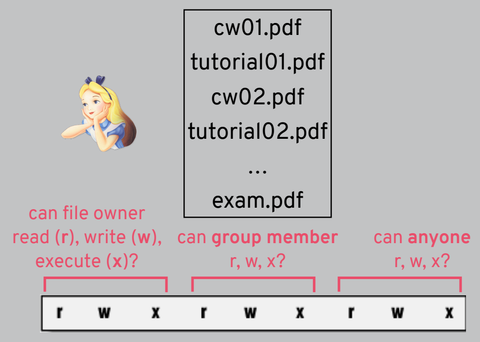
SECURITY (COMP0141): SOFTWARE SECURITY

UNIX PERMISSIONS



2

Security in file systems has a lot to do with access control, so let's start by revisiting the main access control mechanism we saw in Week 8, which is UNIX permissions. We saw them just as a mechanism for achieving security but now we should analyse how effective they are in this goal

THREAT MODEL

Motivation:

- **Run “unauthorised” code:** take over machine
- **Access “unauthorised” files:** exceed given permissions

Capabilities:

- **Access machine:** starts with one set of credentials
- **Execute system-defined programs:** like `passwd`
- **Execute user-defined programs:** written by adversary

permissions

permissions + setuid

?

Unix permissions are effective at addressing the weaker goal of accessing files and the weakest capability, and if setuid is used effectively (and without vulnerabilities) this addresses the second type of capability too. But what about the strongest possible adversary and the strongest goal of compromising the machine?

SECURITY DESIGN

define program

How to ~~design~~ a secure ~~system~~?

one that meets a specific security policy

A lot of what we’ve covered recently has been quite broad, in terms of broader systems like organisations. Today though we’re going to zoom way way in to the low-level question of programs. Defining security for these seems much harder because there is no inherent policy here

WHEN IS A PROGRAM SECURE?

When it does exactly what it **should**
- Not more
- Not less

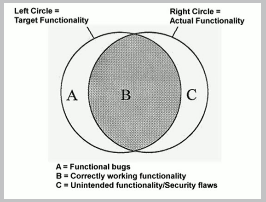
But… what should a program do? How do we know?
- Somebody tells us (do we trust them?)
- We write the code ourselves (how often is this true?)

Okay, a program is secure when it doesn't do "bad things"
- Crash my system
- Delete or corrupt important files
- Send my password over the Internet

But what if it **could** do bad things? Is it still secure?

UNINTENDED FUNCTIONALITY

An **exploit** is a mechanism by which an attacker triggers some unintended functionality of the system (blind spot for the developer)



Security involves understanding both the intended and unintended functionalities of the system

WHAT MAKES SECURITY SPECIAL?

Correctness: For a given input, a program should provide the correct output

Safety: Well-formed programs cannot have bad (wrong or dangerous) outputs, no matter the input

Robustness: Programs should be able to cope with errors in execution

These properties must hold even in the presence of a **resourceful** and **strategic** adversary

This really goes right back to what I said in Week 1, security is all about looking at these unintended functionalities

---

SOFTWARE VULNERABILITY

An **exploit** is a mechanism by which an attacker triggers some unintended functionality of the system (blind spot for the developer)

A **software vulnerability** is a bug in a program that allows a user capabilities that should be denied to them

One very common type of vulnerability is ones that violate **control flow integrity** (CFI)

Today we'll look at **buffer overflows**

A bug without security implications isn't a vulnerability, it's just a bug

BUFFER OVERFLOWS

Program variables get allocated regions of physical memory in the form of **buffers**

Buffer overflows happen when a program writes data beyond its allocated buffers

These are ubiquitous in systems-level languages (C/C++), made worse by the fact that many standard library functions make it easy to go beyond array bounds

String functions like `strcpy()` and `strcat()` write to the destination buffer until they encounter `\0` in input, so the user providing the input (who can easily be the attacker!) controls how much gets written

Fun fact: the standard string functions in C are completely insecure

---

EXAMPLE: STRCPY

```
char A[8] = "";
unsigned short B = 1979;
```

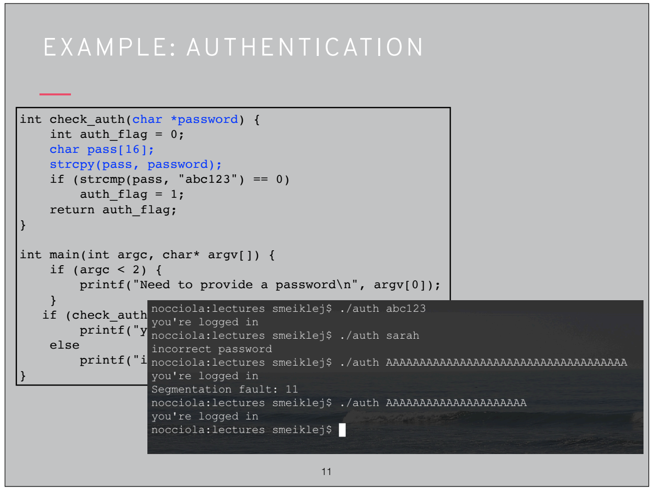
variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

```
strcpy(A, "excessive");
```

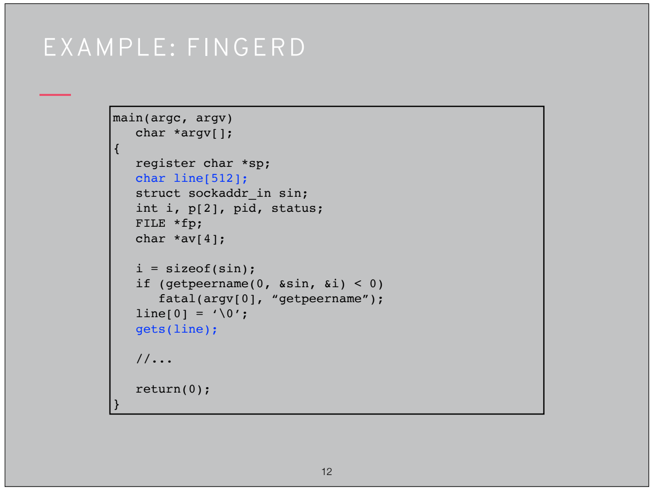
variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

The extra characters changed the value in the buffer for B

This is the example from the Wikipedia article on buffer overflows (https://en.wikipedia.org/wiki/Buffer_overflow) so feel free to read more about it there



In this example the password buffer is only 16 bytes long, so if we overflow it we get unintended behaviour: either a segfault (using 36 As) or, worse, a successful login without the right password (using 21 As)



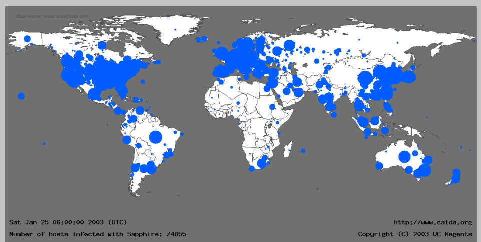
Similar vulnerability here: arbitrary user input fed into 512-byte buffer using gets (which is unsafe), so can easily overflow

MORRIS WORM

first (accidental) worm (1988)
required **the entire Internet** to reboot

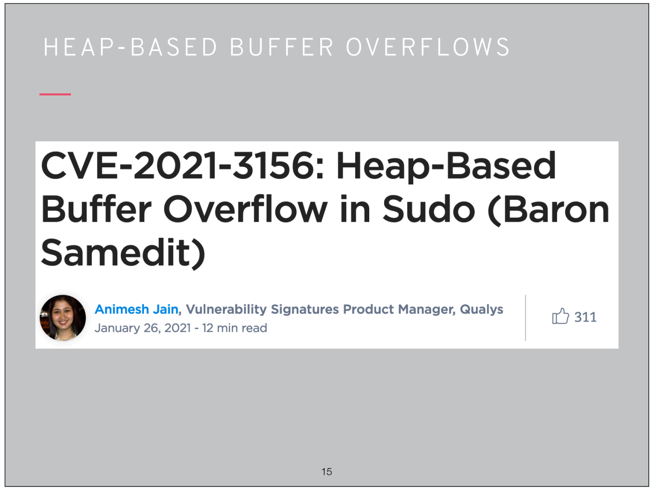
This is one of the vulnerabilities that was exploited by Robert Morris in the worm we heard about in Week 5

EXPANDING BOTNET: WORMS



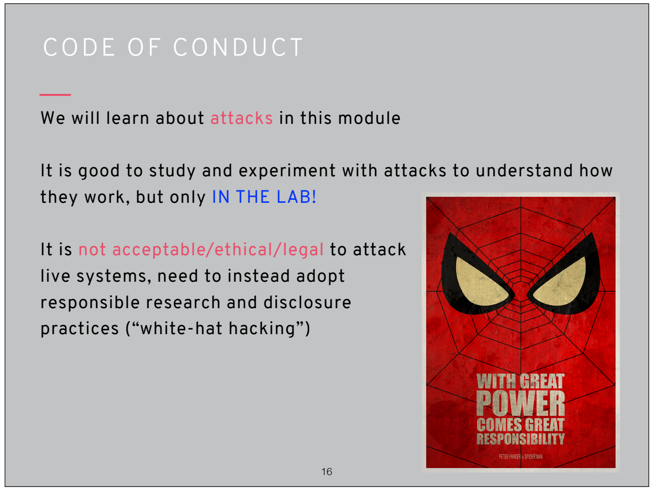
spread autonomously by exploiting vulnerabilities
spread quickly and unpredictably, easy to detect
Slammer worm infected 75,000 within 10 minutes

Slammer also worked by exploiting a buffer overflow, and in fact many other worms do too



Buffer overflows still happen in some form today! Can read about this recent vulnerability at https://blog.qualys.com/vulnerabilities-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit

---



Here is a reminder: you can do this (carefully!) on your own laptop or in some contained environment, but never otherwise – even something that seems like you're just testing stuff out can get out of hand and have serious consequences

BUFFER OVERFLOWS

Why should buffer overflows allow you to take over the machine?

Your program manipulates data...

...but data also manipulates your program!