# Computer Systems Security

CSE 466
Fall 2018
Yan Shoshitaishvili

http://pwn.college
http://groups.google.com/group/cse-466
https://goo.gl/y9p351

## Homework 3 Retrospective

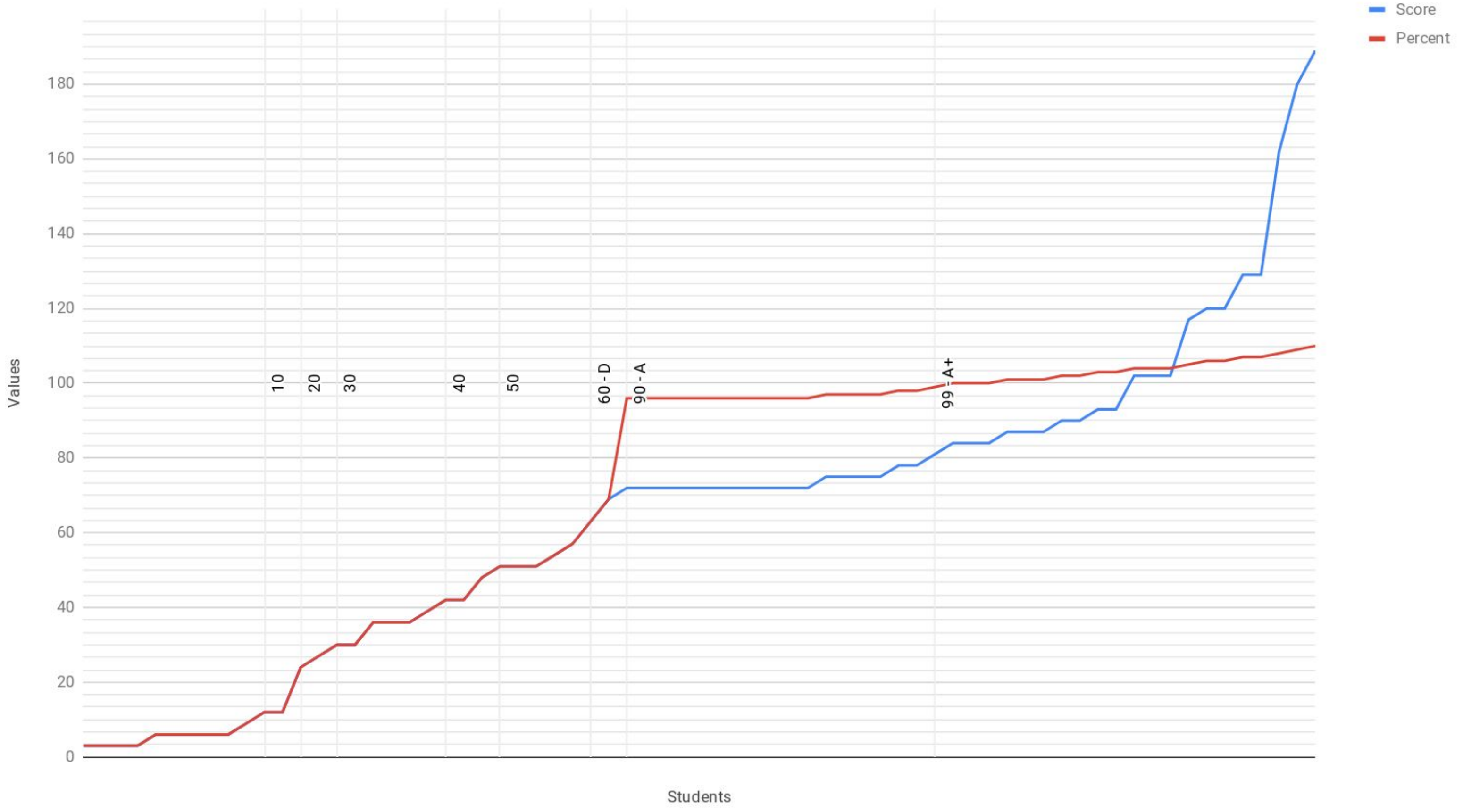
## Statistics

85 graded students (+1 auditing)

22 A+ (>= 99%)
17A (>= 90%)
0 B (>= 80%)
0 C (>= 70%)
2 D (>= 60%)
28 E (< 60%)
16 did not attempt

Because of smaller spread (fewer targets leading to many ties), anyone with over 70 flags got an A.

## Hw1 Scores and Percents



## Takeaways

The curve again represented a mega bonus.

- Great for those that got over the line.
- Not so great for those that *almost* made it over.
- Might lower the bar from 70 for future assignments.

Almost half the class is failing... Ideas:

- We will pull down the letter grade cutoffs (except for A+) by 3 percentage points (eliminate half of the negative impact of this hw).
- That only does so much.
- Statistics impacted by people waiting to withdraw.

## Top 3

Master hackers:

1. hunter2 - 189 points
2. smpwnd - 180 points
3. bazinga - 162 points

How did they do it?

## Determining Interaction Method

There were six interaction methods:

`fd`: read() straight from a file descriptor (stdin for adobe chals)
`arg`: get input from a commandline argument
`env`: get input from the environment variable
`mmap`: open and memory-map a file, read from that file
`file`: open a file and read from it (sometimes, this was a decoy!)
`fifo`: create and read from a fifo

`ltrace` distinguishes between all of these. Two need an extra step:

- For `env` input, you can get the environment variable name using `objdump`.
- For `arg` input, you can get the arg number from the error message.

## Understanding these Challenges

Each challenge has roughly the same structure:

- take input
- mangle input
- compare against solution
- exec /get_flag

Every step can be gleamed from ltrace *except for manglers*.

## Manglers

The following manglers were involved:

- `reverse`: reverses the input
- `shuffle`: shuffles the input
- `sort`: sorts the input
- `xor_ff`: xors the input by 0xff (bitflip), byte by byte
- `xor_42`: xors the input by 0xff, byte by byte
- `xor_1337`: xors the input by 0x1337, two bytes at a time (little endian)

Of these, three are easy, one is medium, and two are hard.

## Manglers Difficulty

Easy: reverse, shuffle, xor_42

- Effects bytes individually.
- Easily identifiable in ltrace.

Medium: xor_1337

- Effects bytes two at a time.
- Still easily identifiable in ltrace.

Hard:

- xor_ff: makes bytes non-printable.
- sort: rearranges the input in a way that depends on the input values

## Solving the Manglers

The easy and medium can be done in gdb (or by hand, if you are so inclined)!

DEMO: gdb scripting

## Solving the (Hard) Manglers

Can also be done in gdb, when they are on their own.

In later levels, with multiple manglers stacked on each other, they can cause problems.

Solution: do them one by one, backwards!

## Mastery Strategies

There are a few possibilities for automating at least parts of the solution:

- Sophisticated gdb scripting.
- Automated binary analysis with angr.
  - Best results when you *understand the code you are analyzing*.

Quick poll: who used complex automation?
What's the take-away from that?

## Used Strategies

Harder to get insight into strategies, as solving was "offline".

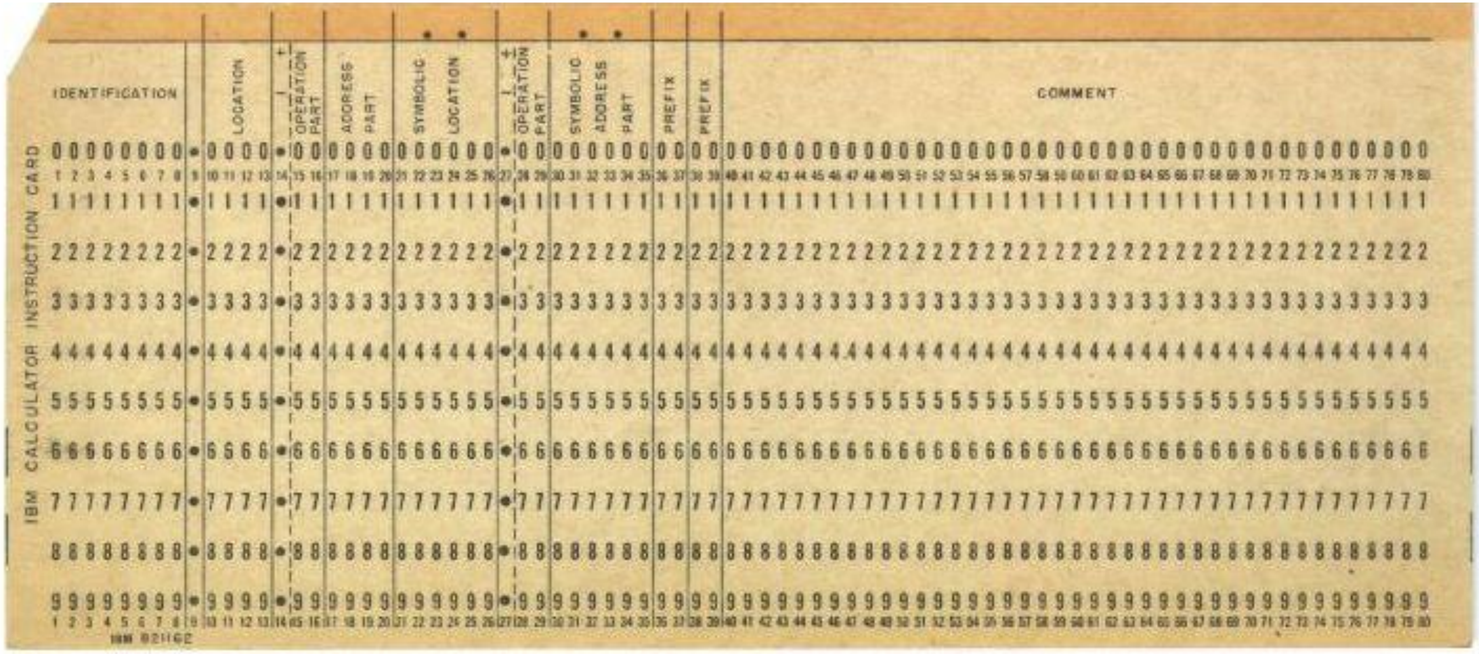
Let's discuss!

# CSE 466 Week 4

Pwning for Fun and Profit

# In the beginning...

Computers were originally programmed through direct input of machine code.



## In the beginning...

In 1952, Grace Hopper proposed one the first compilers.

Problem:

early compilers created inefficient code

Problem 2:

early computers were very slow

## Since then...

Early compiled languages were designed to allow the programmer as much control as possible. Things were slow to change.

A (simplified) timeline:

1970s: C is developed, maps almost directly to assembly (security implications).
1980s: Focus on features, C++ developed, compiled languages still dangerous.
1990s: Birth of modern VM-based languages. Mainstream compiled languages still dangerous.
2000s: Rise of JIT to improve VM-based/interpreted languages. Compiled languages still dangerous.
2010s: Finally exploring mainstream memory-safe compiled languages (i.e., Rust).

## The result...

An astonishing amount of software has been developed in languages with no memory safety!

C is *still* the second-most popular programming language.
C++ is 4th.
Objective-C is 10th.

C is the fastest growing language in terms of popularity!

Why?

https://www.tiobe.com/tiobe-index

## What's the problem with C?

In 1968, we start seeing concerns about memory corruption.

`Robert Graham.`

`"Protection in an information processing utility."`

`Communications of the ACM, 1968.`

In one of the first papers proposing memory isolation between processes, Graham, et al. muse (paraphrased):

"What if a program allows someone to overwrite memory they're not supposed to?"

## Disaster strikes!

In 1988, Robert Teppan Morris launches the first documented "buffer overflow" attack.

Accidentally **brought down the entire internet**.

The **whole internet** was taken apart and rebooted to fix it. Down for **days**.

Morris was the first person convicted under the 1986 Computer Fraud and Abuse Act.

Now a CS professor at MIT.

## This week...

We will take a first glance several types of common "memory corruption" issues:

- Buffer overflows.
- Return pointer overwrites.
- Signed/unsigned confusion.
- Off-by-one errors.
- Memory corruption on the heap.

And some mitigations:

- ASLR
- Stack canaries.

... and workarounds!

## Out-of-bounds memory access

In Python:

```
>>> a = [ 1, 2, 3 ]
>>> print a[3]
IndexError: list index out of range
```

In C:

```
int a[3] = { 1, 2, 3 };
printf("%d\n", a[3]);
// no problem!
```

Why does C let this go?
What happens here?

# Out-of-bounds memory write

In Python:

```
>>> a = [ 1, 2, 3, 4 ]
>>> a[4] = 1
IndexError: list index out of range
```

In C:

```
int a[3] = { 1, 2, 3, 4 };
a[4] = 1;
// no problem!
```

What happens here?
Why is it bad? Let's look in gdb!

## Out-of-bounds memory write

When do buffer overflows happen?

- Lazy/insecure programming practices (gets, strcpy, scanf, snprintf, etc).
- Passing pointers around without their size.
- Size calculation errors (more on this later).
- ... so many other reasons

So what?

## Out-of-bounds memory write

What can we do with an out-of-bound memory write?

... almost anything!

Worst-case:

- Overwrite program data to influence logic (leading to further vulnerabilities).
- Overwrite control flow data (saved return address, function pointers) to hijack control flow.

Let's do a demo!

## Return pointer overwrites

Ultimate power: overwriting the return address of a program to control what it executes next.

Lets you jump to arbitrary functions.

What else can you do?

- Lets you jump to arbitrary *instructions*.
- Lets you *chain functionality*.
- (On x86 and amd64) lets you jump *between* instructions...

## Out-of-bounds memory write

When do buffer overflows happen?

- Lazy/insecure programming practices (getc, strcpy, scanf, snprintf, etc).
- Passing pointers around without their size.
- **Size calculation errors (more on this NOW).**
- ... so many other reasons

So what?

## Signedness Mixups

The standard C library uses *unsigned integers* for sizes (i.e., the last argument to **`read`**, memcmp, strncpy, and others).

The default integer types (**`short`**, **`int`**, **`long`**) are *signed*.

Why is this a problem?

```
    int size;
    char buf[16];
    scanf("%i", &size);
    if (size > 16) exit(1);
    read(0, buf, size);
```

Let's look at a demo!

## Off-by-one Errors

Like many things in C, strings do not have explicit size metadata in memory.

To solve this, they are *null-terminated*.

```
char name[4] = "Yan";
// stored as hex 59 61 6e 00
```

Problem: people often forget the null byte!

```
char name[10] = {0};
printf("Name: ");
read(0, name, 10);
printf("Hello %s!\n", name);
```

Let's see what happens here...

## The rise of memory corruption mitigations

With so much C around, and with C so easily buggy, some thought has been given to *memory corruption mitigation*.

Philosophy: let's assume there are memory corruption bugs, but let's make them hard to exploit!

- ASLR (and PIE).
- Stack canaries.
- Non-executable stack (we'll talk about this next week).

These raise the bar significantly!

## ASLR

How do you redirect execution if you don't know where any code is?

Glad you asked!

Method 1:

- The addresses still (mostly) have to be in memory so that the program can find its own assets.
- Let's leak them out!

Method 2:

- Program assets are *page-aligned*.
- Let's just overwrite the page offset!
- Requires some brute-forcing.

# Overwriting Page Offsets

Pages are (for the most part) 0x1000 bytes on modern architectures.
Pages are also always *aligned* to a 0x1000 alignment.

Possible page addresses:

```
    0x00007f8dce27f000 0x56531c9c5000 0xffffffffff600000 0x400000
```

Upshot is: the last three *nibbles* of an address are never changed.

If we overwrite the two least significant bytes of a pointer, we only have to brute-force *one nibble* (16 possible values) to successfully redirect the pointer to another location on the same page.

With *little endian*, these are the first two bytes that we will ovewrite!

## Stack Canaries

To fight buffer overflows into the return address, researchers introduced *stack canaries*.

1. In function prologue, write random value at the end of the stack frame.
2. In function epilogue, make sure this value is still intact.

Very effective in general.
Bypass methods:

1. Jumping the canary.
2. Just go with the flow! (who guards the guards?)
    a. Let the canary check trigger, but mess with the alerting functionality.

# Homework 3 - Pwnables!

Now that you are all reverse engineering experts, it's time to start pwning these challenges!!

The homework:

- netcat to cse466.pwn.college, port 23 (the password will be posted on the class mailing list)
- enter your hacker alias (you can choose a new one) and ASU ID.
- Choose one of your personalized 33 challenges (in /pwn) to attempt.
- Each challenge contains one (intentional) memory corruption vulnerability. Exploit it get the challenge to read "/flag"!
- Log out ("exit" command).
- Provide the flag when asked.
- Scoring is done automatically.
- **3 points** per challenge.