EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Computer Security

Security is a big field, encompassing:

- Application security
- Network security
- Authentication
- Digital forensics
- Cryptography
- Privacy/Anonymity
- etc.

Operating Systems Security

- In this course we are only going to concern ourselves with security as it applies to operating systems
- This is some mix of *authentication*, *access control*, and *application security*
- First though, some definitions and ideas about how to think about security

Computer Security

Generally, we talk about computer security in three broad categories:

- ***C***onfidentiality – preventing others from finding out information we don't want them to have
- ***I***ntegrity – preventing others from modifying our data without permission
- ***A***vailability – preventing others from denying us access to some service (denial of service)

Threat Modeling

- It usually doesn't make sense to talk about a system being "secure" or "unsecure"
- Instead, we need to be more precise:
  - What are we trying to protect?
  - Who do we need to protect against? What are their capabilities?

A Practical Threat Model

Threat:

- Ex-girlfriend/boyfriend breaking into your email account and publicly releasing your correspondence with the My Little Pony fan club

Solution:

- Strong passwords

A Practical Threat Model

Threat:

- Organized criminals breaking into your email account and sending spam using your identity

Solution:

- Strong passwords + common sense (e.g., don't click on unsolicited herbal Viagra ads that result in keyloggers and sorrow)

A Practical Threat Model

Threat:

- The Mossad doing Mossad things with your email account

Solution:

- Magical amulets(cross, voodoo doll)?
- Fake your own death, move into a submarine?
- YOU’RE STILL GONNA BE MOSSAD’ED UPON

Threat Modeling

Roughly, we can divide this into three steps:

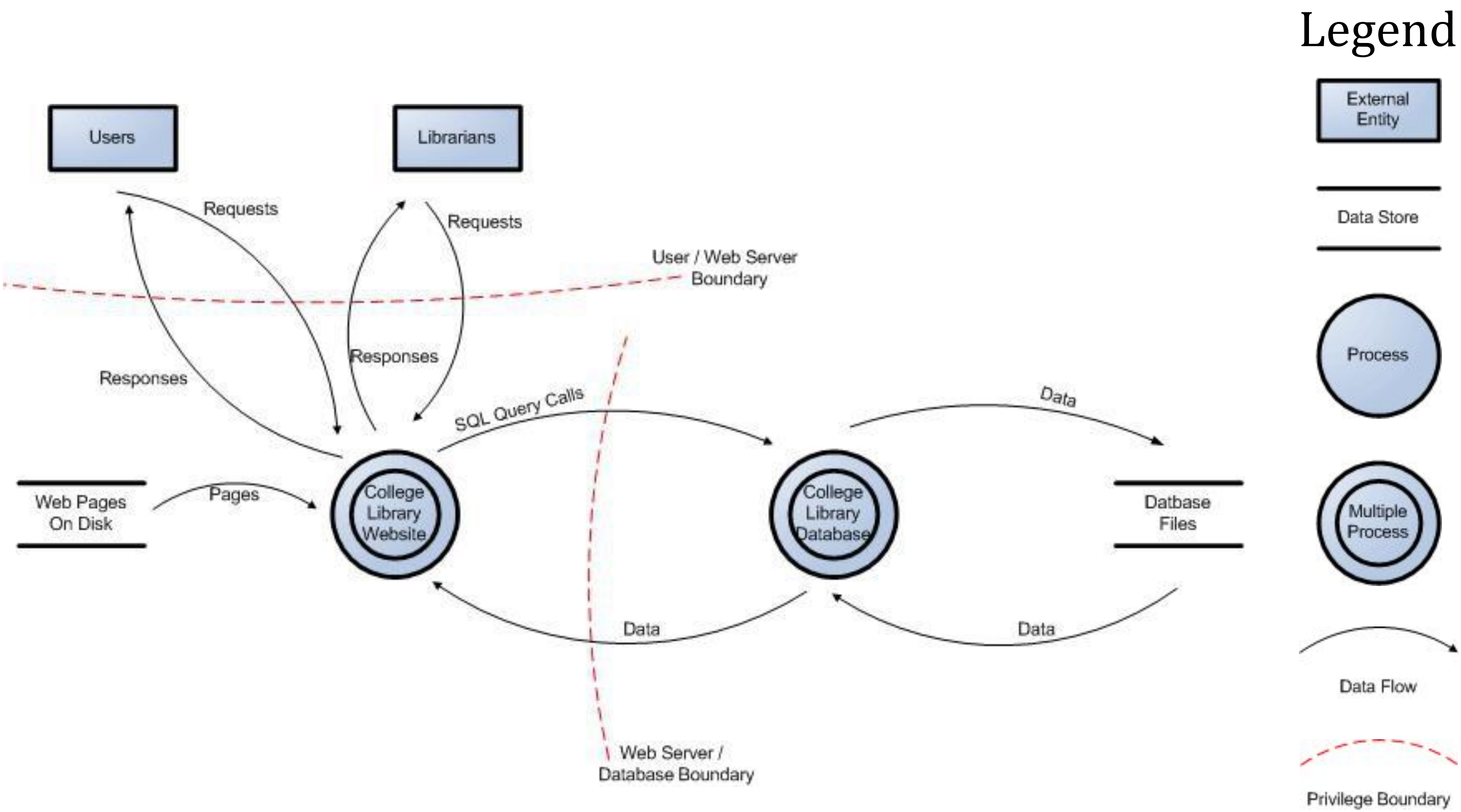
- System understanding
- Threat categorization
- Countermeasures and mitigation

System Understanding

To properly protect a system, you need to understand it:

- Identify assets that need protecting
- Look at ways the system can be used and assets can be accessed
- Figure out what right should be granted to what assets and classes of users
- Identify *privilege boundaries* – places where a program or user changes their privilege level

Example System Diagram



Source: OWASP

Threat Categorization

- Look at the system from an attacker's point of view
- What goals might an attacker have?
- How could they achieve these goals?
- May help to use a threat categorization such as **STRIDE**: **S**poofing, **T**ampering, **R**epudiation, **I**nformation Disclosure, **D**enial of Service, **E**levation of Privilege

Countermeasures and Mitigation

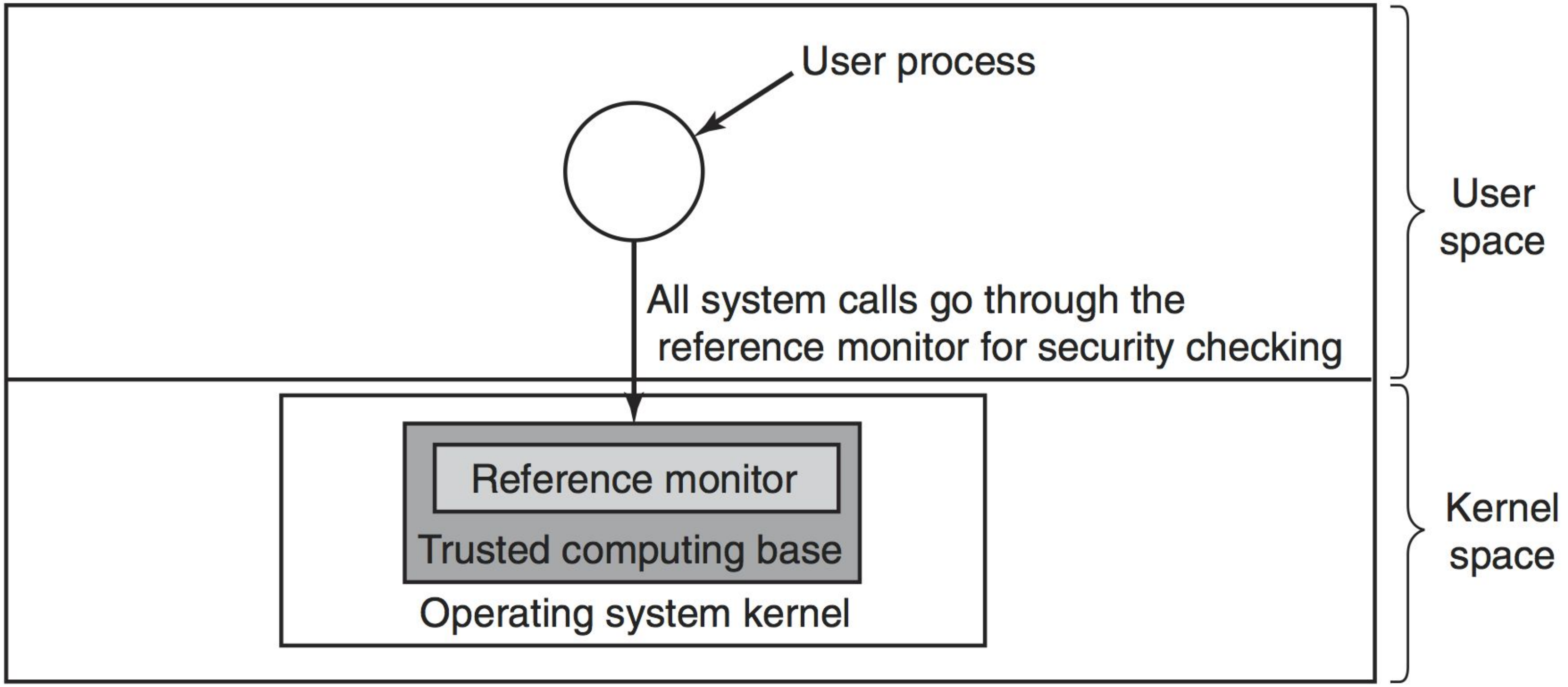
For each of the threats to some asset, come up with a plan for mitigating or nullifying the threat, for example:

- Attacker might guess someone's password
  - -> enforce password complexity requirements
- Attacker might snoop on network traffic
  - -> encrypt data that is sent on the network

Trusted Computing Base(TCB)

- One strategy for building secure operating systems is to organize them into *trusted* and *untrusted* components
- The goal is that if the *trusted* components performs according to its specification, then some specific set of guarantees about security must hold
- A *reference monitor* checks all accesses between trusted and untrusted components

TCB + Reference Monitor



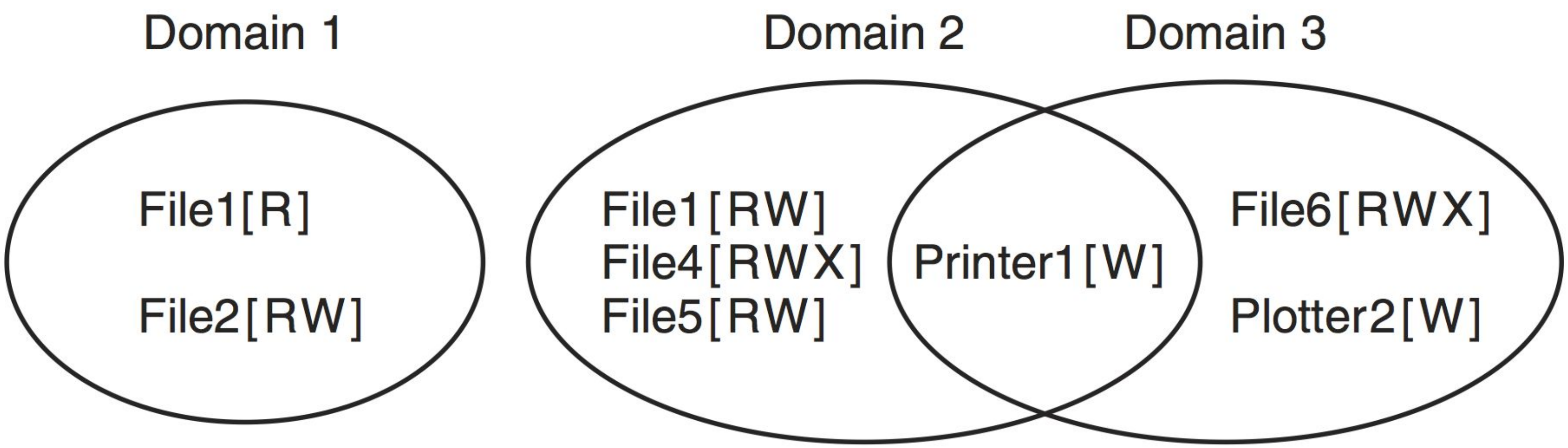
Aside: Bugs & Program Size

- One rule of thumb is that as program size increases, the number of bugs increases as well
- It’s harder to reason about code the more of it there is and the more complex it is.
- Therefore, if we want to be confident in our trusted computing base, we should work to *minimize* the amount of code in it
- How big is the TCB for widely-used operating systems?(HUGE!)

Protection Domain

- A protection domain is a set of (object, rights) pairs
- A right, in this context, means an operation that can be performed on an object
- So, for example, a protection domain might correspond to a user and the set of objects they have access to
- Or, a group of users that all share the same rights

Protection Domains



Principle of Least Privilege

- One principle for designing secure systems is the *principle of least privilege*.
- The core idea is that the set of objects and rights for a given protection domain should be as small as possible
- This seems obvious, but is often violated in practice

Processes and Protection Domains

- At any given time, a process operates in one protection domain
  - i.e., there is some specific set of objects that the process has permissions to perform some actions on
- Processes can also typically switch between protection domains as they run
  - The rules for when and how they do this vary widely between different operating systems

The UNIX Protection Model

- The protection domain of a process in UNIX is defined by its user id (uid) and group id (gid)
- The objects are files (including special files like hardware devices)

The UNIX Protection Model

- Each process is further divided into two halves:
  - kernel mode and user mode
- The kernel half can access a different set of objects from the user half, so the change from user to kernel is a domain switch
- Executable files can also have SETUID and SETGID bits, meaning that when they are executed they will run under different permissions

User Management

Code running in user mode is always linked to a certain identity
- Security checks and access control decisions are based on user identity

Unix is user-centric
- No roles
- Users Identified by user id (uid) & group id (gid)
- Authenticated by password (stored encrypted)

User root
- Superuser, system administrator
- Special privileges (access resources, configure the OS)
- Cannot decrypt user passwords

Process Management (PCB)

Process Attributes

- Process id (PID)
  - Uniquely identifies a process
  - PIDs are reused
- User id (UID)
  - ID of owner of process
- Effective user id (EUID)
  - ID used for permission checks / access control
- Saved user id (SUID)
  - To temporarily drop and restore privileges
- Lots of management information
  - Scheduling
  - Memory management
  - Resource management

User Authentication

- **How does a process get a user ID?**
  - Authentication via `login`
- **Passwords**
  - User passwords used as key for `crypt()` function
  - 12 bit public salt (prevent pw collisions)
  - Repeatedly apply encryption
- **Password cracking**
  - Dictionary attacks
  - Crack, JohnTheRipper

User Authentication

File /etc/passwd

- Maps user names to user ids (many applications legitimately need this)
- No legitimate need for encrypted passwords

File /etc/shadow

- Contains encrypted passwords
- Account information (last change, expiration)
- Readable only by superuser and privileged programs
- Different hash algorithms supported
  - DES
  - MD5
  - SHA-{256,512}

Unix Groups

Users belong to one or more groups

- Primary group (stored in `/etc/passwd`)
- Additional groups (stored in `/etc/group`)
- Possibility to set group password
- Become group member with `newgrp`

File /etc/group

```
groupname : password : group id : additional users
root:x:0:root
bin:x:1:root,bin,daemon
users:x:1000:pizzaman
```

Special group wheel

- Group for users that can call `su`

File System

Access Control

- Permission bits
- chmod, chown, chgrp, umask
- File listing

```
    -         rwx    rwx        rwx
(file type) (user) (group) (other/world)
```

Type	**r**	**w**	**x**	**s**	**t**
File	Read access	Write access	Execute	suid / sgid inherit id	sticky bit
Directory	List files	Add and remove files	Stat / execute files, chdir	New files have dir-gid	Files only deletable by owner

SUID Programs

- **Each process has *real* and *effective* user / group id**
  - Usually identical
  - Real id
    - Determined by current user
    - `login, su`
  - Effective ids
    - Determine access rights of a process
    - System calls (e.g., `setuid()`)
  - `suid/sgid` bits
    - Start process with effective ID different from real ID
    - Attractive targets for attacker
- **No SUID shell scripts anymore**

Extended Attributes

```
# lsattr /etc/passwd /etc/ssl
-------------e-- /etc/passwd
----------I--e-- /etc/ssl/certs
```

- Require support from file system
- Management via `lsattr`, `chattr`
  - Undeletable (u)
  - Append only (a)
  - Immutability (i)
  - Secure deletion (s)
  - Compression (c)
  - Hashed trees indexing for directories (I)

POSIX ACLs

Extend UNIX permission model to support fine-grained access control

```
$ sudo setfacl –m u:pizzaman:r secret
$ getfacl secret
  # file: secret
  # owner: root
  # group: root
  user::rw-
  user:pizzaman:r--
  group::---
  mask::r--
  other::---
```

Software Security

- One final aspect to OS security is *software security* – identifying and preventing programming flaws that could let an attacker take control

- Many of these are caused by the fact that languages currently in use are not *memory safe* – it is possible to write to data outside of program variables

- A big offender here is C/C++

Operating System Defenses

- OSes can be designed to make problems in user-level applications harder to exploit
- These generally don't make attacks on software impossible, but they can make them much more difficult
- This is something of an arms race – defenders come up with new mechanisms, attackers find ways around them

Classic Problem

```
#include<stdio.h>
#include<string.h>

int main(int argc, char ** argv) {
    char buf[256];
    strcpy(buf, argv[1]);
    printf("%s\n", buf);
    return 0;
}
```

What can possibly go wrong?

Stack Buffer Overflow

- Recall the standard stack frame:
  - Local variables
  - Return address
- If we try to store too much data in a local stack variable (e.g., a character array) we will overwrite the return address
- When the ret instruction is executed, control will jump to somewhere controlled by user input

Stack Canaries / Cookies

- Idea: put a special value in between the local variables and the return address so that overflowing a local buffer can be detected
- Upon entering the function, set a randomly-generated *cookie* value on the stack and store a backup copy somewhere
- Before executing a ret, check the stack cookie value against the backup and raise an error if it fails

PCB In Linux – task_struct

```
struct task_struct {
    volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
    /* task state */
    int exit_state;
    pid_t pid;
    /* Canary value for the -fstack-protector gcc feature */
    unsigned long stack_canary;
    struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
    struct timespec start_time;        /* monotonic time */
    char comm[TASK_COMM_LEN]; /* executable name excluding path */
    /* CPU-specific state of this task */
    struct thread_struct thread;
    /* signal handlers */
    struct signal_struct *signal;
    sigset_t blocked, real_blocked;
    ... 300 lines ...
}
```

What's the Problem?

Attack can ***inject*** new code and then corrupt the program to ***execute*** this code ….

What's the solution?

– Make it so that:
  1. The attacker cannot inject new code, or
     (Hard to do for programs written in C/C++)
  2. The new code cannot be executed!

DEP/NX/W⊕X

- Another defense is to try and make sure that even if an attacker can overflow a buffer and change the return address, they cannot execute the injected code
- In the previous example, attacker code was placed into a stack buffer
- So, simple solution: don’t allow data to be executable!
- Generally this requires hardware support
  - NX bit in x86 page protections

63 62 61 60 59 58 57 56 55 54 53 52 51	M[1] M-1	32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0													
Reserved[2]	Address of PML4 table	Ignored	PCD	PWT	Ign.											CR3
XD[3]	Ignored	Rsvd.	Address of page-directory-pointer table	Ign.	Rsvd	Ign	A	PCD	PWT	U/S	R/W	1				PML4E: present
Ignored												0				PML4E: not present
XD	Ignored	Rsvd.	Address of 1GB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored	Rsvd.	Address of page directory	Ign.	0	Ign	A	PCD	PWT	U/S	R/W	1				PDPTE: page directory
Ignored												0				PDTPE: not present
XD	Ignored	Rsvd.	Address of 2MB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored	Rsvd.	Address of page table	Ign.	0	Ign	A	PCD	PWT	U/S	R/W	1				PDE: page table
Ignored												0				PDE: not present
XD	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1			PTE: 4KB page
Ignored												0				PTE: not present

Current State of Affairs

- Great, we just made sure that the attacker can no longer execute any code that he injected into the program!
- Are we done?
- If the attacker cannot run injected code what else can he do?

Reuse existing code (which must be executable as it is code) → Code Reuse Attacks

Code Reuse Attacks

- Despite DEP, attacks can still run code!
- Instead of trying to execute their own code, attackers can change the return address to point to existing code in memory
- By setting up values on the stack, they can bounce around executing tiny snippets of code ending in ret
- Thus, by chaining these together, arbitrary computation can be performed – without executing anything marked as data

Code Reuse Attacks

- Problem: Attacker *knows where* existing code is in memory and manages to run that code
- What can we do to thwart such attacks?
- Make it so the attacker does no longer know where code is in memory
- How would we do that?
- We make sure the code is loaded at random addresses every time a new process starts (fork, exec, both?)

Address Space Layout Randomization

- Exploiting a buffer overflow typically requires knowing about the precise layout of memory
- For example, we may need to know where the stack is located, or where a certain library has been loaded
- Thus, to make attackers' lives more difficult, we can place the program, stack, and libraries at random locations each time the program starts

ASLR and Address Space

- We can estimate the amount of randomness provided by ASLR by counting the number of possible locations to load things
- On a 32-bit system, address space is not very large, so there are not many ways to randomize
- In 2004, researchers showed that on 32-bit systems, ASLR only has about 16 bits of entropy (65,536 possible values)
- The correct location can be guessed in a matter of seconds by just trying each possibility

Side Channel Attacks

- Sophisticated techniques that measure performance/timing differences to determine the contents of caches and therefore memory.
- Fix required hardware updates and/or firmware/kernel modifications that usually result in performance degradation.
- Meltdown - sharing page table between user and kernel results in security holes
- Spectre - hardware prefetch and speculation results in security holes