CS 3650 Computer Systems – Summer 2025

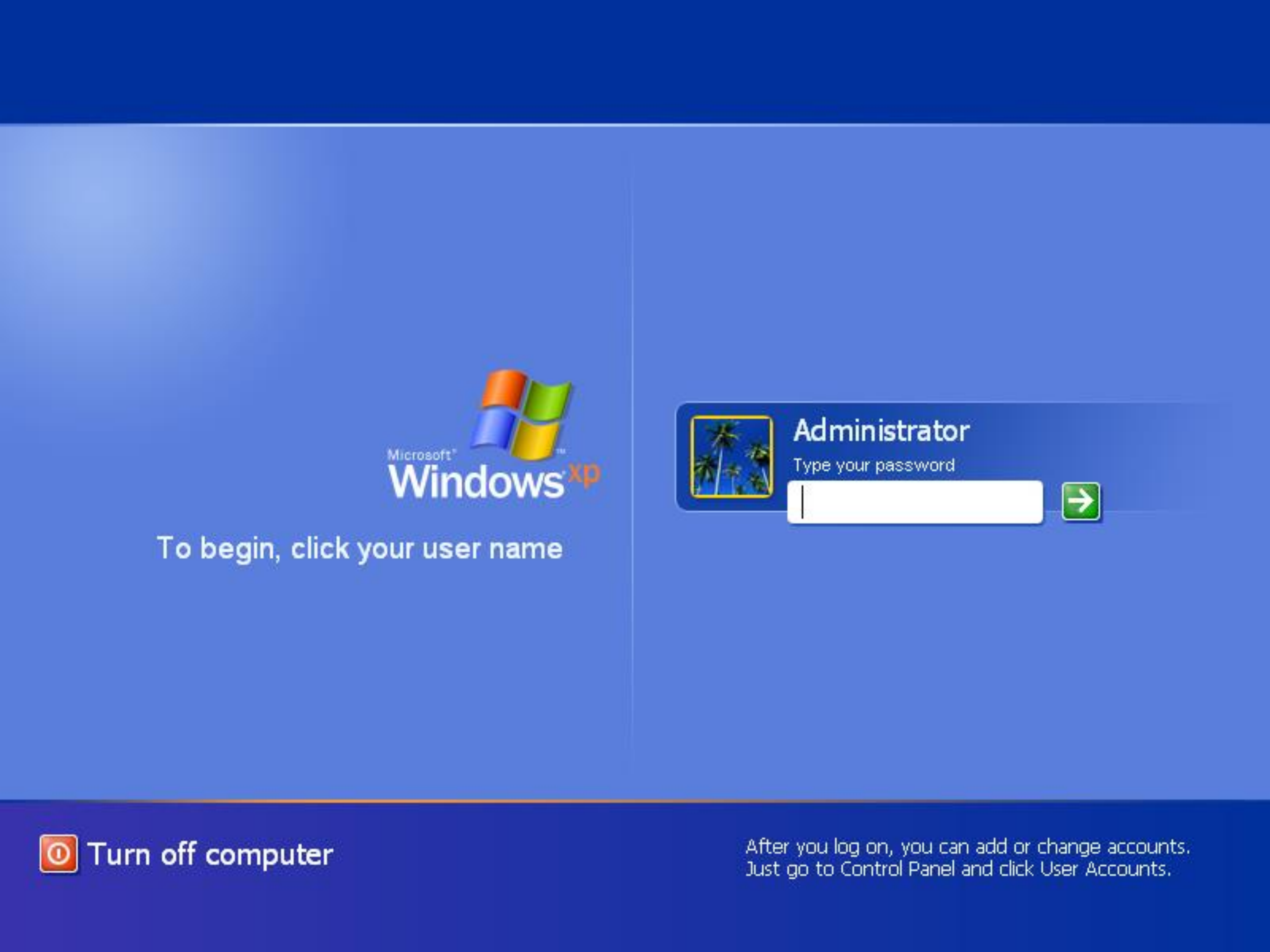
Security

Unit 14

Northeastern University

* Acknowledgements: created based on Christo Wilson's lecture slides for the same course.

- Authentication
- Access Control


Microsoft
Windows xp
To begin, click your user name
Administrator
Type your password
Turn off computer
After you log on, you can add or change accounts.
Just go to Control Panel and click User Accounts.

Authentication

- Authentication is the process of verifying an actor's identity
- Critical for security of systems
  - Permissions, capabilities, and access control are all contingent upon knowing the identity of the actor
- Typically parameterized as a username and a secret
  - The secret attempts to limit unauthorized access

Types of Secrets

- Actors provide their secret to log-in to a system
- Three classes of secrets:
  1. Something you know
     - Example: a password
  2. Something you have
     - Examples: a smart card or smart phone
  3. Something you are
     - Examples: fingerprint, voice scan, iris scan

Checking Passwords

- The system must validate passwords provided by users
- Thus, passwords must be stored somewhere
- Basic storage: plain text

password.txt	
cbw	p4ssw0rd
sandi	i heart doggies
amislove	93Gd9#jv*0x3N
bob	security

Problem: Password File Theft

- Attackers often compromise systems
- They may be able to steal the password file
  - Linux: /etc/shadow
  - Windows: c:\windows\system32\config\sam
- If the passwords are plain text, what happens?
  - The attacker can now log-in as any user, including root/administrator
- **Passwords should never be stored in plain text**

Hashed Passwords

- Key idea: store encrypted versions of passwords
  - Use one-way cryptographic hash functions
  - Examples: md5, sha1, sha256, sha512
- Cryptographic hash function transform input data into scrambled output data
  - Deterministic: hash(A) = hash(A)
  - High entropy:
    - md5('security') = e91e6348157868de9dd8b25c81aebfb9
    - md5('security1') = 8632c375e9eba096df51844a5a43ae93
    - md5('Security') = 2fae32629d4ef4fc6341f1751b405e45
  - Collision resistant
    - Locating A' such that hash(A) = hash(A') takes a long time
    - Example: 2^{21} tries for md5

Hashed Password Example

User: cbw

md5('p4ssw0rd') =
2a9d119df47ff993b662a8ef36f9ea20

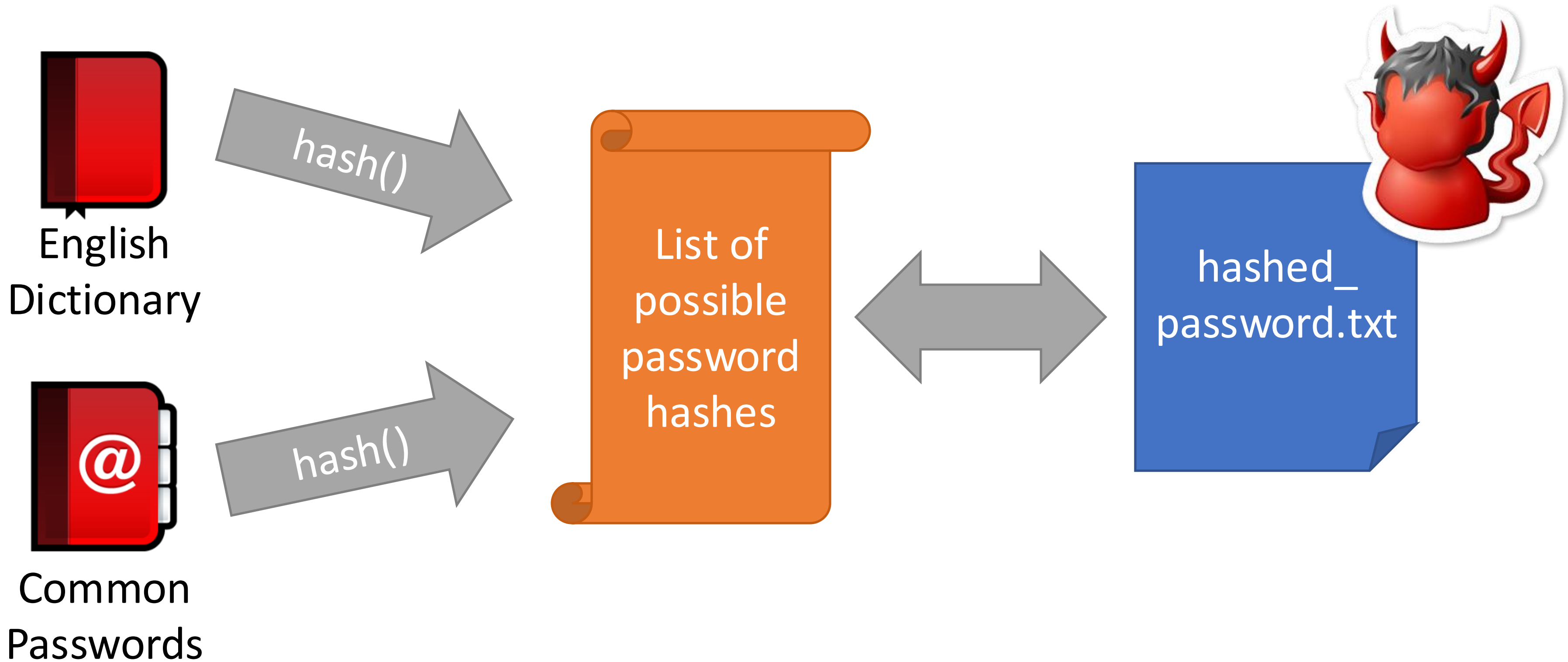
md5('2a9d119df47ff993b662a8ef36f9ea20')
= b35596ed3f0d5134739292faa04f7ca3

hashed_passw[illegible]xt	
cbw	2a9d119df47ff993b662a8ef36f9ea20
sandi	23eb06699da16a3ee5003e5f4636e79f
amislove	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9

Attacking Password Hashes

- Recall: cryptographic hashes are collision resistant
  - Locating A' such that hash(A) = hash(A') takes a very long time
- Are hashed password secure from cracking?
  - **No!**
- Problem: users choose poor passwords
  - Most common passwords: 123456, password
  - Username: cbw, Password: cbw
- Weak passwords enable dictionary attacks

Dictionary Attacks



- Common for 60-70% of hashed passwords to be cracked in <24 hours

Hardening Password Hashes

- Key problem: cryptographic hashes are deterministic
  - hash('p4ssw0rd') = hash('p4ssw0rd')
  - This enables attackers to build lists of hashes
- Solution: make each password hash unique
  - Add a salt to each password before hashing
  - hash(salt + password) = password hash
  - Each user has a unique, random salt
  - Salts can be stores in plain text

Example Salted Hashes

hashed_password.txt	
cbw	2a9d119df47ff993b662a8ef36f9ea20
sandi	23eb06699da16a3ee5003e5f4636e79f
amislove	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9

hashed_and_salted_password.txt		
cbw	a8	af19c842f0c781ad726de7aba439b033
sandi	0X	67710c2c2797441efb8501f063d42fb6
amislove	hz	9d03e1f28d39ab373c59c7bb338d0095
bob	K@	479a6d9e59707af4bb2c618fed89c245

Password Storage on Linux

/etc/passwd

username:x:UID:GID:full_name:home_directory:shell

cbw:x:1001:1000:Christo Wilson:/home/cbw/:/bin/bash
amislove:1002:2000:Alan Mislove:/home/amislove/:/bin/sh

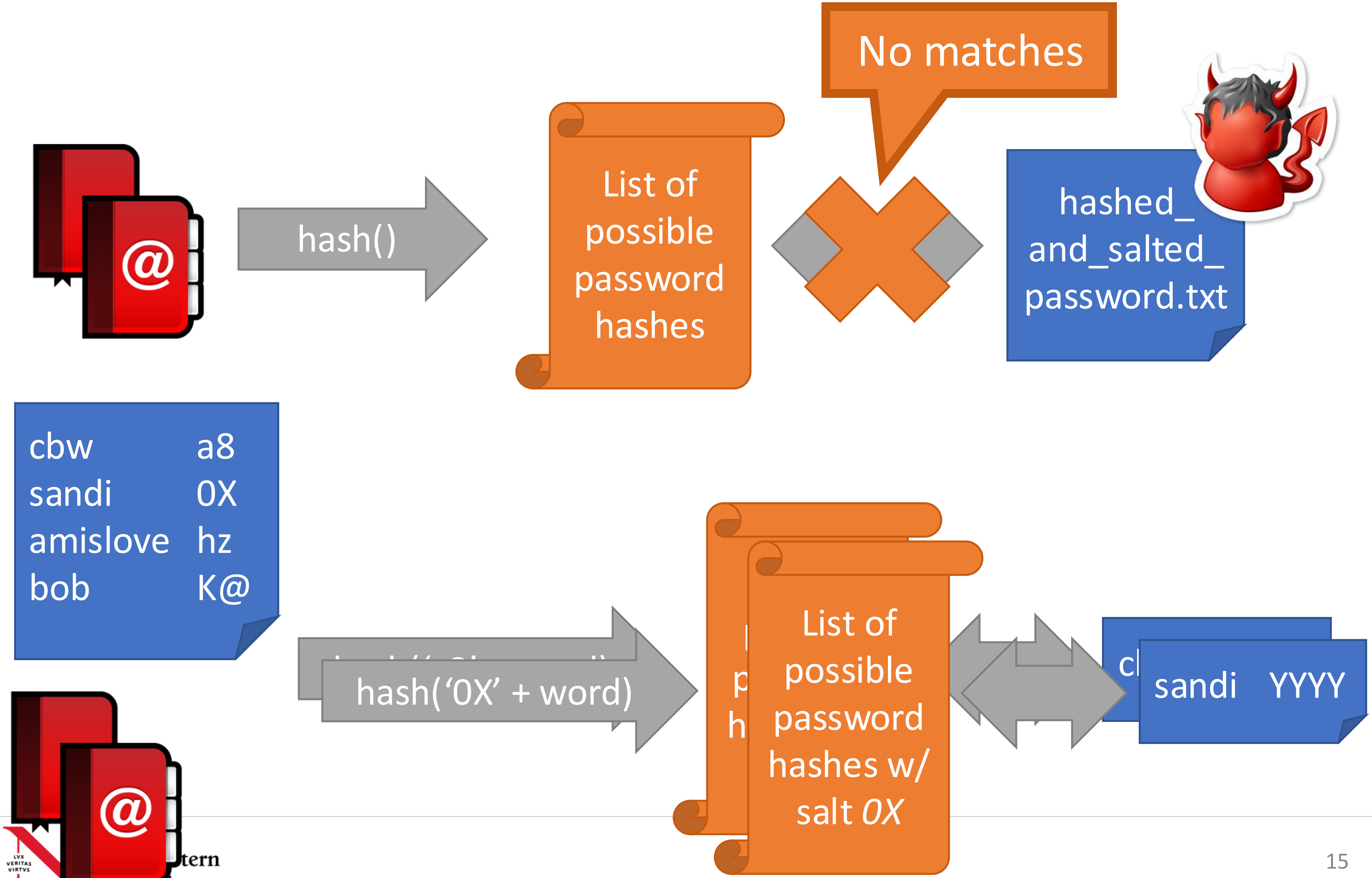
/etc/shadow

First two characters are the salt

username:password:last:may:must:warn:expire:disable:reserved

cbw:a8ge08pfz4wuk:9479:0:10000::::
amislove:hz560s9vnalh1:8172:0:10000::::

Attacking Salted Passwords



Breaking Hashed Passwords

- **Stored passwords should always be salted**
  - Forces the attacker to brute-force each password individually
- Problem: it is now possible to compute cryptographic hashes very quickly
  - GPU computing: hundreds of small CPU cores
  - nVidia GeForce GTX Titan Z: 5,760 cores
  - GPUs can be rented from the cloud very cheaply
    - 2x GPUs for $0.65 per hour (2014 prices)

Examples of Hashing Speed (2014)

- A modern x86 server can hash all possible 6 character long passwords in 3.5 hours
  - Upper and lowercase letters, numbers, symbols
  - $(26+26+10+32)^6$ = 690 billion combinations
- A modern GPU can do the same thing in 16 minutes
- Most users use (slightly permuted) dictionary words, no symbols
  - Predictability makes cracking much faster
  - Lowercase + numbers → $(26+10)^6$ = 2B combinations

Hardening Salted Passwords

- Problem: typical hashing algorithms are too fast
  - Enables GPUs to brute-force passwords
- Solution: use hash functions that are designed to be **slow**
  - Examples: bcrypt, scrypt, PBKDF2
  - These algorithms include a work factor that increases the time complexity of the calculation
  - scrypt also requires a large amount of memory to compute, further complicating brute-force attacks

bcrypt Example

- Python example; install the *bcrypt* package

```
[cbw@ativ9 ~] python
>>> import bcrypt
>>> password = "my super secret password"
>>> fast_hashed = bcrypt.hashpw(password, bcrypt.gensalt(0))
>>> slow_hashed = bcrypt.hashpw(password, bcrypt.gensalt(12))
>>> pw_from_user = raw_input("Enter your password:")
>>> if bcrypt.hashpw(pw_from_user, slow_hashed) == slow_hashed:
…       print "It matches! You may enter the system"
…   else:
…       print "No match. You may not proceed"
```

Work factor

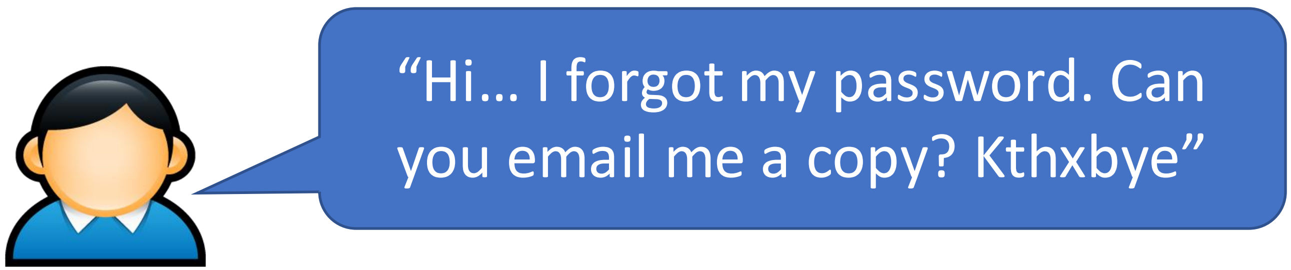
Password Storage Summary

1. **Never store passwords in plain text**
2. **Always salt and hash passwords before storing them**
3. **Use hash functions with a high work factor**

- These rules apply to any system that needs to authenticate users
  - Operating systems, websites, etc.

Password Recovery/Reset

- Problem: hashed passwords cannot be recovered



- This is why systems typically implement password reset
  - Use out-of-band info to authenticate the user
  - Overwrite hash(old_pw) with hash(new_pw)

- Authentication
- Access Control

Status Check

- At this point, we can authenticate users
  - And we are securely storing their password
- How do we control what users can do, and what they can access?

Simple Access Control

- Basic security in an OS is based on access control
- Simple policies can be written as an access control matrix
  - Specifies actions that actors can take on objects
  - Unix actions: read, write and execute
    - For directories, x → traverse

	file 1	file 2	dir 1	file 3
user 1	---	r--	---	rw-
user 2	r--	r--	rwx	r--
user 3	r--	r--	---	---
user 4	rw-	rwx	---	---

Users and Groups on Unix

- Actors are users, each user has a unique ID
  - Users also belong to >=1 groups

/etc/passwd
cbw:x:13273:65100:Christo Wilson:/home/cbw/:/bin/bash

```
[cbw@finalfight ~] id cbw
uid=13273(cbw) gid=65100(faculty) groups=65100(faculty),
1314(cs5700f13),1316(cs5750f13),1328(cs5600sp13)
```

File Permissions on Unix

- Files and directories have an owner and a group
- Three sets of permissions:
    1. For the owner
    2. For members of the group
    3. For everybody else (other)

```
[cbw@finalfight ~] ls -lh
-rw-r--r--   1 cbw   faculty 244K Mar   2 13:01 pintos.tar.gz
drwxr-xr--   3 cbw   faculty 4.0K Mar   2 13:01 pintos
```

Owner

Group

Number of links

Group permissions

Owner permissions

File or directory?

Permission Examples

```
[cbw@finalfight ~] ls -lh
-rw-r--r--  1 cbw  faculty 244K Mar  2 13:01 pintos.tar.gz
drwxr-xr--  3 cbw  faculty 4.0K Mar  2 13:01 pintos
```

cbw:faculty

- May read both objects
- May modify the file
- May not execute the file
- May enter the directory
- May add files to the directory
- May modify the permissions of both objects

amislove:faculty

- May read both objects
- May not modify the file
- May not execute the file
- May enter the directory
- May not add files to the directory
- May not modify permissions

bob:student

- May read both objects
- May not modify the file
- May not execute the file
- May not enter the directory
- May not add files to the directory
- May not modify permissions

Modifying Permissions

u – user
g – group
o - other

+ add permissions
- remove permissions
= set permissions

r – read
w – write
x - executable

• Us

```
[cbw@finalfight ~] ls -lh
-rw------- 1 amislove faculty 5.1K Jan 23 11:25 alans_file
-rw------- 4 cbw      faculty 3.5K Jan 23 11:25 christos_file
[cbw@finalfight ~] chmod ugo+rw alans_file
chmod: changing permissions of `alans_file': Operation not permitted
[cbw@finalfight ~] chmod go+r christos_file
[cbw@finalfight ~] chmod u+w christos_file
[cbw@finalfight ~] chmod u-r christos_file
[cbw@finalfight ~] ls -lh
-rw------- 1 amislove faculty 5.1K Jan 23 11:25 alans_file
--wxr--r-- 4 cbw      faculty 3.5K Jan 23 11:25 christos_file
```

Modifying Users and Groups

```
[cbw@finalfight ~] id cbw
uid=13273(cbw) gid=65100(faculty) groups=65100(faculty),
1314(cs5700f13),1316(cs5750f13),1328(cs5600sp13)
[cbw@finalfight ~] ls -lh
-rw------- 4 cbw faculty 3.5K Jan 23 11:25 christos_file
[cbw@finalfight ~] chown cbw:cs5600sp13 christos_file
[cbw@finalfight ~] ls -lh
-rw------- 4 cbw cs5600sp13 3.5K Jan 23 11:25 christos_file
```

- Users may not change the owner of a file*
  - Even if they own it
- Users may only change to a group they belong to

* unless you are root

Permissions of Processes

- Processes also have permissions
  - They have to, since they read files, etc.
- What is the user:group of a process?
  1. ~~The user:group of the executable file?~~
  2. The user:group of the user running the process?
- Processes inherit the credentials of the user who runs* them
  - Child processes inherit their parent's credentials

* except when the program is setuid

Privileged Operations

- Other aspects of the OS may also require special privileges
- Fortunately, on Unix most aspects of the system are represented as files
  - E.g. /dev contains devices like disks
  - Formatting a disk requires permissions to /dev/sd*
- Processes may only signal other processes with the same user ID*
  - Otherwise, you could send SIGKILL to other user's processes

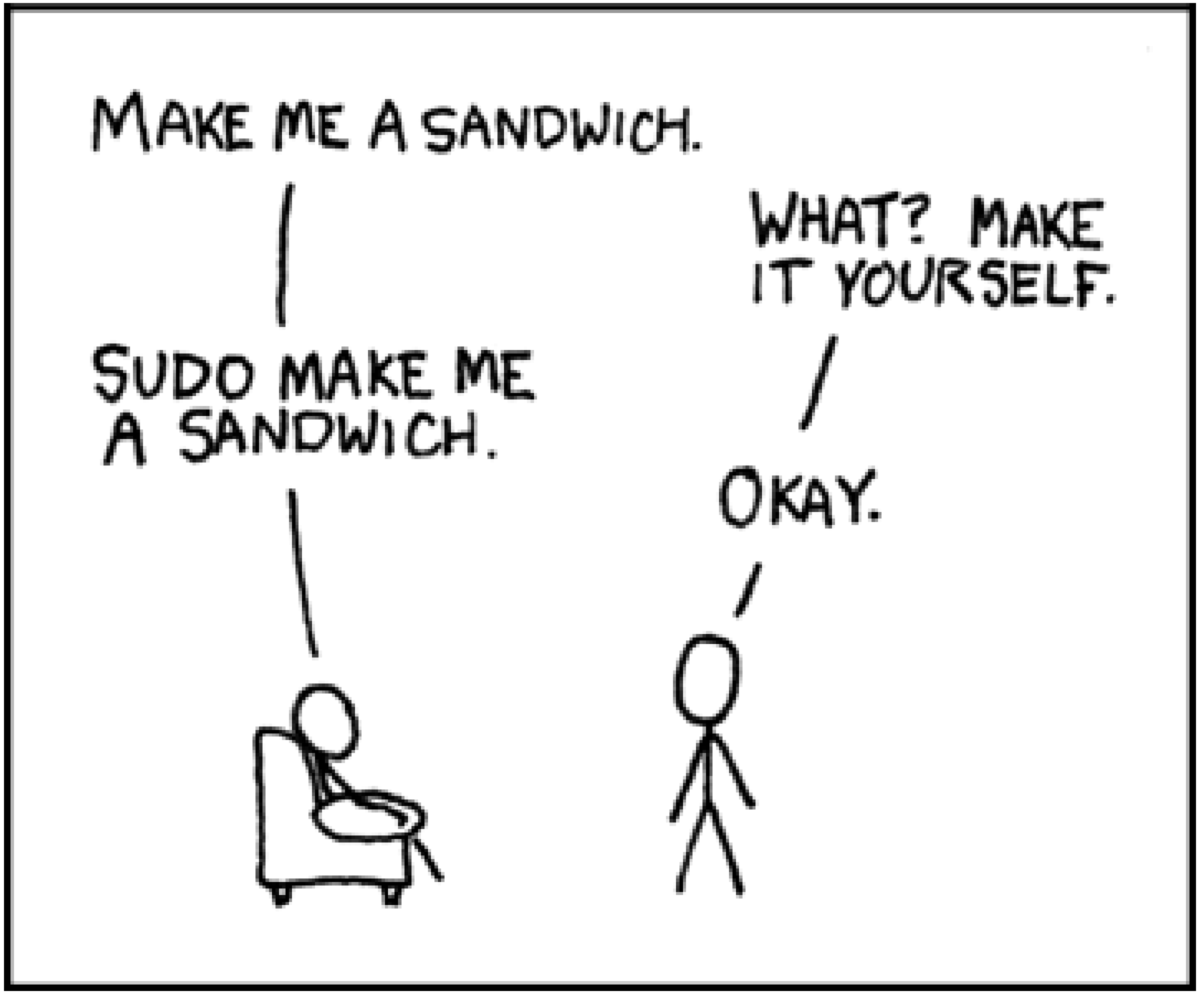
* unless the process is root

The Exception to Every Rule

- On Unix, the root user (ID=0) can do whatever it wants
  - Access any file
  - Change any permission
- On Windows, called the Administrator account
- **Your everyday user account should never be Admin/root**

Ways to Access Root

- Suppose you need to run a privileged command
  - Example: $ *apt-get install python*
- How can you get root privileges?
  1. Log in as root
     - $ ssh root@mymachine.ccs.neu.edu
  2. The Switch User command (su)
     - $ su
     - Opens a new shell with as root:root
  3. The Switch User Do Command (sudo)
     - $ sudo apt-get install python
     - Runs the given command as root:root


MAKE ME A SANDWICH.
WHAT? MAKE IT YOURSELF.
SUDO MAKE ME A SANDWICH.
OKAY.

Set Effective User ID

- In some cases, you may need a program to run as the file owner, not the invoking user
- Imagine a command-line guessing game
  - Users may input numbers as guesses
    - The user should not be able to read the file with the correct answers
  - Program must check if guesses are correct
    - The program must be able to read the file with correct answers

setuid example

Game executable is setuid

```
[cbw(     alfight game] ls -lh
-rw-    ----- 1 amislove faculty  180 Jan 23 11:25 secrets.txt
-rwsr-sr-x 4 amislove faculty 8.5K Jan 23 11:25 guessinggame
[cbw@finalfight game] cat secrets.txt
cat: secrets.txt: Permission denied
[cbw@finalfight game] ./guessinggame 4 8 15 16 23 42
Sorry, none of those number are correct :(
[cbw@finalfight game] ./guessinggame 37
Correct, 37 is one of the hidden numbers!
```

How to setuid

```
[cbw@finalfight tmp] gcc -o my_program my_program.c
[cbw@finalfight tmp] ls -lh
-rwxr-xr-x 1 cbw faculty 2.3K Jan 23 11:25 my_program
[cbw@finalfight tmp] chmod u+s my_program
[cbw@finalfight tmp] ls -lh
-rwsr-xr-x 1 cbw faculty 2.3K Jan 23 11:25 my_program
```

- **Be very careful with setuid**
  - You are giving other users the ability to run a program as **you**, with **your privileges**
- Programs that are setuid=root should drop privileges
  - Google “setuid demystified” for more info

setuid and scripts

```
[cb
-rw                                    1:25 server.py
```

This is known as a TOCTOU vulnerability: Time-Of-Check, Time-of-Use

- Steps to run a setuid script
  1. Kernel checks setuid bit of the script
  2. Kernel loads the interpreter (i.e. python) with setuid permissions
  3. Interpreter executes the script
- **Never set a script as setuid**

Replace server.py with modified, evil script

Limitations of the Unix Model

- The Unix model is very simple
  - Users and groups, read/write/execute
- Not all possible policies can be encoded

	file 1	file 2
user 1	---	rw-
user 2	r--	r--
user 3	rw-	rwx
user 4	rw-	---

- file 1: two users have high privileges
  – If user 3 and user 4 are in a group, how to give user 2 read and user 1 nothing?
- file 2: four distinct privilege levels
  – Maximum of three levels (user, group, other)

Access Control Lists

- ACLs are explicit rules that grant or deny permissions to users and groups
  - Typically associated with files as meta-data

	file 1	file 2
user 1	---	rw-
user 2	r--	r--
user 3	rw-	rwx
user 4	rw-	---

- file 1: owner = user 4, group = {user 4, user 3}

  owner: rw- group: rw-

  user 2: r-- other: ---

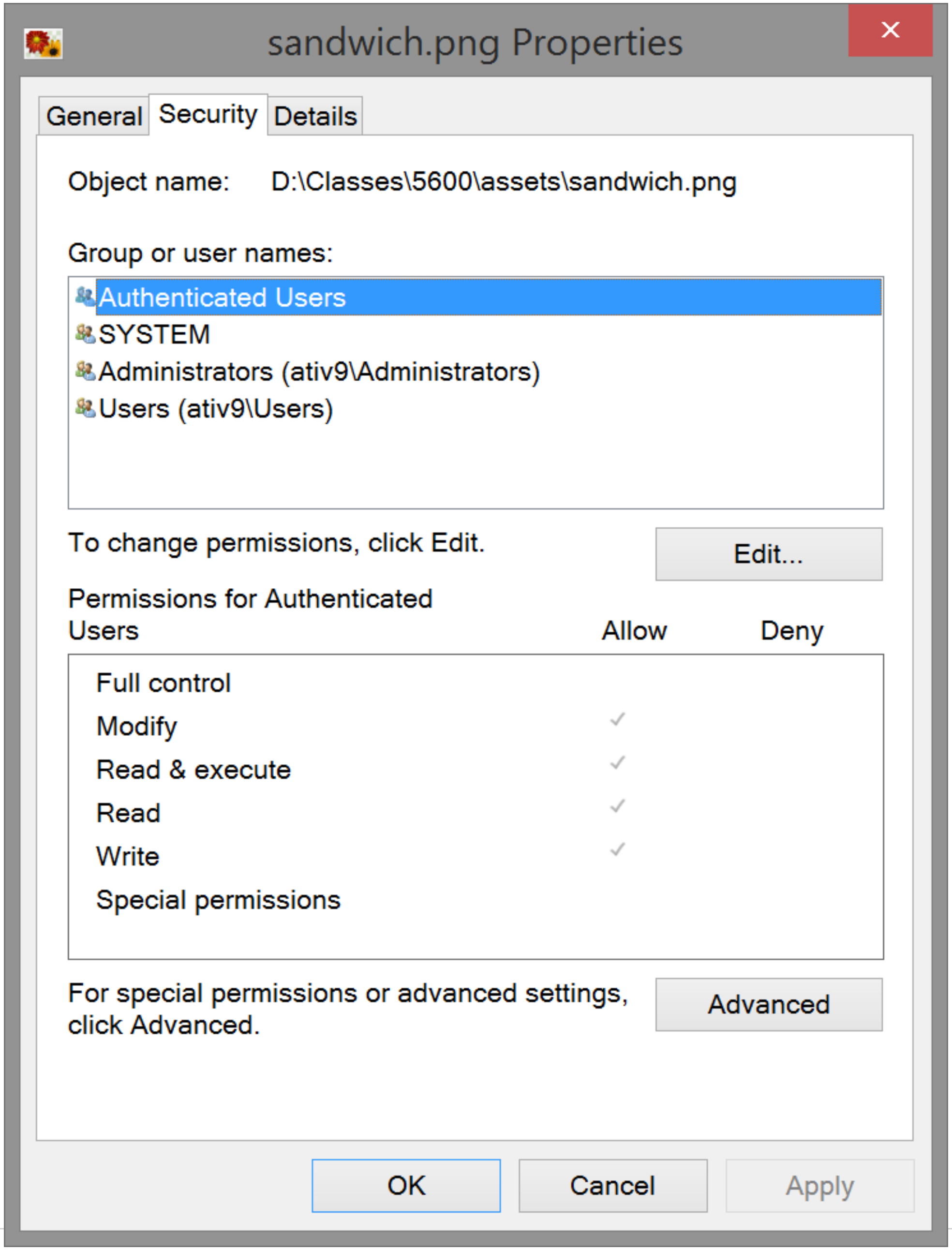
- file 2: owner = user 3, group = {user 3, user 1}

  owner: rwx group: rw-

  user 2: r-- other: ---

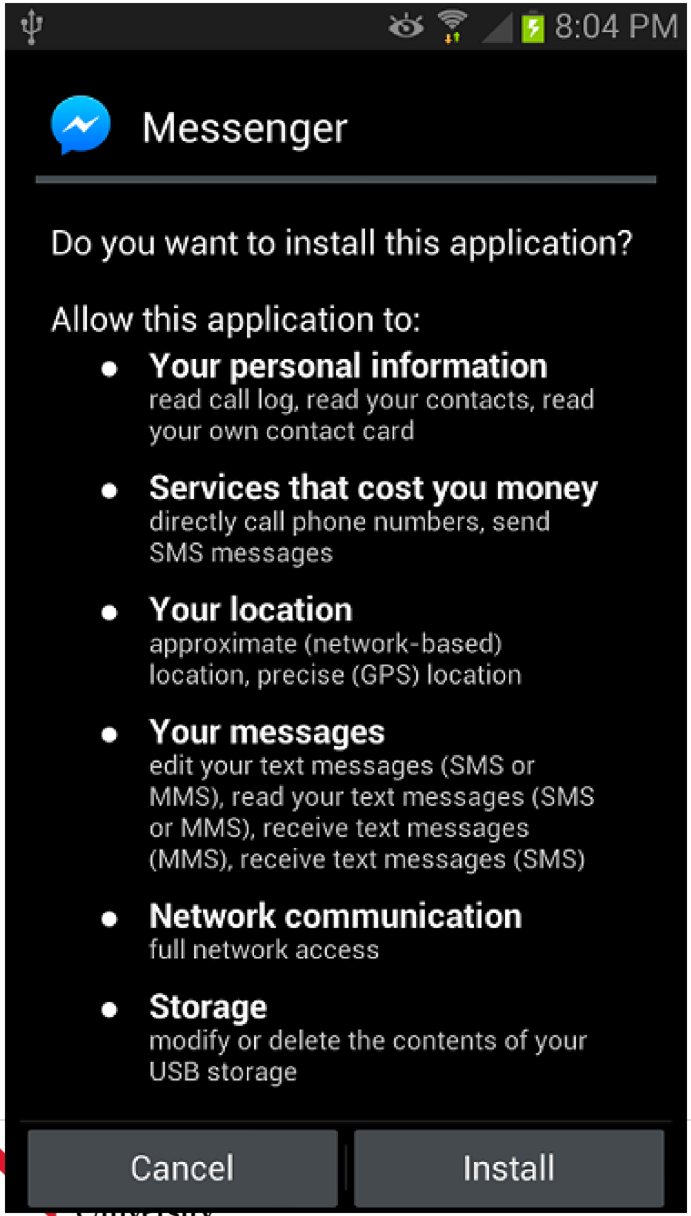
More ACLs

- OSX and some versions of Linux also support ACLs



Northeastern University

API Permissions



- On Android, apps need permission to access some sensitive API calls
- Android is based on Linux
- Behind the scenes, each app is given its own user and group
- Kernel enforces permission checks when system calls are made

Exploits and Exploit Prevention

- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP

Setting the Stage

Game executable is setuid

```
[cbw@   ifight game] ls -lh
-rw-  ----- 1 amislove faculty  180 Jan 23 11:25 secrets.txt
-rwsr-sr-x 4 amislove faculty 8.5K Jan 23 11:25 guessinggame
```

- Suppose I really want to see the secret answers
  - But I'm not willing to play the game
- How can I run arbitrary code as amislove?
  - If I could run code as amislove, I could read secrets.txt
  - Example: execvp("/bin/sh", 0);

Looking for Vulnerabilities

- Code snippet for *guessinggame*

```
char buf[8];
for (int x = 1; x < argc; ++x) {
    strcpy(buf, argv[x]);
    num = atoi(buf);
    check_for_secret(num);
}
```

Stack buffer overflow

Confirmation

```
[cbw@finalfight game] ls -lh
-rw------- 1 amislove faculty  180 Jan 23 11:25 secrets.txt
-rwsr-sr-x 4 amislove faculty 8.5K Jan 23 11:25 guessinggame
[cbw@finalfight game] ./guessinggame 1 2 3
Sorry, none of those number are correct :(
[cbw@finalfight game] ./guessinggame AAAAAAAAAAAAAAAAAAAAAAA
Sorry, none of those number are correct :(
Segmentation fault (core dumped)
```

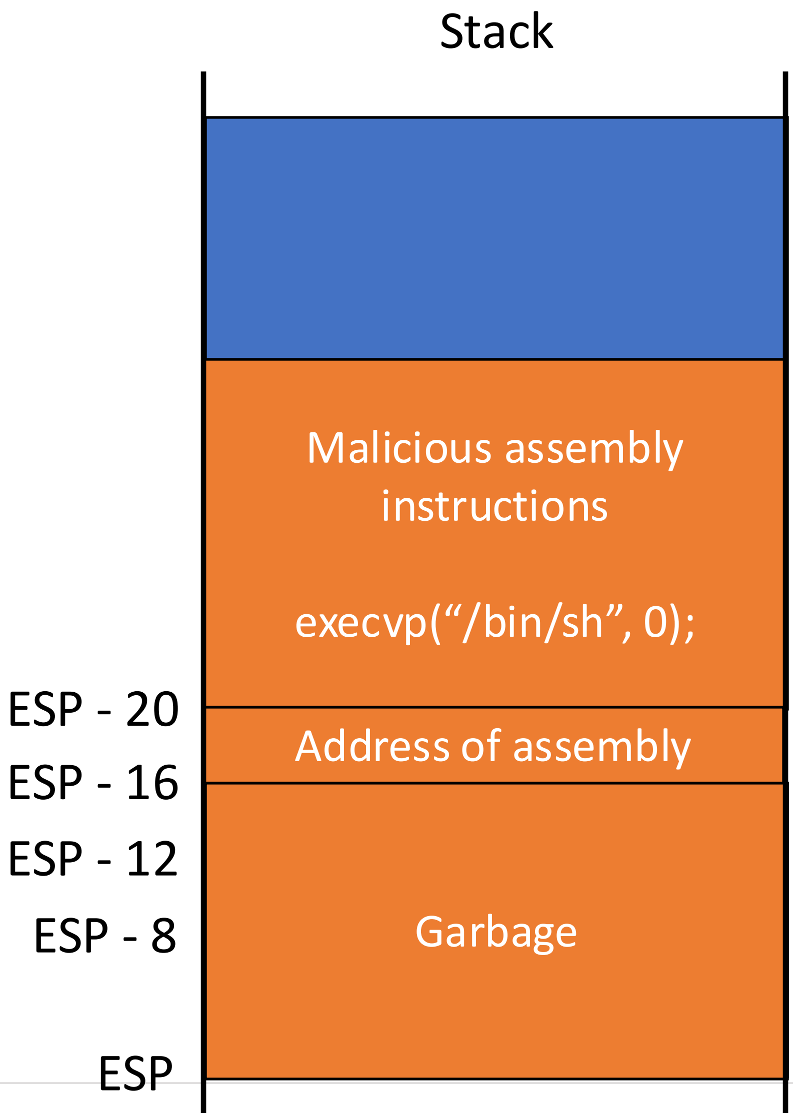
```
(gdb) bt
#0  0x0000000000400514 in myfunc ()
#1  0x4141414141414141 in ?? ()
#2  0x4141414141414141 in ?? ()
#3  0x4141414141414141 in ?? ()
#4  0x0000004141414141 in ?? ()
```

‘A’ = 0x41 in ASCII

Exploiting Stack Buffer Overflows

- Preconditions for a successful exploit

1. Overflow is able to overwrite the return address
2. Contents of the buffer are under the attackers control



Exploitation, Try #1

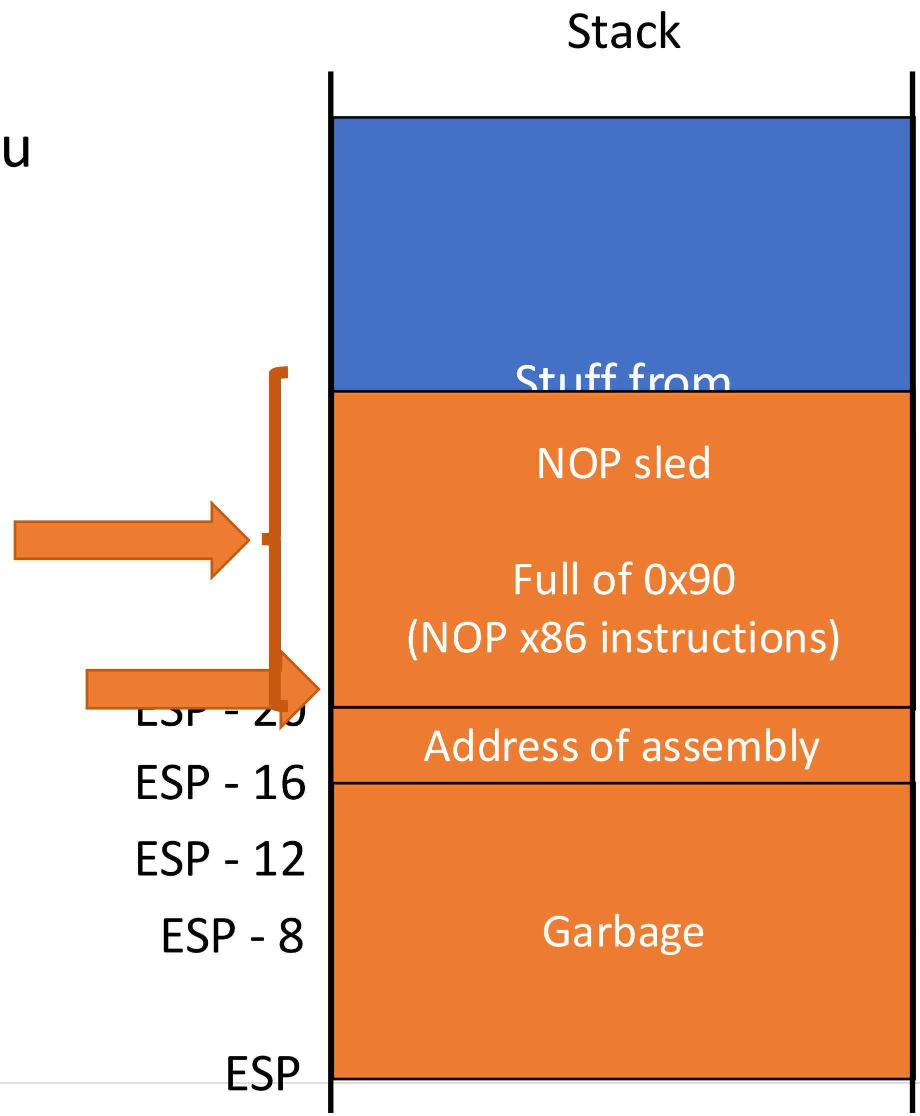
```
[cbw@finalfight game] ./guessinggame [16-bytes of
garbage][4-byte stack pointer][evil shellcode assembly]
Segmentation fault (core dumped)
```

This is not what we want :(

- Problem: how do you know the address of the shellcode on the stack?
  - To execute the shellcode, you have to return to its exact start address
  - This is a small target

NOP Sled

- To execute the shellcode, you have to return to its exact start address
- You can increase the size of the target using a NOP sled (a.k.a. slide, ramp)



./guessinggame ran the shellcode, turned into /bin/sh

```
[c[illegible]nalfight game] ./guessinggame [16 bytes of garbage][4 byte stack pointer][2048 bytes of 0x90][evil shellcode assembly]
$
```

- There is a lot more to writing a successful exploits
  - Depending on the type of flaw, compiler countermeasures, and OS countermeasures
  - If you like this stuff, take a security course

Types of Exploitable Flaws

- Stack overflow
- Heap overflow

```
char * buf = malloc(100);
strcpy(buf, argv[1]);
```

- Double free

```
free(buf);
free(buf);
```

- Format string

```
printf(argv[1]);
```

- Off-by-one

```
int vectors[100];
for (i = 0; i <= 100; i++)
    vector[i] = x;
```

- ... and many more

Triggering Exploitable Flaws

- Local vulnerabilities:
  - Command line arguments
  - Environment variables
  - Data read from a file
  - Date from shared memory or pipes
- Remote vulnerabilities
  - Data read from a socket
- Basically, any place where an attacker can give input to your process

Attacker can inject code into your machine via the Internet

Leveraging an Exploit

- After a successful exploit, what can the attacker do?
  - Anything the exploited process could do
  - The shellcode has full API access
- Typical shellcode payload is to open a shell
  - Remote exploit: open a shell and bind STDIN/STDOUT to a socket (remote shell)
- If process is uid=root or setuid=root, exploitation results in privilege escalation
- If the process is the kernel, the exploit also results in privilege escalation

- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP

Defending Against Stack Exploits

- Exploits leverage programmer bugs
  - Programmers are never going to write code that is 100% bug-free
- What can the system do to help prevent processes from being exploited?
- Mechanisms that prevent stack-based exploits
  - Stack canaries
  - Non-executable stack pages (NX-bit)

The Canary in the Coal Mine

- Miners used to take canaries down into mines
- The birds are very sensitive to poisonous gases
- If the bird dies, it means something is very wrong!
- The bird is an early warning system

Northeastern University

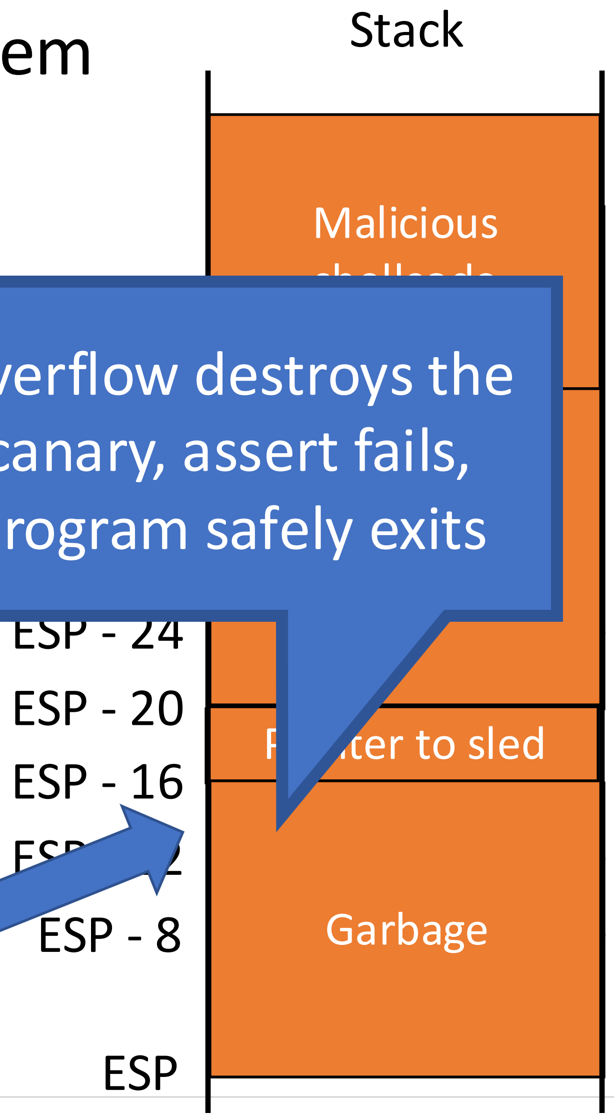
Stack Canaries

- A stack canary is an early warning system that alerts you to stack overflows

Automatically added by the compiler

```
int canary = secret_canary;
char buf[8];
for (x = 1; x < argc; ++x) {
  strcpy(buf, argv[x]);
  num = atoi(buf);
  check_for_secret(num);
}
...
assert(canary==secret_canary);
return 0;
```

Overflow destroys the canary, assert fails, program safely exits



Canary Implementation

- Canary code and data are inserted by the compiler
  - gcc supports canaries
  - Disable using the –fno-stack-protector argument
- Canary secret must be random
  - Otherwise the attacker could guess it
- Canary secret is stored on its own page at semi-random location in virtual memory
  - Makes it difficult to locate and read from memory

Canaries in Action

```
[cbw@finalfight game] ./guessinggame AAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: ./guessinggame terminated
Segmentation fault (core dumped)
```

- Note: canaries do not prevent the buffer overflow
- The canary prevents the overflow from being exploited

When Canaries Fail

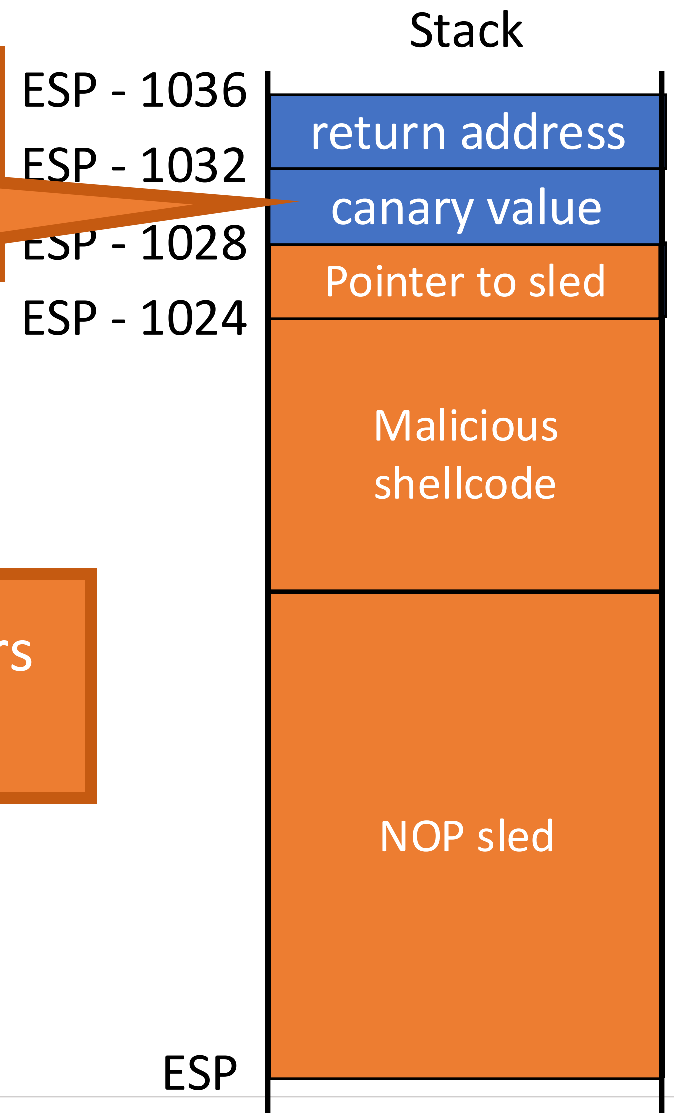
```
void my_func() { ... }

int canary = secret_canary;
void (*fptr)(void);
char buf[1024];
fptr = &my_func;
strcpy(buf, argv[1]);
fptr();
assert(canary==secret_canary);
return 0;
```

Function pointer

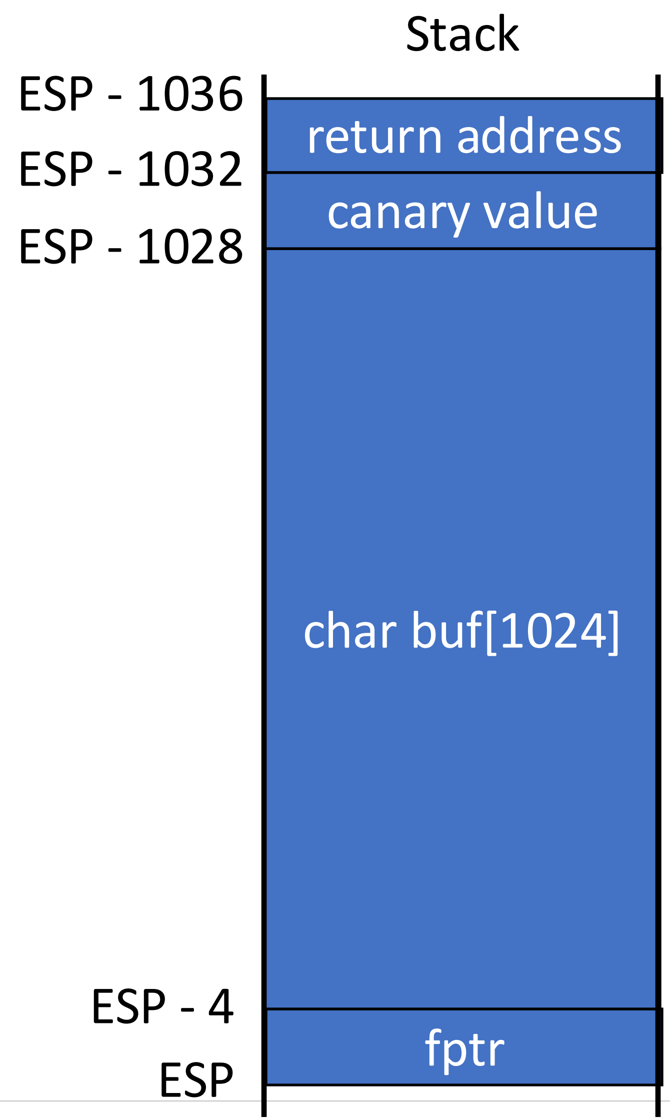
Calling fptr triggers the exploit

Canary is left intact



ProPolice Compiler

- Security oriented compiler technique
- Attempts to place arrays above other local variables on the stack
- Integrated into gcc



When ProPolice Fails

```
void my_func() { ... }

struct my_stuff {
  void (*fptr)(void);
  char buf[1024];
};

int canary = secret_canary;
struct my_stuff stuff;
stuff.fptr = &my_func;
strcpy(stuff.buf, argv[1]);
stuff.fptr();
assert(canary==secret_canary);
return 0;
```

- The C specification states that the fields of a struct cannot be reordered by the compiler

Non-Executable Stack

- Problem: compiler techniques cannot prevent all stack-based exploits
- Key insight: many exploits require placing code in the stack and executing it
  - Code doesn't typically go on stack pages
- Solution: make stack pages non-executable
  - Compiler marks stack segment as non-executable
  - Loader sets the corresponding page as non-executable

x86 Page Table Entry, Again

- On x86, page table entries (PTE) are 4 bytes

31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

- W bit determines writeable status
- … but there is no bit for executable/non-executable
- On x86-64, the most significant bit of each PTE (bit 63) determines if a page is executable
  – AMD calls it the NX bit: No-eXecute
  – Intel calls it the XD bit: eXecute Disable

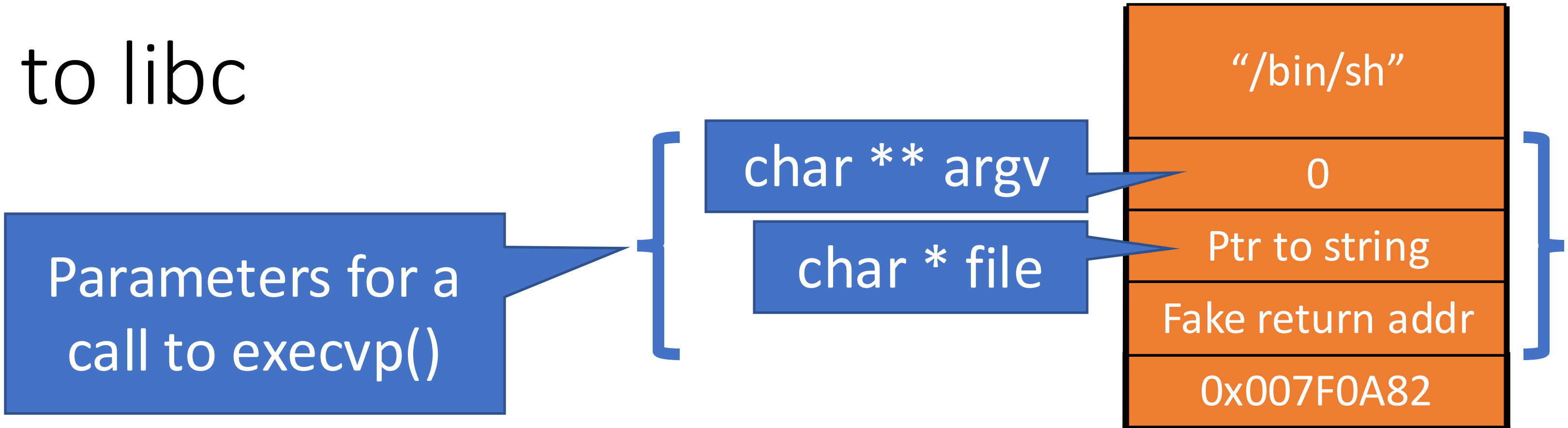
Northeastern University

When NX bits Fail

- NX prevents shellcode from being placed on the stack
  - NX must be enabled by the process
  - NX must be supported by the OS
- Can exploit writers get around NX?
  - Of course ;)
  - Return-to-libc
  - Return-oriented programming (ROP)

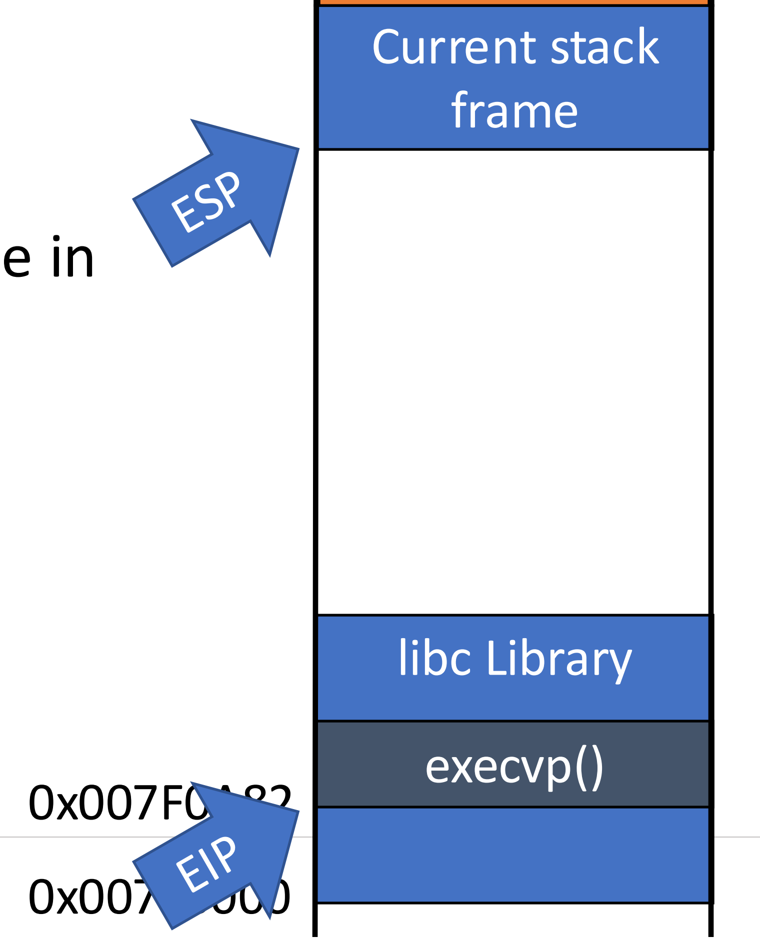
- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP

Return to libc



- Example exploits thus far have leveraged code injection
- Why not use code that is already available in the process?

execvp(char * file, char ** argv);



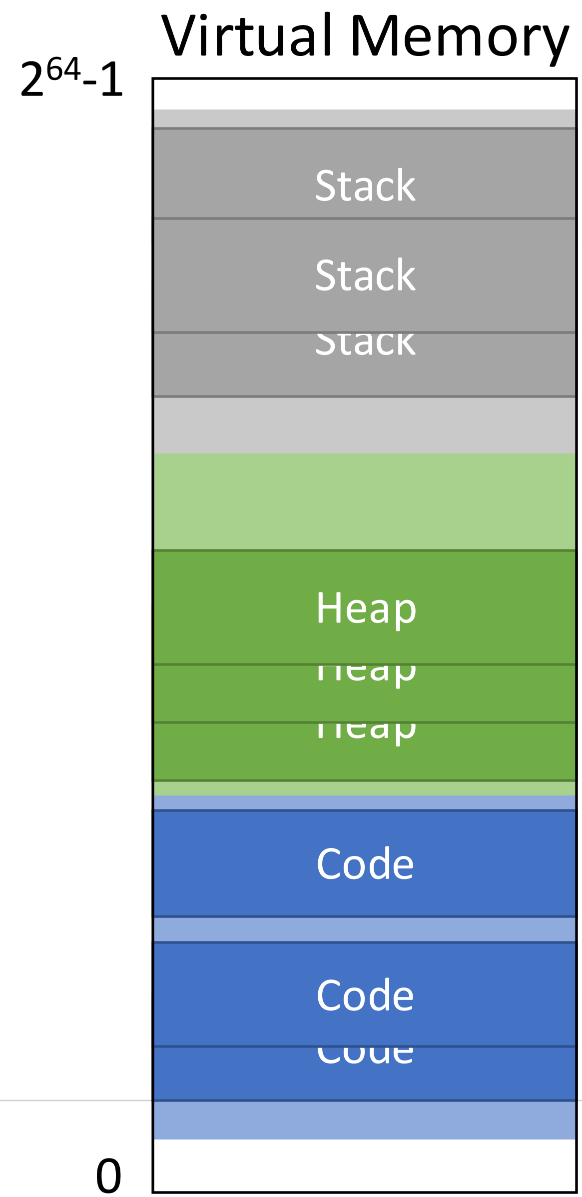
Stack Control = Program Control

- Return to libc works by crafting special stack frames and using existing library code
  - No need to inject code, just data onto the stack
- Return-oriented programming (ROP) is a generalization of return to libc
  - Why only jump to existing functions?
  - You can jump to code **anywhere** in the program
  - Gadgets are snippets of assembly that form a Turing complete language
  - Gadgets + control of the stack = arbitrary code execution power

- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP

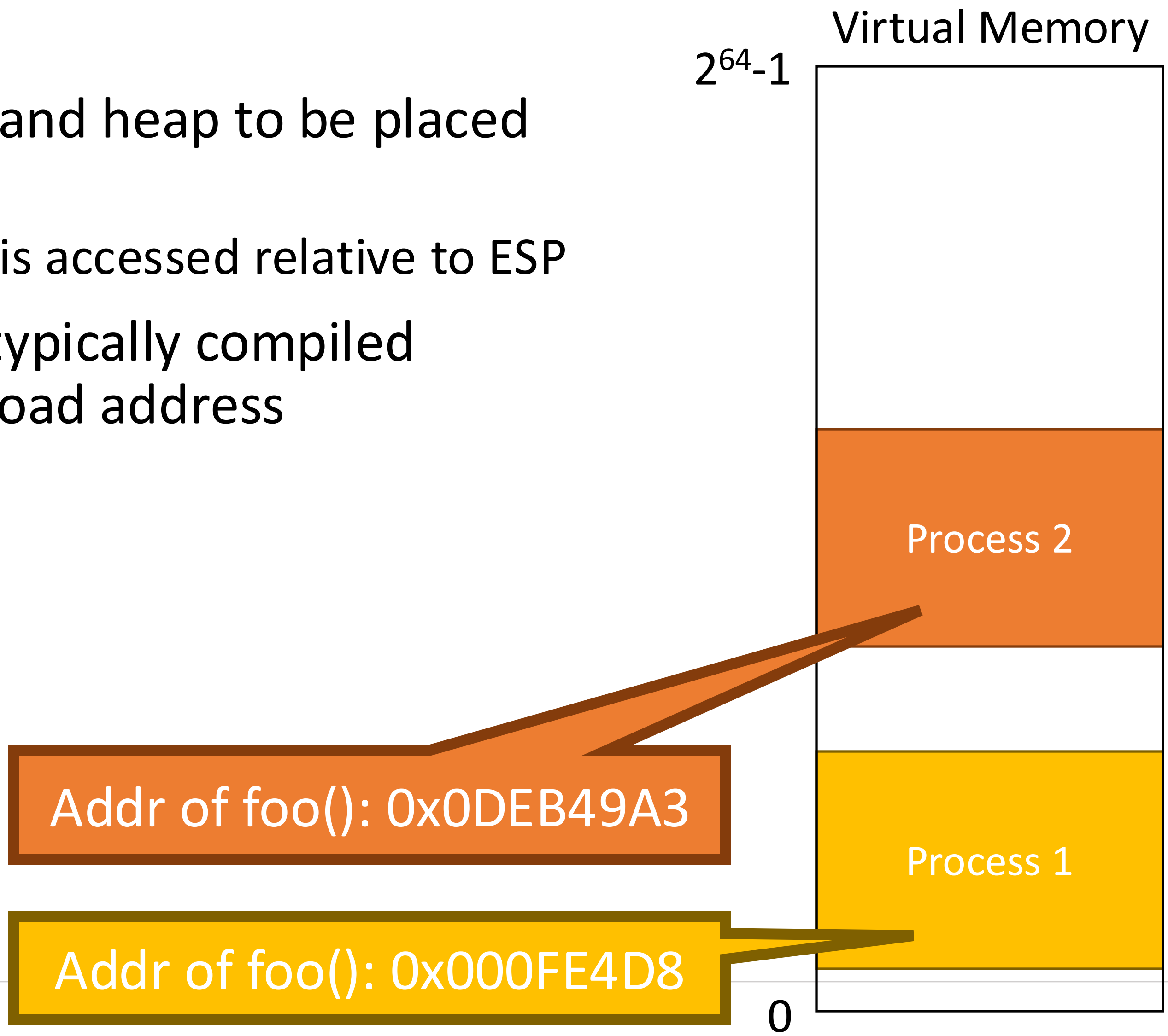
Defending Against Return to libc

- Return to libc and ROP work by repeatedly returning to known pieces of code
  - This assumes the attacker knows the addresses of this code in memory
- Key idea: place code and data at **random** places in memory
  - Address Space Layout Randomization (ASLR)
  - Supported by all modern OSes



Randomizing Code Placement

- It's okay for stack and heap to be placed randomly
  - Example: stack is accessed relative to ESP
- Problem: code is typically compiled assuming a fixed load address



Position Independent Code Example

- Modern compil
  (PIC)
  - Also called P

- e8 is the opcode for a **relative** function call
- Address is calculated as EIP + given value
- Example: 0x4004c7 + 0xffffffe8 = 0x4004af

Global data is accessed relative to EIP

```
int global_var = 20;

int func() { return 30; }

int main() {
  int x = func();
  global_var = 10;
  return 0;
}
```

```
4004bf:                          p
4004c0:   4  89 e5               ebp, esp
4004c3:   48 83 ec 10            b    esp, 0x10
4004c7:   e8 e8 ff ff ff         call  4004b4 <func>
4004cc:   89 45 fc               mov [ebp-0x4], eax
4004cf:   c7 05 3f 0b 20 00 10   mov
                                 [eip+0x200b3f], 0x10
4004d6:   00 00 00
4004d9:   b8 00 00 00 00         mov    eax, 0x0
4004de:   c9                     leave
4004df:   c3                     ret
```

Tradeoffs with PIC/PIE

- Pro
  - Enables the OS to place the code and data segments at a random place in memory (ASLR)
- Con
  - Code is slightly less efficient
  - Some addresses must be calculated
- In general, the security benefits of ASLR far outweigh the cost

When ASLR Fails

- ASLR is much less effective on 32-bit architectures
  - Less ability to move pages around randomly
  - May allow the attacker to brute-force the exploit
- Use a huge NOP sled
  - If the sled is enormous, even a random jump will hit it
- Use heap spraying
  - Technique that creates many, many, many copies of shellcode in memory
  - Attempts to fill all available heap memory
  - Jump to a random address is likely to hit a copy

Exploitation Prevention Wrap-up

- Modern OSes and compilers implement many strategies to prevent exploitation
  - More advanced techniques exist and are under development
- Exploitation strategies are also becoming more sophisticated
  - Just scratched the surface of attack strategies
- Bottom line: **don't write buggy code**
  - Compiler and OS techniques don't fix bugs, they just try to prevent exploitation
  - Even minor flaws can be exploited

Strategies for Writing Secure Code

- **Assume all external data is under the control of an attacker**
- Avoid unsafe library calls
  - strcpy(), memcpy(), gets(), etc.
  - Use bounded versions instead, i.e. strncpy()
- Use static analysis tools, e.g. Valgrind
- Use a fuzzer
  - Runs your program repeatedly with crafted inputs
  - Designed to trigger flaws
- Use security best-practices
  - Drop privileges, use chroot jails, etc.

- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP

Cybersecurity and Ethics

- Many laws govern cybersecurity
  - Designed to help prosecute criminals
  - Discourage destructive or fraudulent activities
- However, these laws are broad and often vague
  - Easy to violate these laws accidentally
  - Security professionals must be cautious and protect themselves
- Cybersecurity raises complex ethical questions
  - When and how to disclose vulnerabilities
  - How to handle leaked data
  - Line between observing and enabling crime
  - Balancing security vs. autonomy
- Ethical norms must be respected
  - Rights and expectations of individuals and companies
  - Community best-practices

Other Topics in Security

- Attacks we have not studied
- Secure Hardware Technologies (TPM, TXT)
- Distributed System Security and Resilience
- Cryptocurrencies and smart contracts
- Protocol Security (wireless, SDN)
- Privacy and regulations
- Post-quantum cryptography
- Program analysis, fuzzing, and software testing
- Formal verification
- Mobile and IoT security
- Machine Learning for Security
- Adversarial Machine Learning

Five Things to Remember

1) Be Morally Ambitious

- There are many tech jobs out there
- Do something meaningful
  - You'll have to work to find that thing

https://public-interest-tech.com/

2) Focus on the Problems

- You can't improve the world unless you are solving a problem
- Identifying the problem is the hard part
  - This is most of research
- Specific technologies are just means to an end, not an end themselves

3) Stay Optimistic

- A lot of uncertainty in the world right now
- Nonetheless, it is a time for hope and action
- Great progress only happens in times of great need

4) Time is Life

- How you spend your time *is* your life

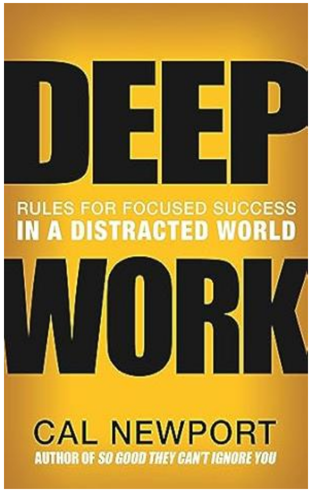
"The time that passes belongs to death."

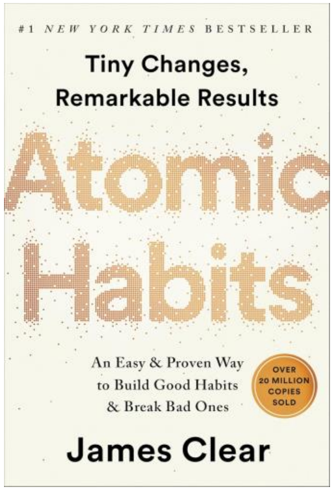
-- Seneca

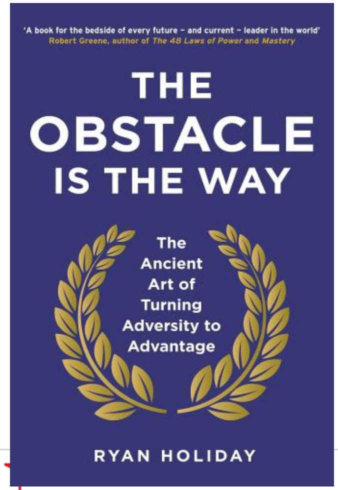
https://en.wikipedia.org/wiki/Seneca_the_Younger#/media/File:Duble_herma_of_Socrates_and_Seneca_Antikensammlung_Berlin_07.jpg

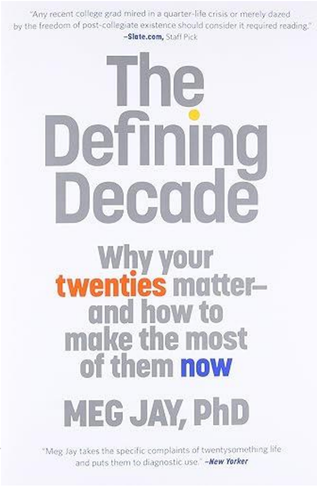
5) Read Books

Productivity









Broadening Horizons