</table>

Local variables, and a record of active functions

(and a whole bunch of other stuff...)

Program instructions
Recap: Process Virtual Memory

Key idea: it’s all “things” pointed to by addresses

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher Memory</td>
<td></td>
</tr>
<tr>
<td>Kernel Virtual Memory</td>
<td></td>
</tr>
<tr>
<td>Stack Memory</td>
<td>unmapped</td>
</tr>
<tr>
<td>Shared Libraries</td>
<td>unmapped</td>
</tr>
<tr>
<td>Heap Memory</td>
<td></td>
</tr>
<tr>
<td>Uninitialized Data</td>
<td></td>
</tr>
<tr>
<td>Initialized Data</td>
<td></td>
</tr>
<tr>
<td>Program Text</td>
<td>unmapped</td>
</tr>
<tr>
<td>Lower Memory</td>
<td></td>
</tr>
</tbody>
</table>
**Recap: Process Virtual Memory**

Key idea: it’s all “things” pointed to by addresses

<table>
<thead>
<tr>
<th>Memory Region</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Virtual Memory</td>
<td>Unmapped</td>
</tr>
<tr>
<td>Stack Memory</td>
<td>Unmapped</td>
</tr>
<tr>
<td>Shared Libraries</td>
<td>Unmapped</td>
</tr>
<tr>
<td>Heap Memory</td>
<td></td>
</tr>
<tr>
<td>Uninitialized Data</td>
<td></td>
</tr>
<tr>
<td>Initialized Data</td>
<td></td>
</tr>
<tr>
<td>Program Text</td>
<td>Unmapped</td>
</tr>
</tbody>
</table>

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

```
$ disas vulnerable:
0x0804a17b <+0>:  endbr32
0x0804a17f <+4>:  push   %ebp
0x0804a180 <+5>:  mov    %esp,%ebp
0x0804a182 <+7>:  push   %ebx
```
# Recap: Process Virtual Memory

## Key idea:
It’s all “things” pointed to by addresses.

## Example: payload NOPs in Stack Memory:

```bash
$ x/32xw 0xffffd8cc
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffd8cc</td>
<td>0x90909090</td>
<td>0x90909090</td>
</tr>
<tr>
<td>0xffffd8d4</td>
<td>0x90909090</td>
<td>0x90909090</td>
</tr>
<tr>
<td>0xffffd8dc</td>
<td>0x90909090</td>
<td>0x90909090</td>
</tr>
<tr>
<td>0xffffd8e4</td>
<td>0x90909090</td>
<td>0x90909090</td>
</tr>
</tbody>
</table>
Leveraging Pointer Dereferences

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

**Target 3:** the return address is stored on the stack. In other words, an **address** in stack memory **points to** a slot **containing** it.

```c
int *p;
int a;
*p = a;
```

- If we set \( *p = 5 \), whatever \( p \) points to will be updated to 5.
- If we take control over both \( a \) and \( p \), we can change arbitrary objects in memory.
Leveraging Pointer Dereferences

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

- Target 3: a pointer dereference
  - If we set `*p = 5`, whatever `p` points to will be updated to 5.
  - If we take control over both `a` and `p`, we can change arbitrary objects in memory.

```c
int *p;
int a;
*p = a;
```

- Target 3: the return address is stored on the stack. In other words, an address in stack memory points to a slot containing it.

- We can exploit the dereference to overwrite the value a stack memory address points to!
void foo(char *str) {
    int *p;
    int a;
    *p = a;
}

Indirect Memory Overwrite
void foo(char *str) {
    int *p;
    int a;
    *p = a;
}

foo()'s retAddr

caller’s EBP

int a

BP

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

foo()'s retAddr

caller’s EBP
Address 0x000000

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

foo()'s retAddr

caller's EBP

Address 0x000000

int a

Stack Addresses

EBP+4
EBP+0
EBP-4
EBP-8
void foo(char *str) {
    int *p;
    int a;
    *p = a;
}

Indirect Memory Overwrite

Stack Addresses

- EBP+4
- EBP+0
- EBP-4
- EBP-8

Contents of 0x000000 updated to a
void foo(char *str) {
    int *p;
    int a;
    *p = a;
}
Indirect Memory Overwrite

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

Stack Addresses:
- `foo()`'s retAddr: EBP+4
- caller's EBP: EBP+0
- Address EBP+4: EBP+4
- Shellcode Address EBP-4: EBP-4
- EBP-8: EBP-8

The diagram shows the stack addresses and the flow of data in the `foo` function.
Indirect Memory Overwrite

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

Contents of EBP+4 updated to the shellcode address!
### Target Reconnaissance

<table>
<thead>
<tr>
<th>Target</th>
<th>What is our attack’s goal?</th>
<th>How to write up the stack?</th>
<th>How far can we write?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Overwrite Variable</td>
<td>gets()</td>
<td>Unbounded</td>
</tr>
<tr>
<td>1</td>
<td>Redirect to Function</td>
<td>strcpy()</td>
<td>Unbounded</td>
</tr>
<tr>
<td>2</td>
<td>Redirect to Shellcode</td>
<td>strcpy()</td>
<td>Unbounded</td>
</tr>
<tr>
<td>3</td>
<td>Redirect to Shellcode</td>
<td>Dereference Return Addr’s stack location</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Redirect to Shellcode</td>
<td>fread()</td>
<td></td>
</tr>
</tbody>
</table>

Now update your high-level plan!
Other Overwritable Objects

- **Not just return addresses!**
  - Function pointers
  - Arbitrary data
  - C++ exceptions
  - C++ objects
  - Heap memory freelist
  - **Any code pointer!**
Questions?
Overcoming Bounded Writes: Integer Overflows
Overcoming Bounded Writes

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

- Target 4: ???

```c
alloca( count * 4 );  // allocate our buffer
fread( &buf[i], 4, count, f );  // fill buffer
```
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 read into it

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

Range of `count`:
- `[0, \( \frac{1}{4}(\text{MAX\_UINT}) \)]`
- `[0, \text{MAX\_UINT}]`
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 read into it**

- 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!

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

Range of `count`:
- `[0, \text{\texttt{MAX_UINT/4}})`
- `[0, \text{\texttt{MAX_UINT}})`
Integer Overflows

- **Integer overflows** behave differently from stack buffer overflows

### 32-bit Integer Range:

Unsigned: [0, \(2^{32} - 1\)]

Signed: \([-2^{31}, (2^{31} - 1)]\)
**Integer Overflows**

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

### 32-bit Integer Range:

**Unsigned:**
- \([0, 4294967295]\)

**Signed:**
- \([-2^{31}, 2^{31} - 1]\)
- \([-2147483648, 2147483647]\)
- **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., $0xFFFFFFFF + 2 = 0x00000002$
Example Integer Overflow

What is unsafe about this code?

```c
void foo(char *array, int len)
{
    int buf[100];

    if(len >= 100) {
        return;
    }

    memcpy(buf, array, len);
}
```
What is **unsafe** about this code?

```c
void foo(char *array, int len) {
    int buf[100];
    if(len >= 100) {
        return;
    }
    memcpy(buf, array, len);
}
```

```c
void *memcpy (void *dest, const void *src, size_t n);
```
What is unsafe about this code?

```c
void foo(char *array, int len)
{
    int buf[100];
    if(len >= 100) {
        return;
    }
    memcpy(buf, array, len);
}
```

`void *memcpy (void *dest, const void *src, size_t n);`

`size_t n` must be a signed int.
What is unsafe about this code?

```c
void foo(char *array, int len) {
    int buf[100];
    if(len >= 100) {
        return;
    }
    memcpy(buf, array, len);
}
```

- `size_t n` must be a `signed int`
- `memcpy` interprets a `negative` `len` as a huge unsigned value!
Example Integer Overflow

What is unsafe about this code?

```c
void foo(char *array, int len) {
    int buf[100];
    if(len >= 100) {
        return;
    }
    memcpy(buf, array, len);
}
```

- `void *memcpy (void *dest, const void *src, size_t n);`
- `size_t n` must be a `signed int`
- `memcpy` interprets a `negative len` as a huge unsigned value!
- **OVERFLOW**—Copy `way more than 100 bytes` into `dst` buffer!
What is unsafe about this code?

```c
void foo(char *array, int len)
{
    int buf[100];
    if (len >= 100) {
        return;
    }
    memcpy(buf, array, len);
}
```

`size_t n` must be a signed int.

`void *memcpy(void *dest, const void *src, size_t n);`

`memcpy` interprets a negative `len` as a huge unsigned value!

OVERFLOW—Copy way more than 100 bytes into dst buffer!
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

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

- 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!
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

- 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!

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

**Target 4:** a very large `count` will trigger an integer overflow in the buffer’s allocation, wrapping `MAX_UINT` to a very small size.
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
- 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!

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

Range of `count`:
- `count` ∈ `[0, MAX_UINT)`
- `<MAX_UINT`

Target 4: a very large `count` will trigger an integer overflow in the buffer’s allocation, wrapping `MAX_UINT` to a very small size.

Since we later write `count` elements into the buffer, this will trigger a buffer overflow... allowing overwriting of objects up the stack!
## Target Reconnaissance

<table>
<thead>
<tr>
<th>Target</th>
<th>What is our attack’s goal?</th>
<th>How to write up the stack?</th>
<th>How far can we write?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Overwrite Variable</td>
<td><code>gets()</code></td>
<td>Unbounded</td>
</tr>
<tr>
<td>1</td>
<td>Redirect to Function</td>
<td><code>strcpy()</code></td>
<td>Unbounded</td>
</tr>
<tr>
<td>2</td>
<td>Redirect to Shellcode</td>
<td><code>strcpy()</code></td>
<td>Unbounded</td>
</tr>
<tr>
<td>3</td>
<td>Redirect to Shellcode</td>
<td><code>strncpy()</code></td>
<td>Bounded</td>
</tr>
<tr>
<td>4</td>
<td>Redirect to Shellcode</td>
<td><code>fread()</code></td>
<td></td>
</tr>
</tbody>
</table>

**Integer Overflow on buf’s allocation size**

Now update your high-level plan!
Questions?
D.E.N.N.I.S.

Estimate the stack frame
Estimating the Stack

- **Objective: understand the memory layout**
  - What is needed for our attack to be successful?

- **Fundamental questions to consider:**
  1. What stack objects do we **control**?
  2. What stack objects can we **reach**?
  3. What’s our desired **final stack state**?

```c
void vulnerable(char *arg)
{
    char buf[100];
    strcpy(buf, arg);
}
```
Objective: understand the memory layout
- What is needed for our attack to be successful?

Fundamental questions to consider:
1. What stack objects do we control?
   - char buf[100]
2. What stack objects can we reach?
   - Everything upwards of buf!
3. What’s our desired final stack state?
   - Inject our shellcode within our vulnerable buffer buf
   - Overwrite vulnerable()’s return address with buf’s address!
Many of you will try to draw the stack based on the assembly...

Dump of assembler code for function vulnerable:

```
0x0804a17b <+0>:     endbr32
0x0804a17f <+4>:     push   %ebp
0x0804a180 <+5>:     mov    %esp,%ebp
0x0804a182 <+7>:     push   %ebx
0x0804a183 <+8>:     sub    $0x74,%esp
0x0804a186 <+11>:    call   0x804a208 <__x86.get_pc_thunk.ax>
0x0804a190 <+21>:    sub    $0x9fe75,%eax
0x0804a193 <+24>:    pushl  0x8(%ebp)
0x0804a196 <+27>:    lea    -0x6c(%ebp),%edx
...
Many of you will try to draw the stack based on the assembly...

Ditch the **assembly**... draw your stack based on the **source code**!
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!

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

Your **high-level stack diagram** should consist of the **Return Address**, **Saved EBP**, and **Locals**.
Drawing 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 be executed after the 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]`

Your **high-level stack diagram** should consist of the **Return Address**, **Saved EBP**, and **Locals**.

**No assembly required**—just look at the **source**!
Drawing 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]`

Your **high-level stack diagram** should consist of the **Return Address**, **Saved EBP**, and **Locals**.

No assembly required—just look at the **source**!

You need to get comfortable with this—highly recommended to revisit **All About Applications**
D.E.N.N.I.S.

NOP-out everything inside the frame!
Then, NOP-out just the return address!
- **Question:** how to calculate the **exact amount** of overflow to reach the return address?
  - Read the assembly code line by line
  - Revisit and tweak your stack diagram
  - If it doesn’t work, go back and look at more assembly

![Stack Diagram]

- RetAddr
- Saved EBP
- other stuff ???
- buf
Building your Attack

- **Question:** how to calculate the **exact amount** of overflow to reach the return address?
  - Read the assembly code line by line
  - Revisit and tweak your stack diagram
  - If it doesn’t work, go back and look at more assembly

- **Don’t do this**—you will go insane reading x86
Question: how to calculate the exact amount of overflow to reach the return address?
- Read the assembly code line by line
- Revisit and tweak your stack diagram
- If it doesn’t work, go back and look at more assembly

Don’t do this—you will go insane reading x86

Ditch the assembly... guesstimate your padding with a few heuristics!
Padding Heuristics

- **How large** is our vulnerable buffer?
  - E.g., char buf[100]
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

RetAddr

buf [100]

~100 bytes
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**
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

```
RetAddr  4 bytes
Saved EBP
other stuff ???  TBD bytes
buf [100]  ~100 bytes
```
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
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**
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

Use them and iteratively refine your payload!

SEGFAULTS are your friend—they indicate you’re on the right track (overwriting things)!

Tweak it to figure out the exact payload size
D.E.N.N.I.S.

Inspect the program’s memory
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**
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!**
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., 0xffff6d000 to 0xffffffff

```bash
$ info proc mapping // list all memory segments
StartAddr   EndAddr       Size     Offset objname
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]
0xffff6d000 0xffffe000  0x91000        0x0 [stack]
0xffff6d000 0xfffffe000  0x91000        0x0 [stack]
```

---

**School of Computing**

**University of Utah**

Stefan Nagy
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

```bash
$ 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 0xfff6d000,0xffffff000,"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

```bash
$ 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 ...
```
Other GDB Resources

- Other GDB resources:
  - CS 4440 GDB Cheat Sheet
  - Beej’s GDB Tutorial
  - Tudor’s GDB Tutorial

- Many others on the web!
Experience with GDB?

- None (that's totally okay!) - 0%
- Some - 0%
- Lots! - 0%
- Not with GDB, but other debuggers - 0%
Other GDB Resources

- CS 4440
- Beej's GDB
- Tudor's GDB Tutorial

We do NOT expect you to “master” GDB...
Other GDB Resources

We do NOT expect you to “master” GDB...

However, you should keep a link or two handy for quick referencing. See the CS 4440 Wiki!
Other GDB Resources

- CS 4440 RefCard
- Beej’s GDB Tutorial
- Tudor’s GDB Tutorial

We do **NOT** expect you to “master” GDB...

However, you should **keep a link or two** handy for quick referencing. **See the CS 4440 Wiki!**

You will definitely be faced with **GDB-style debugging scenarios** in **your future careers!**
D.E.N.N.I.S.

Setup and stabilize your attack!
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!
Troubleshooting

- E.g., “My attack **segfaults** and I don’t know why!”

- **Check your padding!**
  - Are you correctly overwriting the return address?

- **Check your payload order!**
  - If **shellcode** first, you must jump to buffer’s **exact start**!
  - If **NOPs** first, you can jump **anywhere** in the NOP slide!

- **Check your destination!**
  - Perform memory inspection to look for **known, friendly** payloads
  - Be sure to set breakpoints on a location **after the buffer is filled!**
Troubleshooting

- E.g., “My attack segfaults and I don’t know why!”

- Most troubleshooting requires just a little trial and error!

- Look for signs of progress (e.g., overwriting stack objects), and test whether your payload tweaks changes things!

- Perform memory inspection to look for known, friendly payloads
- Be sure to set breakpoints on a location after the buffer is filled!
Troubleshooting

- E.g., “My attack **segfaults** and I don’t know why!”

- Check your padding!

- Are you correctly overwriting the return address?

- Check your payload order!

- If **shellcode** first, you must jump to buffer’s **exact start**!

- If **NOPs** first, you can jump anywhere in the NOP slide!

- Check your destination!

- Perform memory inspection to look for known, friendly payloads

- Be sure to set breakpoints on a location after the buffer is filled!

Most troubleshooting requires just a little trial and error!

Look for signs of progress (e.g., overwriting stack objects), and **test** whether your payload tweaks changes things!
Questions?
Next time on CS 4440...

Defending Applications
And beating those defenses!