CSE 127 Midterm Review

Midterm Logistics

- **Date :** 2/12, during class hours, Center Hall 109
- **Question format :** Multiple choice, short answer, PA questions
- Cheat sheets (one letter size page, double sided is okay, printing is okay)
- **Scope:** Things talked about in lecture, and what you did in the PAs

Topics

Threat Modelling and Security Properties

Control Flow Vulnerabilities :

- Different types of buffer overflow attacks
- Mitigation strategies
- techniques for evading mitigations
- Relationship between each other

Memory Safety:

- Return Oriented Programming (ROP)
-
- Control Flow Integrity (CFI)

System Security :

- Principles of secure system design
- Isolation (memory isolation, resource isolation in Unix, user/kernel isolation)
- VMs

Web Security :

- how the web works (Http, DOMs and JS)
- Attacker model, Security model
- Same-Origin Policy (SOP)
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- SQL Injection (SQLI)

Threat Modelling

- *Asset* we are trying to protect, and from which *Attacker ? - WHAT and WHO*
- Security Boundary? Attack Surface?
- The threat model defines the problem to be solved and problem scope

Assets

Example assets we are trying to protect?

- Password (hashes): Secret code for authentication.
- Emails: System for sending and receiving messages electronically.
- Browsing history: Pages visited, useful for web marketing and forensics.

Security Properties

What properties are we trying to enforce? (CIA triad)

- Confidentiality: Prevention of unauthorized access to information
- Integrity: Prevention of unauthorized changes
- Authenticity: Identification and assurance of origin
- Availability: Prevention of unauthorized *denial of service* to others
- Privacy: Protect sensitive information, such as personally identifiable information, etc.

Buffer Overflows

- What is a buffer overflow?
- What assumptions do buffer overflows violate?
- Where do buffer overflows typically occur and why?
- What is the problem with gets() and strcpy() ?

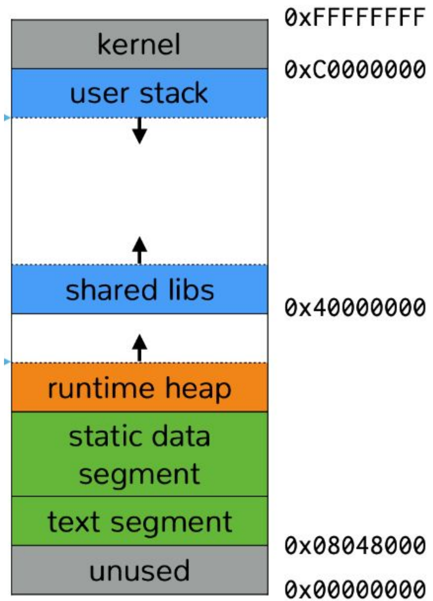
Buffer Overflows

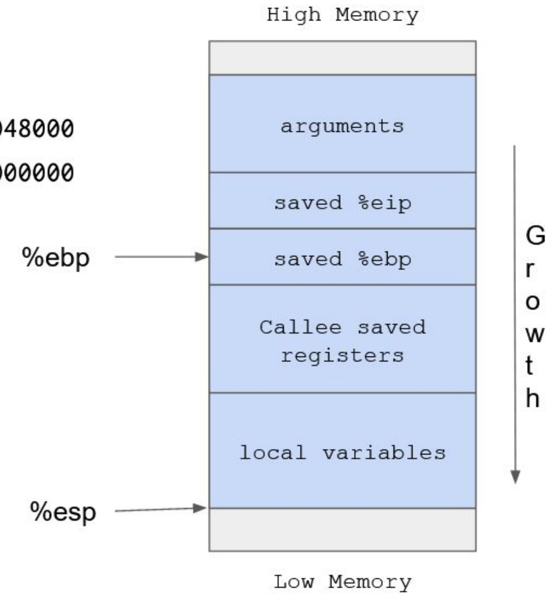
What are different ways to exploit a buffer overflow?

- Format String vulnerabilities
- Integer overflows
- Pointers

Memory layout and the Stack

- Stack
  - Local variables, function calls
- Heap
  - malloc, new, etc.
- Stack Frames
  - Each frame stores local vars and arguments to called functions
- Stack Pointer (%esp)
  - Points to the top of the stack
  - Grows down (High to low addrs)
- Frame Pointer (%ebp)
  - Points to the base of the caller's stack frame



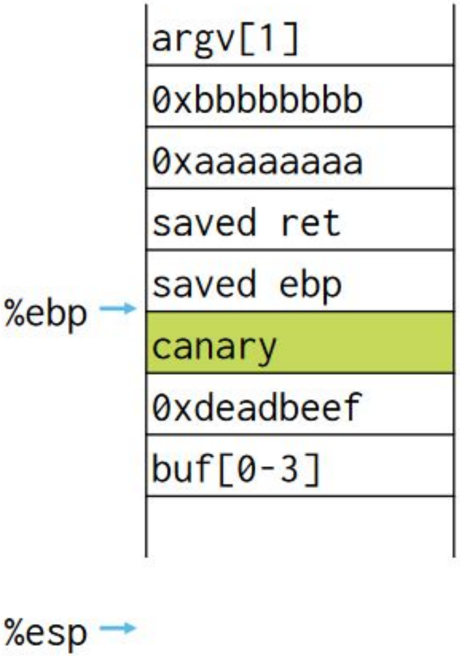


Mitigations: Stack Canaries

Detect overwriting of the return address

– Place a special value (aka canary or cookie) between local variables and the saved frame pointer

– Check that value before popping saved frame pointer and return address from the stack



Bypass:

- Learning the Canary
- Pointer subterfuge

Mitigations: DEP (Data Execution Prevention)

Make all pages either writable or executable, but not both

- Stack and heap are writeable, but not executable
- Code is executable, but not writeable
- Also known as W^X (Write XOR eXecute)
- prevent shell code from being executed in stack and heap

Bypasses:

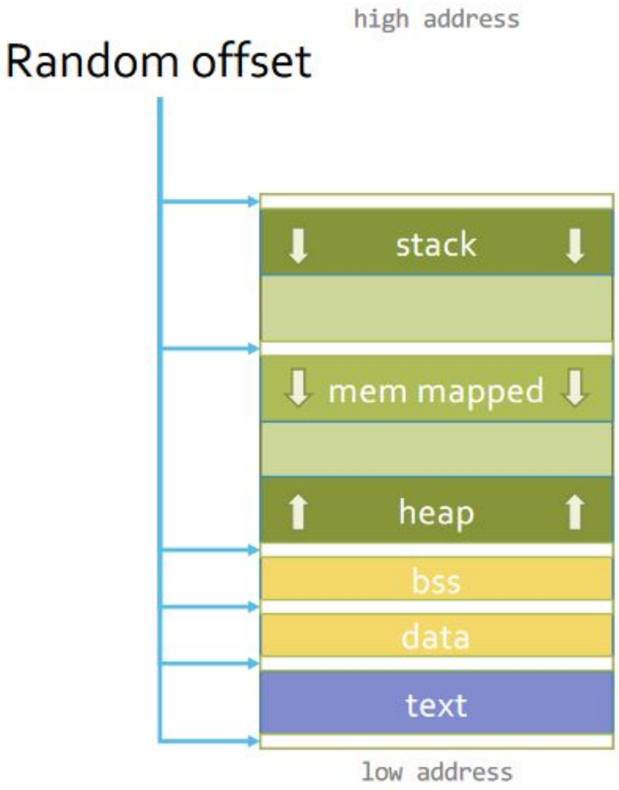
- Transfer control flow to an existing function (return-to-libc) (target 5)
- Return Oriented Programming (target 7)

Mitigations: ASLR (Address Space Layout Randomization)

Add random offsets to sections of process memory.

Bypasses:

- Guessing, Longer NOP sled (target 6)
- Heap Spraying



Evading Mitigations: Return-to-Libc

Motivation: Bypass DEP. Can't execute code we inject, so need to reuse existing code.

Idea: Overwrite the return address to point to start of system()

- Place address of "/bin/sh" on the stack so that system() uses it as the argument.
- Target 5

Evading Mitigations: Return Oriented Programming

- Why do we need return oriented programming? What does it help us do?
    - Perform exploits in the face of W^X (DEP) when cannot find just the right function
- Make complex shellcode out of existing application code
    - Call these gadgets
    - Where can you find the gadgets?
        - From executable pages in memory (app code, libc, other libraries)
        - Use attack tools
    - Where can you “stitch” these gadgets together?
        - Stack
- How can we defend ROP?
    - Control Flow Integrity
    - Type-safe/memory-safe languages

Mitigations: CFI (Control Flow Integrity)

Idea: Protecting indirect transfer of control flow instructions. Go after root of problem.

Direct control flow transfer:

- Advancing to next sequential instruction
- Jumping to (or calling a function at) an address hard-coded in the instruction
- Generally not a problem. In code where attackers cannot control

Indirect control flow transfer

- Jumping to (or calling a function at) an address in register or memory
- Forward path: indirect calls and branches (e.g., a function you are calling)
- Reverse path: return addresses on the stack (returning from a called function)

Restrict program control flow to the control flow graph (how it was written)

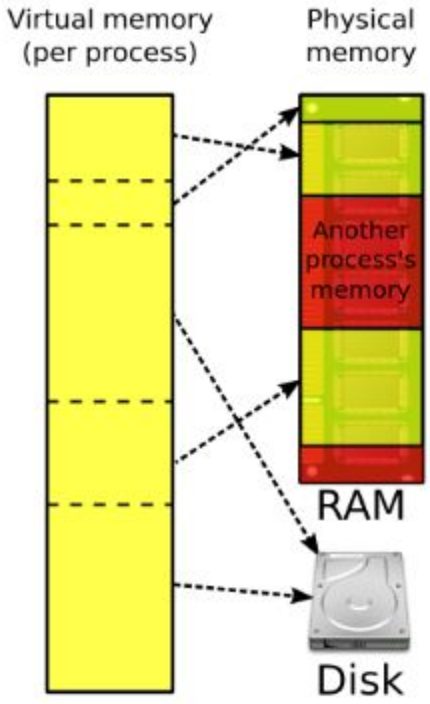
Put label at call site and target. Before jump, validate if target label matches jump site.

Principles of secure system design

- Least Privilege
  - Only provide as much privilege to a program as is needed to do its job
- Privilege separation
  - Multi-user operating system
- Complete mediation
  - Check every access that crosses a trust boundary against security policy
- Defence-in-depth
  - Use more than one security mechanism
- Keep-it-simple

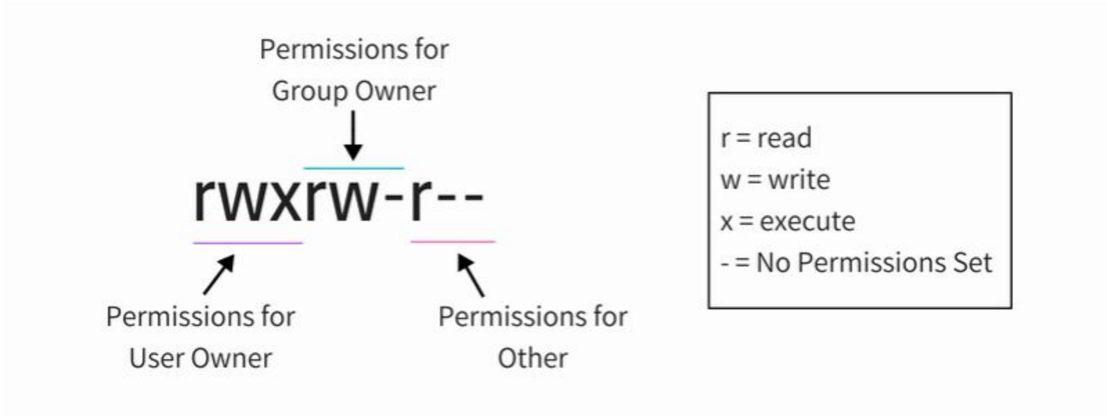
Memory Isolation

- Process should not be able to access another process’s memory
- Each process gets its own virtual address space, managed by the operating system
- Memory addresses used by processes are virtual addresses (VAs) not physical addresses (PAs)
- Address translation and Page tables

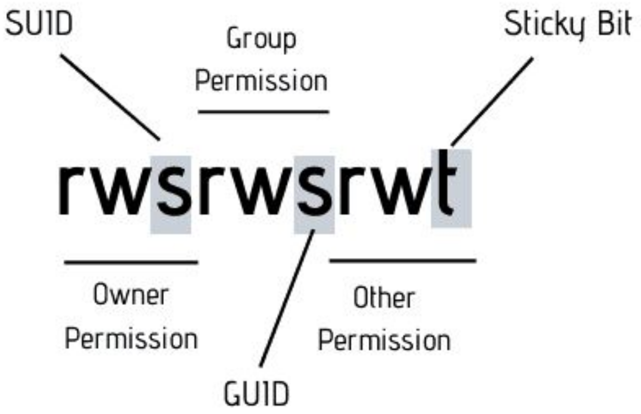


Process Isolation in Unix

- Process should only be able to access certain resources
- Permissions to access files are granted based on user IDs
- Access Operations on file: Read, Write, eXecute
- Each file has an access control list (ACL)
- Role based: user group other



Process Isolation in Unix

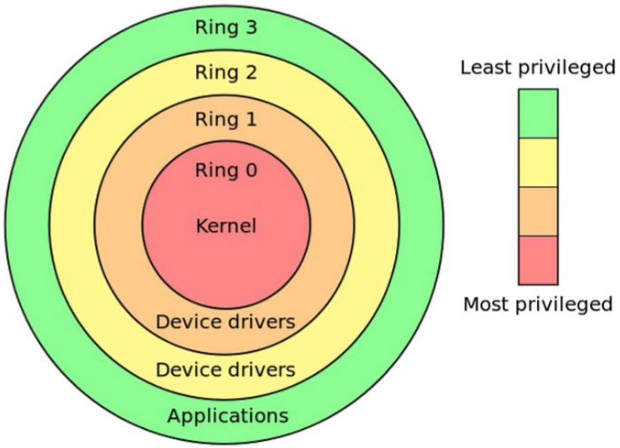


- RUID: Determines who started the process
- EUID: Determines the permissions for process
- Setuid bit
  - If setuid bit set: use UID of file owner as EUID
- Setgid bit
  - Same thing but for group

```
nadiah@login:$ ls -l
total 32
-rwxrw-r--  1 nadiah  professor 18660 Jan 14 00:34 foo.py
drwxrwxr-x  2 nadiah  professor  4096 Jan 13 08:42 pa
-rwsrwxr-x  3    leo         ta 12345 Jan 14 10:23 hello.py
```

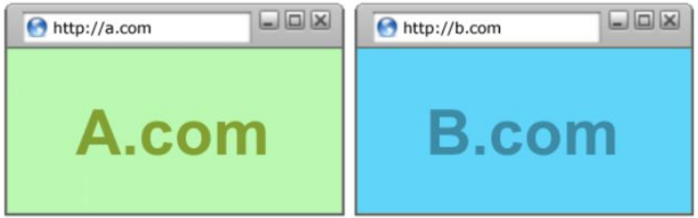
Kernel/User Isolation

- Kernel is isolated from user processes
  - Processor privilege levels
  - page table
- Interface between userspace and kernel: system calls
  - To damage a system, must make system calls
- Kernel Mapping
  - kernel's virtual memory space is mapped into every process, but made inaccessible when in user mode

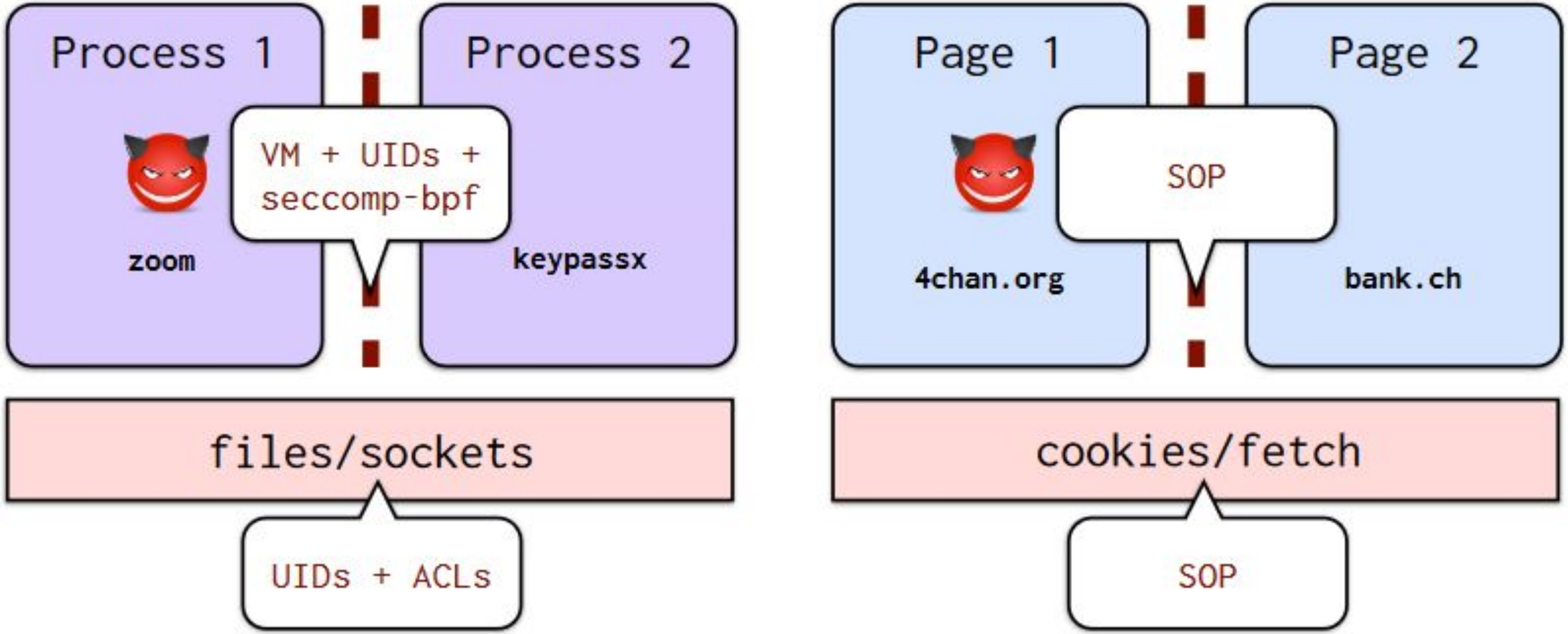


Web Security

- Browser
  - Load and execute content
  - Basic/Nested execution model
    - Frame and iFrame
  - Document Object Model (DOM)
  - DOM and JS
  - Same Origin Policy (SOP)
  - Cookies

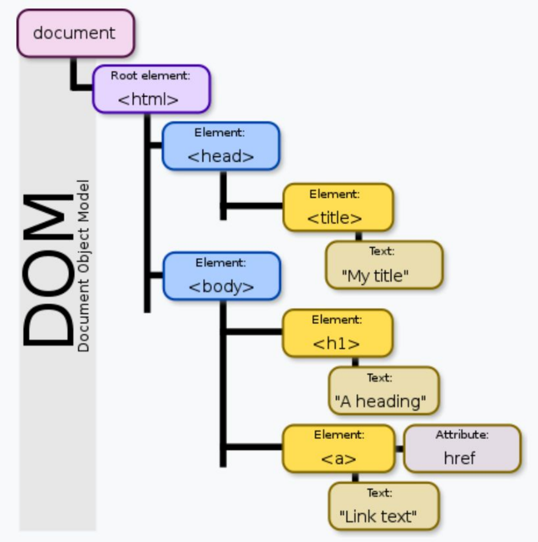


Security model



Web Security

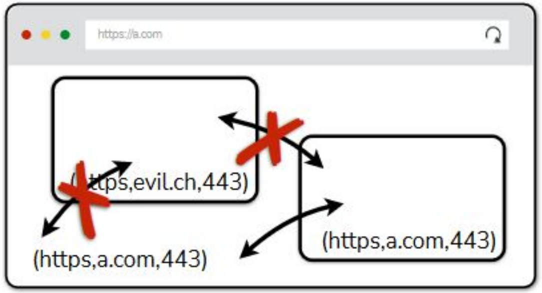
- Document object model (DOM)
  - treats HTML as a tree structure wherein each node is an object representing a part of the document.
  - Javascript can read and modify page by interacting with DOM



Same origin policy (SOP)

- goal: isolate content of different origins
- There is no one SOP. We focus on:
  - the DOM. Origin is a (scheme, domain, port)
  - Cookies. domain/path + Secure/SameSite

- Frame can only access data with the same origin
  - DOM tree, local storage, cookies, etc.



Web Attacks and Defenses

- Server-Side Injection
  - SQL Injection
    - SQL basics
    - Mitigations: Prepared statement
- Client-Side Injection
  - Cross Site Scripting (XSS): Injecting malicious scripts into benign and trusted website
  - Prevention: Content Security Policy
- Cross Site Request Forgery (CSRF)
  - Bad website forces the user's browser to send a request to a good website
  - Cookies
  - Mitigations: Tokens, Referer, SameSite
- Understand how the attack works

Good luck!