SECURITY (COMP0141):
SOFTWARE SECURITY

UNIX PERMISSIONS

cw01.pdf

tutorial01.pdf

cw02.pdf

tutorial02.pdf

...

exam.pdf

can file owner
read (**r**), write (**w**),
execute (**x**)?

can **group member**
r, w, x?

can **anyone**
r, w, x?

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

?

define program

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

one that meets a specific security policy

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?)

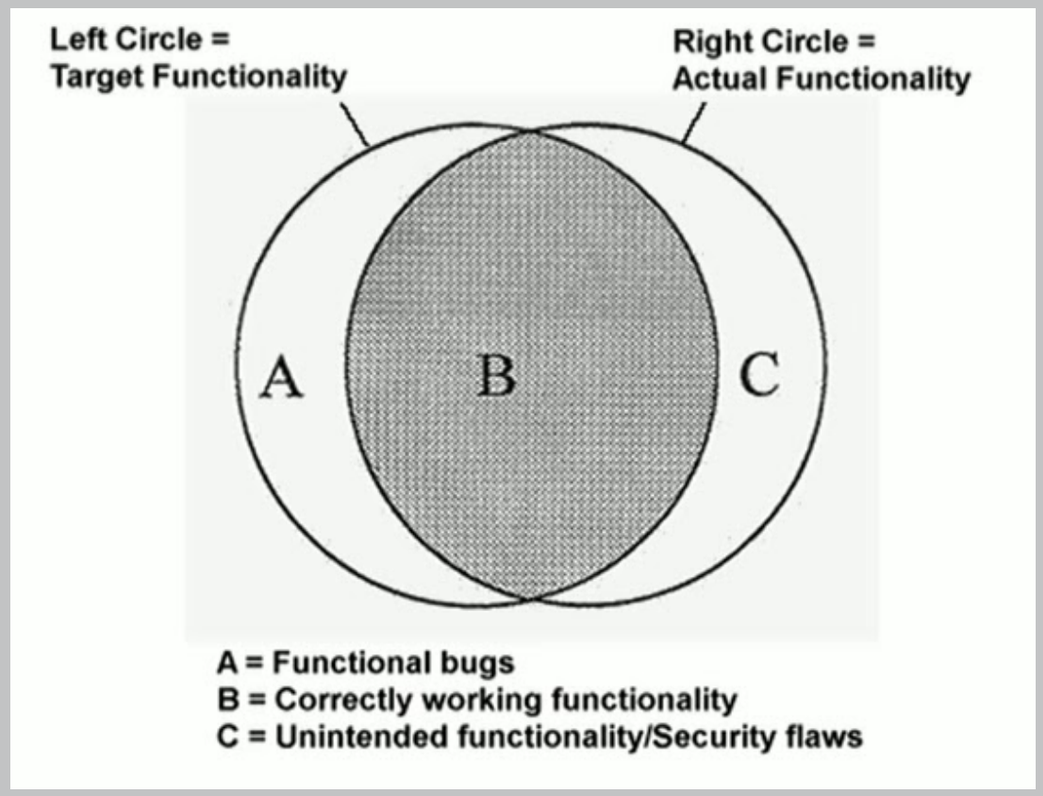
Okay, a program is secure when it doesn't do "bad things"

- 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

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**

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

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

EXAMPLE: AUTHENTICATION

```
int check_auth(char *password) {
    int auth_flag = 0;
    char pass[16];
    strcpy(pass, password);
    if (strcmp(pass, "abc123") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Need to provide a password\n", argv[0]);
    }
   if (check_auth
        printf("y
    else
        printf("i
}
```

```
nocciola:lectures smeiklej$ ./auth abc123
you're logged in
nocciola:lectures smeiklej$ ./auth sarah
incorrect password
nocciola:lectures smeiklej$ ./auth AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
you're logged in
Segmentation fault: 11
nocciola:lectures smeiklej$ ./auth AAAAAAAAAAAAAAAAAAAA
you're logged in
nocciola:lectures smeiklej$
```

EXAMPLE: FINGERD

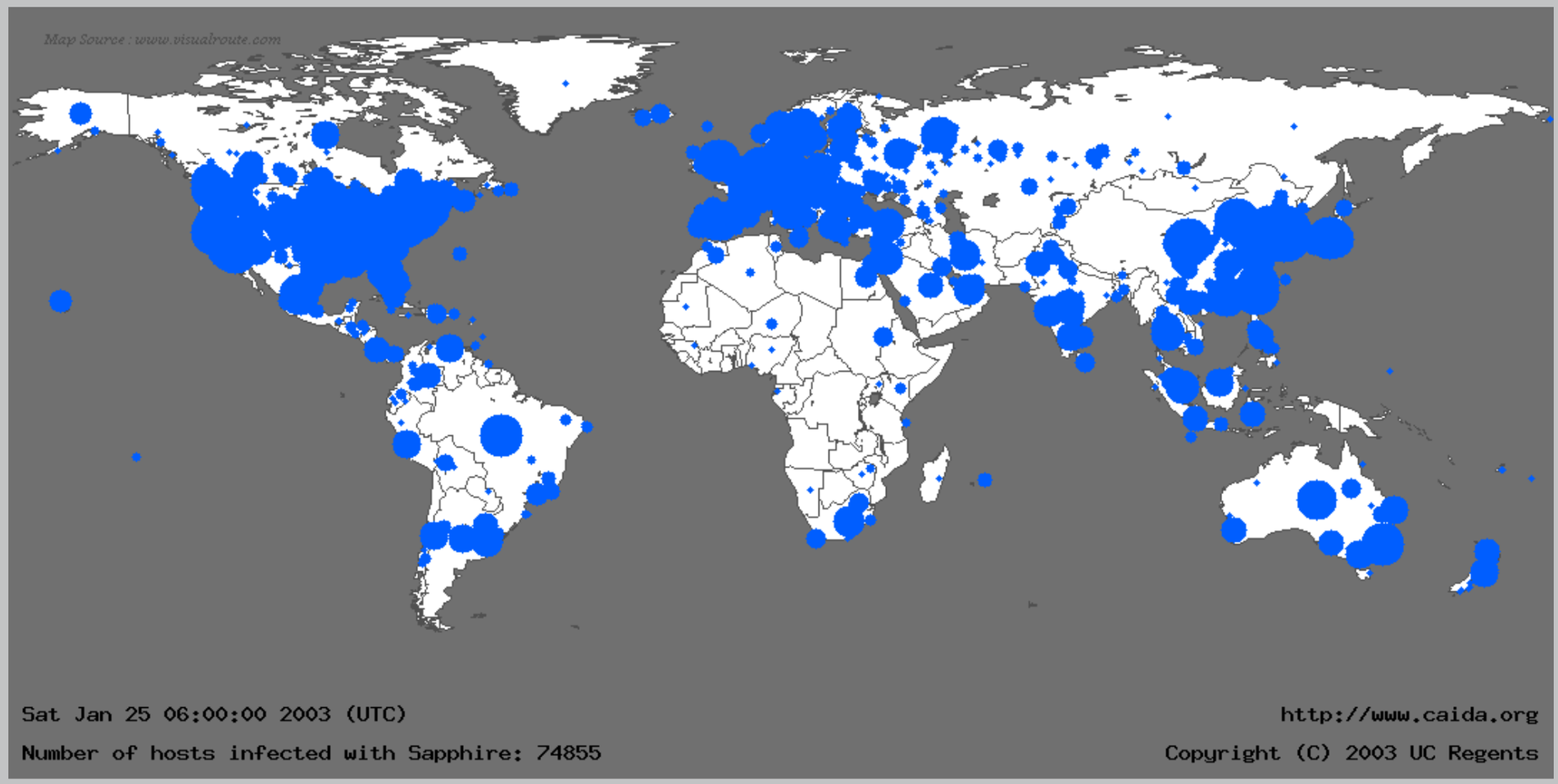
```
main(argc, argv)
   char *argv[];
{
   register char *sp;
   char line[512];
   struct sockaddr_in sin;
   int i, p[2], pid, status;
   FILE *fp;
   char *av[4];

   i = sizeof(sin);
   if (getpeername(0, &sin, &i) < 0)
      fatal(argv[0], “getpeername”);
   line[0] = ‘\0’;
   gets(line);

   //...

   return(0);
}
```

MORRIS WORM

first (accidental) worm (1988)

required **the entire Internet** to reboot

EXPANDING BOTNET: WORMS



spread autonomously by exploiting vulnerabilities

spread quickly and unpredictably, easy to detect

Slammer worm infected 75,000 within 10 minutes

CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit)

Animesh Jain, Vulnerability Signatures Product Manager, Qualys

January 26, 2021 - 12 min read

311

CODE OF CONDUCT

We will learn about attacks in this module

It is good to study and experiment with attacks to understand how they work, but only IN THE LAB!

It is not acceptable/ethical/legal to attack live systems, need to instead adopt responsible research and disclosure practices (“white-hat hacking”)

BUFFER OVERFLOWS

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

Your program manipulates data...

...but data also manipulates your program!