Computer Systems Security

CSE 466
Fall 2018
Yan Shoshitaishvili

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

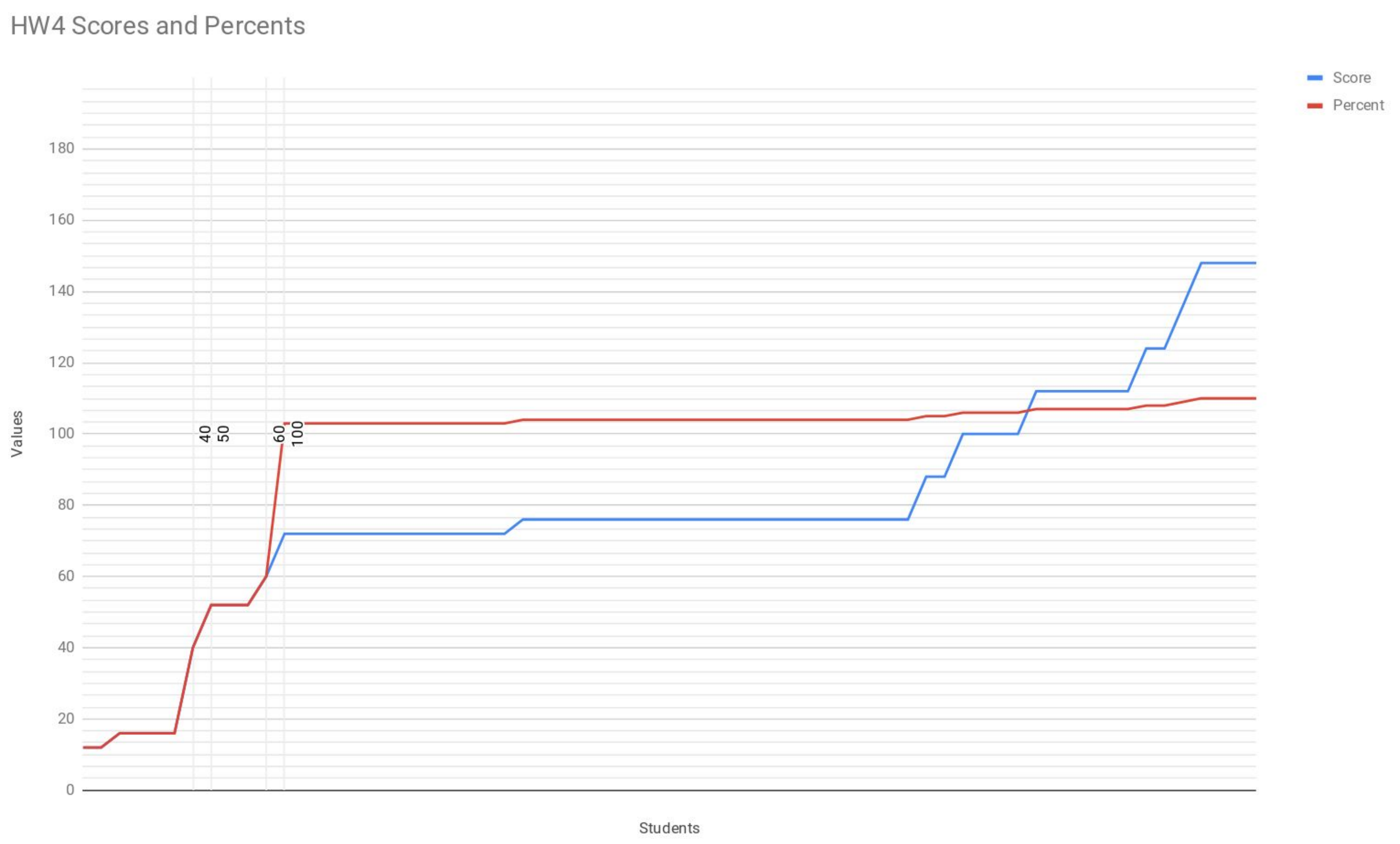
Homework 4 Retrospective

Statistics

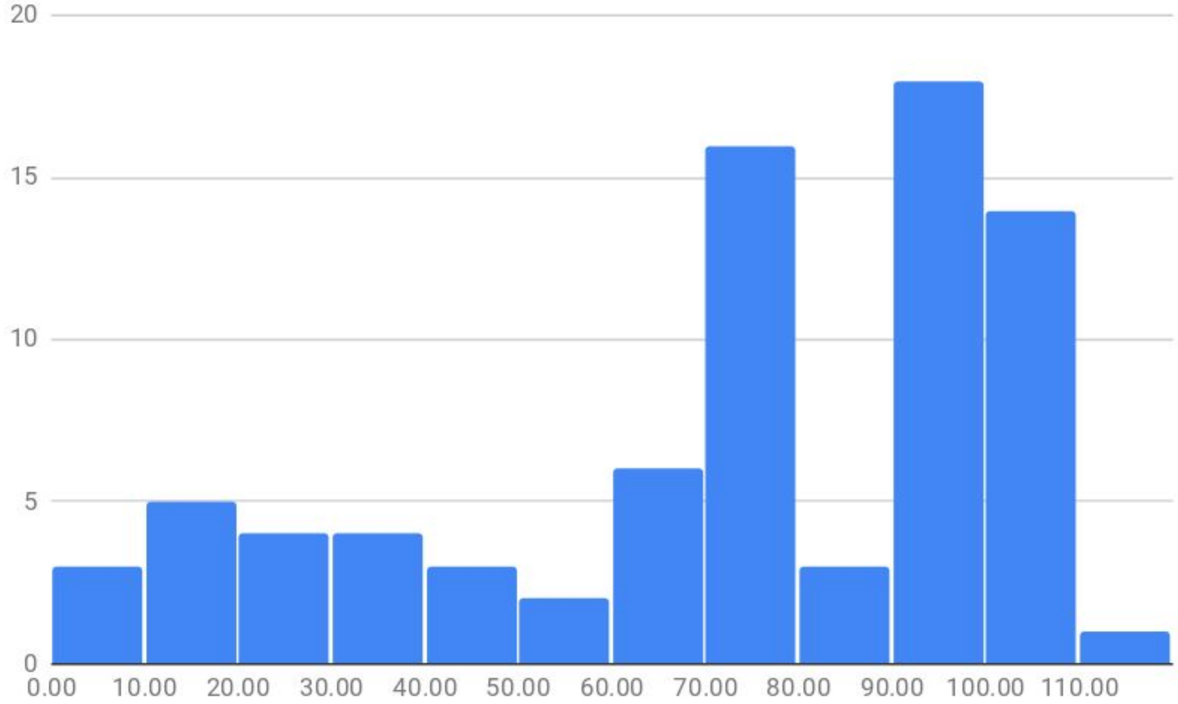
79 graded students (+1 auditing)

54 A+ (>= 99%)
0 (>= 90%)
0 B (>= 80%)
0 C (>= 70%)
1 D (>= 60%)
10 E (< 60%)
7 just logged in (0%)
7 did not attempt

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


HW4 Scores and Percents
Score
Percent
Values
Students
0
20
40
60
80
100
120
140
160
180
40
50
60
100

Student grade distribution



Takeaways

Compensating for difficulty is hard.

- On Monday, everyone was failing.
- Surprising amount of panicked emails.

Adjustments:

- added "walkthrough" challenges (did these help?)
- raised flag values
- emergency office hours

Going forward:

- more walkthrough challenges
- less panicking on my end

Master Hackers

Four-way tie between master hackers (sorted alphabetically):

1. clasm
2. dasher
3. i_use_archlinux
4. git_me_that_hug

How did they do it?

CSE 466 Week 5

Reversing and Pwning Recap

Concept: Program Interaction

HW3 focused on different ways of interacting with programs.

HW4 kept it to stdin/stdout.

Review: what is stdin and stdout?

System call operation

Some of HW3 depended on understanding how to interrupt the sleep() system call.

Some of HW4 depended on understanding *short reads*.

How do short reads happen?

- network services
- Ctrl-D
- EOF

Buffering

All HW4 challenges had a bug that broke the buffering:

```
int main(int argc, char **argv, char **envp)
{
	puts("==================================================");
	printf("\tWelcome to %s!\n", argv[0]);
	puts("==================================================");
	setvbuf(stdin, NULL, _IONBF, 0);
	setvbuf(stdin, NULL, _IONBF, 1);

	return vuln(argc, argv, envp);
}
```

Can you see the bug?

Workarounds:

- stdbuf command apparently uses LD_PRELOAD, but unbuffer does not.
- really, buffering is **sorcery** that is impossible to truly comprehend

Last week week...

We took a first glance at 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.

Hands-on Experimentation

Homework 5 - Makeup Challenges

If you had trouble with the previous two homeworks, here is your chance.

The homework:

- Standard setup.
- **3 points** per challenge.
- Has a mix of reverse-engineering and pwnable challenges.
- Walkthroughs for almost every level.
- The lowest grade of this homework is capped at your average score score on the previous two homeworks.
- If you are happy with your prior average score, you can relax this week.

First four (pwning) levels up now, remaining levels will be pushed tonight (walkthroughs are time-consuming to write).