CSE 127 Midterm Review

Any questions on PA2?

Midterm Logistics

- Time : Oct 25 -> 2-3:20 CENTR 119 (Lecture time and location)
- Question format : True/False, Multiple choice (select all that apply)
- Cheat sheets (one page, both sides, readable with the unaided eye)
- Topics cover everything up to side channels but not including crypto
- Key idea in the readings

Topics

Threat Modelling and Security Properties

control flow vulnerabilities :

- Stack Overflow
- Heap Overflow
- printf
- Pointer subterfuge
- Integer overflow
- memory allocation exploits (e.g., UAF, DF)

mitigations:

- Stack Canaries
- DEP
- ALSR
- CFI

System Security :

- basic memory isolation (processes, kernel/user) and implementation (to the degree covered in class)

Side-channel :

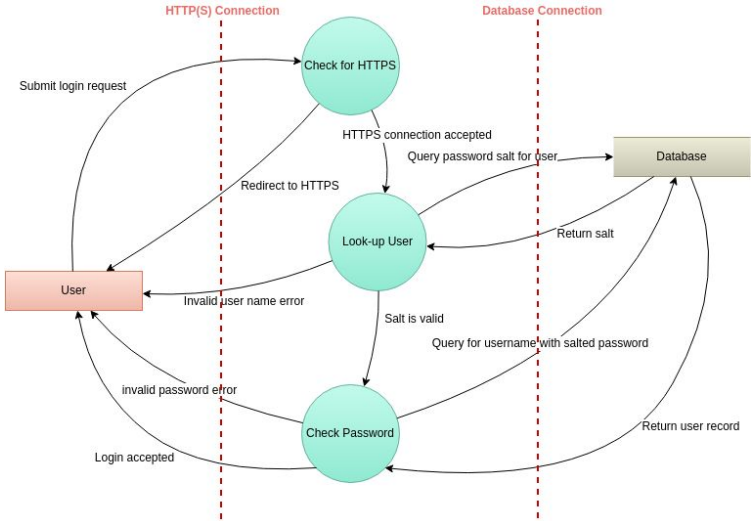
- Timing side channel, cache side channel.
- spectre/meltdown ideas

techniques for evading mitigations:

- Heap spray
- return-to-libc
- ROP
- etc.

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.
- Your threat model is <u>your</u> problem scope



Example Threat Model diagram

Security Properties

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() ?

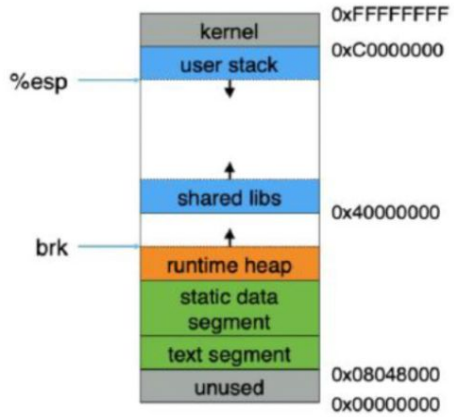
Buffer overflows

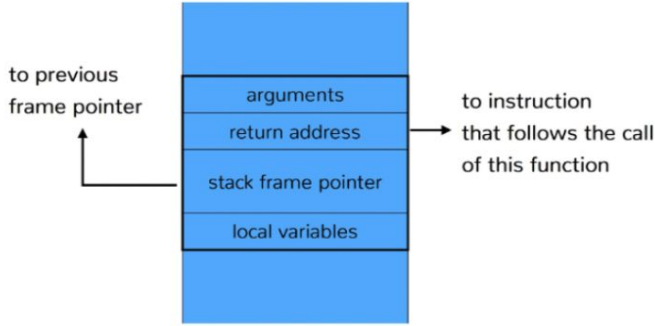
What are different ways to exploit a buffer overflow?

- Format String vulnerabilities
- Heap vulnerabilities
- Integers
- Pointers

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



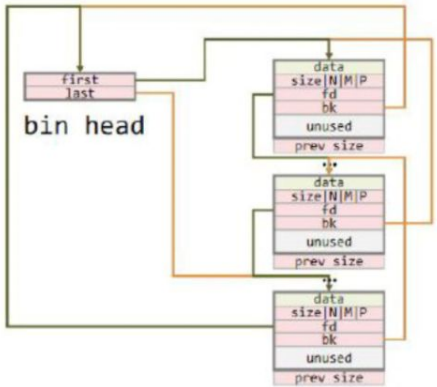


Heap Vulnerabilities

- Dynamically allocated memory in program
- Programmer is responsible for many of the details
  - Variable liveliness and validity
- Heap are kept in doubly-linked lists (bins)
- What happens to freed memory in the heap?
  - Double free and use after free

- Unlink operation to remove a chunk from the free list:

```
#define unlink(P, BK, FD)
{
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

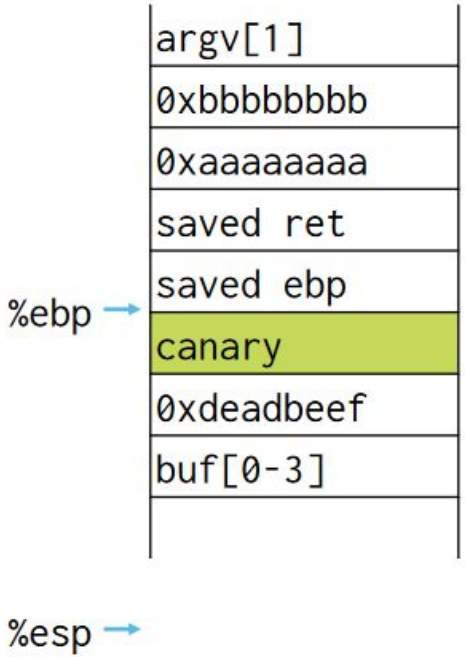


Mitigations: Stack Canaries

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
- (non-sequential overwrites)

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:

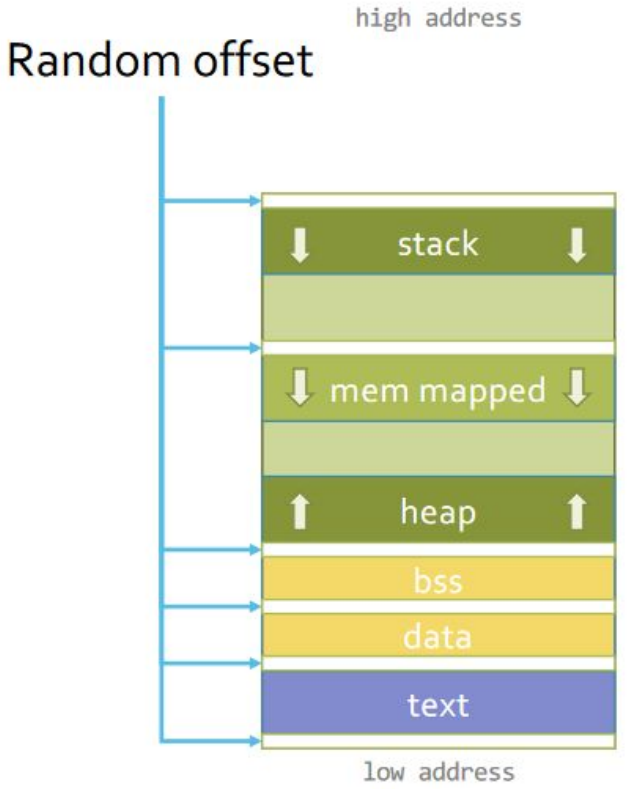
- Transfer control flow to an existing function (return-to-libc) (target 5)
- Return Oriented Programming (target 7)

Mitigations: ASLR (Address Space Layout Randomization)

Add random offsets to sections of process memory.

Bypasses:

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



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.

Evading Mitigations: Heap spraying

Motivation: Overflow can be used to cause control transfer into heap, but we don’t know where shellcode is stored due to ASLR.

Idea: use brute force. Allocate many copies of the shellcode (with big NOP sleds) and then jump blindly into the heap. Probabilistically, it will work?

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

Principles of secure system design

- Least Privilege
  - Faculty can only change grades for classes they teach
- Privilege separation
  - Multi-user operating system
- Complete mediation
  - Software fault isolation (SFI)
- Failsafe/closed
  - System call
- Defence-in-depth
- Keep-it-simple
  - Keeping the Trusted Computing Base (TCB) small and simple

System Security: Basic Memory Isolation

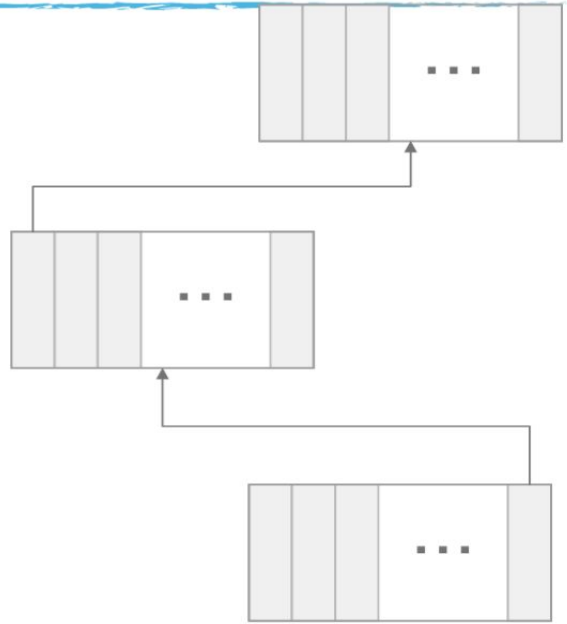
- **Process abstraction**
  - Each user can have one or more processes
  - Processes have UIDs (User IDs) that indicate what they're allow to access
- **Process isolation**
  - Keep processes from touching each other's memory or state directly
- **User/Kernel privilege separation**
  - Limit privileged operations to operating system kernel
  - Check requests from user against security policy
  - Protect operating system kernel from user processes

Basic Memory Isolation - Processes

- UID (RUID and EUID) / ACL / setuid
- How are individual processes isolated from each other?
  - Virtual Memory
  - Memory addresses used by processes are virtual addresses
  - Virtual addresses are mapped by the operating system into physical addresses, corresponding to actual storage locations
  - Address Translation : virtual -> physical

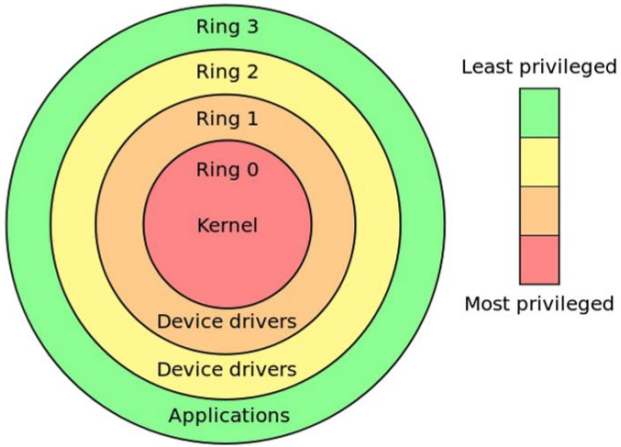
Page Tables

- Data structures used to store address mapping
  - Nodes of the tree
- Each table/node is:
  - Array of translation descriptors
  - Same size as a memory page
- Organized into a tree
  - Iteratively resolve n bits of address at a time
  - Each descriptor is either
    - Table descriptor (internal node)
    - Page descriptor (leaf node)



Basic Memory Isolation - Kernel/User

- OS has different privilege levels
- Each privilege level defines a set of entry points for less privileged callers



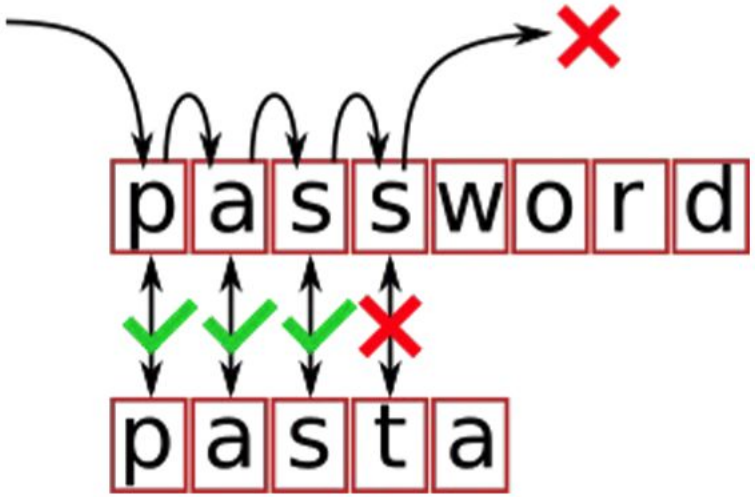
Basic Memory Isolation - Kernel/User

- These are the only valid entry points when calling from less privileged state : System Call : Need to be fast and efficient
- To make system calls fast, kernel’s virtual memory space is mapped into every process, but made inaccessible when in user mode
- Thus, separate permission bits in page tables

What is a side channel attack

- A side-channel attack is **a security exploit that aims to gather information from or influence the program execution of a system by measuring or exploiting indirect effects of the system or its hardware** -- rather than targeting the program or its code directly.

Side Channel: Time based



Side Channel: Cache based

- Processors try to “cache” recently used memory in faster, but smaller capacity, memory cells closer to the actual processing core
- Cache is a shared system resource
- Not isolated by process / VM / privilege

Side Channels - Cache

Possible attacks :

- Evict and time
  - Kick stuff out of the cache and see if victim slows down as a result
- Prime and probe
  - Put stuff in the cache, run the victim and see if your accesses are still fast (no conflict) or slow down (have been displaced by victim memory accesses)
- Flush and reload
  - Flush a particular line from the cache, run the victim and see if your accesses are still fast as a result

Side Channel: Spectre and Meltdown

Meltdown and Spectre exploit critical vulnerabilities in modern processors. These hardware vulnerabilities allow programs to steal data which is currently processed on the computer.

Meltdown breaks the most fundamental isolation between user applications and the operating system. This attack allows a program to access the memory, and thus also the secrets, of other programs and the operating system.

Spectre breaks the isolation between different applications. It allows an attacker to trick error-free programs, which follow best practices, into leaking their secrets.

Good luck!