Lecture 1

What is security?

- Computer security studies how systems behave in the presence of **an adversary**

The Security Mindset

- Thinking like an **attacker**
    - Understand techniques for circumventing security
    - Look for ways security can break (not reasons why it won't)
        - Looks for weakest links
        - Identify assumptions that security depends on
        - Not constrained by system designer's worldview

- Thinking like a **defender**
    - Know what you are defending, and against whom
    - Weigh benefit vs. costs
    - Rational paranoia (suspicion)

Thinking like a **defender**

- Security policies
    - What assets are we trying to protect
    - What properties are we trying to enforce
        - Confidentiality
        - Integrity
        - Privacy
        - Authenticity

- Threat models
    - Who are our adversaries
    - What's their motives and capabilities
    - What kinds of attacks do we need to prevent

- Assessing risk
    - What would security breaches cost us
        - Direct: money, property, safety
        - Indirect: reputation, future business, well being
    - How likely are these costs
        - Probability of attacks
        - Probability of success

- Countermeasures
    - Technical countermeasures
    - Nontechnical countermeasures
        - Law, policy, procedures, training, auditing, incentives

- Security costs
    - No security mechanism is free
        - Direct: design, implementation, enforcement, false positives
        - Indirect: lost productivity, the added complexity
    - Challenge is to rationally weigh the cost vs. risk

Secure Design

- Common mistake: convince yourself that the system is secure
- Better approach: identify the weakness of your design and focus on correcting them
- Secure design is a process

Where to focus defenses

- Trusted components
    - Parts that must function correctly for the system to be secure
- Attack surface
    - Parts of the system exposed to the attacker

Lecture 2

When is a program secure

- When it does exactly what it should
- When it does **NOT** do bad things
    - Delete or corrupt important files
    - Crash my system
    - Send my password over the internet
    - …

Weird machines

- Complex systems always contain **unintended functionality**
- An **exploit** is a mechanism by which an attacker triggers **unintended functionality**
- Security requires understanding the intended and the unintended functionality

What is a software vulnerability

- A bug: allows an **unprivileged** user capability that should be **denied**
- Most classic: **violating "control-flow integrity" (the attacker can run their code)**
- Involves violating assumptions of the programming language or its run-time

Starting exploits

- Low-level details of how exploits work (how can a remote attacker run their code)
- Threat model
    - Victim code is **handling input** that comes from across a security boundary
    - Want to protect the **integrity fo execution & confidentiality of data**

Buffer Overflows

- Definition: an anomaly (abnormal thing) that occurs when a program writes data beyond the boundary of a buffer
- Archetypal (original) software vulnerability
    - If your program crashes with memory faults, you probably have a buffer overflow vulnerability

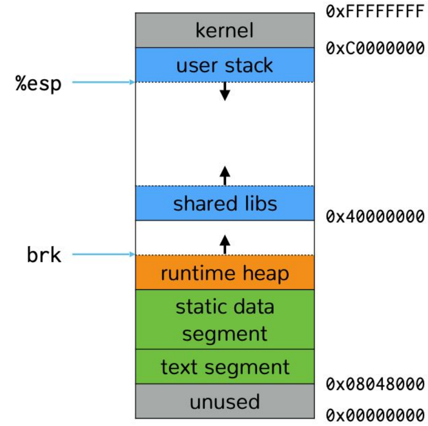
- **Why interesting**
    - Sometimes a single byte is all the attacker needs
    - Co-evolution of defenses and exploitation techniques

- **How are they introduced**
    - No automatic bounds checking in c/c++
    - Many C stdlib functions make it easy to go pass the bounds
    - ***String manipulation functions like gets(), strcpy(), and strcat() all write to the destination buffer until they encounter a terminating '\0' byte in the input***
    - !!! **whoever is providing the input controls**

- What do we need to know
    - How c arrays work
    - How memory is laid out
    - How function calls work
    - How to turn an array overflow into an exploit

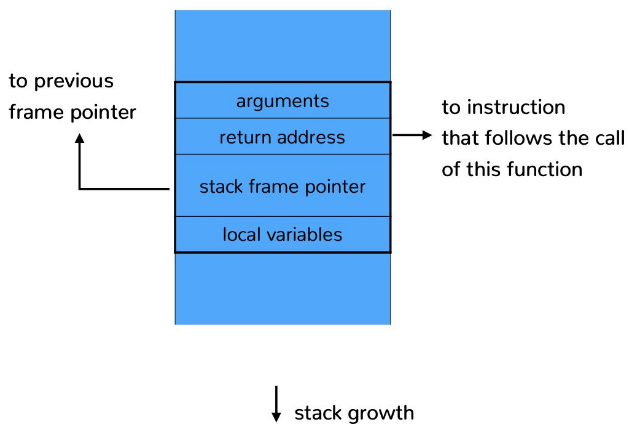
Linux process memory layout

- Stack: top
- Heap: under the stack
- Data: under the heap
- Text: under the data; executable code



Stack

- Divided into frames
- Stack pointer points to the top of stack (esp)
- Frame pointer points to caller's stack frame (ebp)
- Stack frame



- Example

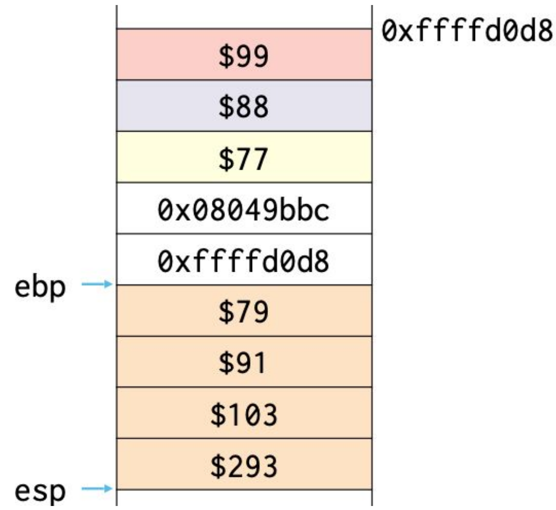
```c
1  int foobar(int a, int b, int c)
2  {
3      int xx = a + 2;
4      int yy = b + 3;
5      int zz = c + 4;
6      int sum = xx + yy + zz;
7
8      return xx * yy * zz + sum;
9  }
10
11 int main()
12 {
13     return foobar(77, 88, 99);
14 }
```

```
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
```

  -
  - Note that line 7: ebp is where foo starts, go down 4 bytes is where the first argument plus 2 stored.
  - Note at line 10: go down 8 bytes is where the second argument plus 3 stored

- Note at line 13: go down 12 bytes is where the third argument plus 4 stored
- Passed in arguments are stored at **eax**

- Then use **edx** as a temp to calculate the sum of xx, yy, and zz



- After returning, ebp jumps back to the saved ebp

<u>Stack Buffer Overflow</u>

- Source string of strcpy is controlled by the attacker, and **destination is on the stack**
    - The attacker gets to control where the function returns
    - The attacker can transfer control to anywhere

- Shellcode
    - Small code fragment that receives initial control in a control flow hijack exploit
    - The earliest attacks used shellcode to execute a shell
    - Restrictions
        - Cannot contain null characters (use NOP instead)
        - Must avoid line-breaks
        - The exact address of shellcode start is not easy to guess (NOP sled)

- Defenses
    - **Avoid unsafe functions**
        - Strcpy, strcat, gets, etc
        - **Cons:**
            - Non-library functions might be vulnerable
            - Requires manual code rewrite
            - No guarantee that you considered every possible vulnerability
            - Alternative functions also error-prone

    - **Stack canary**
        - Special value put before return address
        - If buffer overflows, it gets overwritten

- Check canary before returning
- **Automatically** inserted by compiler
- **Pros:** no code changes required, only recompile
- **Cons:**
    - Performance penalty
    - Only protects against stack smashing
    - Fails if attacker can read memory

- **Separate control stack**
    - WebAssembly has a separate stack
    - Separating the program into two **distinct** regions: safe & unsafe stack
        - Safe stack: *return address, register spills, local variables …*
        - Unsafe stack: everything else
    - **Cons:** control data is stored next to the user data
    - Modern usage: *Intel's shadow stack*
        - *Cannot update shadow stack manually*
        - *Need to rewrite code that manipulates stack manually*

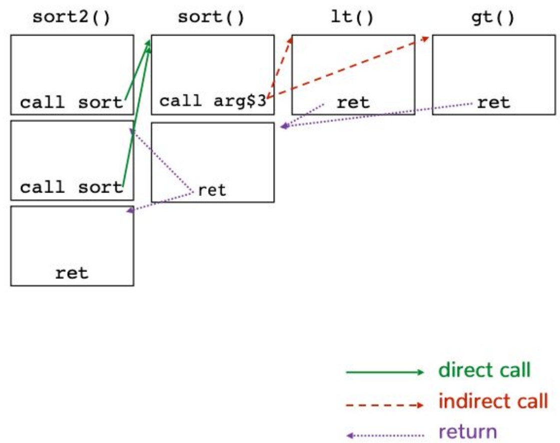
- **ASLR (address space layout randomization)**
    - Change location of stack, heap, code, static variables
    - Layout must be **unknown** to the attacker
    - Randomize on every launch at **compile time**
    - Implemented on the most modern **OS**es
    - **PaX memory layout → add random base between**
    - **Pros:** no code changes or recompile required
    - **Cons:**
        - Need compiler, linker, loader support
        - 32-bit architecture get limited protection
        - Fails if the attacker can read memory
        - Load-time overhead
        - No execution img sharing between processes

- **Memory writable or executable, not both (W ^ X)**
    - Use MMU (memory management unit) to avoid shellcode execution
    - Ensure memory cannot be both writable & executable
    - Code → executable, not writable
    - Stack, hea[. Static vars → writable, not executable
    - Supported by modern processors and implemented in modern systems
    - **Pros:** no code changes or recompiles required
    - **Cons:**
        - Require hardware support
        - Can be defeated by [**return-oriented** programming]
        - Does not protect [JITed code (Just In Time)]

- **CFI (Control flow integrity)**
  - Check the destination of every indirect jump
  - Make sure **function returns, function pointers, and virtual methods** are jumping to somewhere allowed and known to the caller
  - **Pros:**
    - No code changes or hardware support
    - Protects against many vulnerabilities
  - **Cons:**
    - Performance overhead
    - Require smarter compiler
    - Require having **all code available** (need to check)
    - Does not protect against **data-only** attacks
  - Basically, restrict control flow to legitimate paths
    - **Direct transfer of control flow (direct jumps...)**: **NO WORRIES**
      - Address is hard-coded. Not under attacker control
    - **Indirect transfer Ways**
      - Forward path: jump to an address in register or memory
      - Reverse path: return from function calls
    - **Control-flow graph example (CFG)**

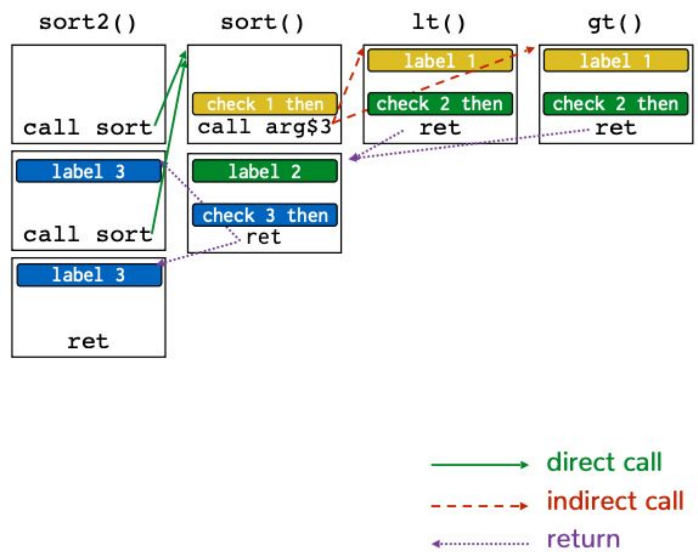
```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
  return x < y;
}

bool gt(int x, int y) {
  return x > y;
}
```

      -



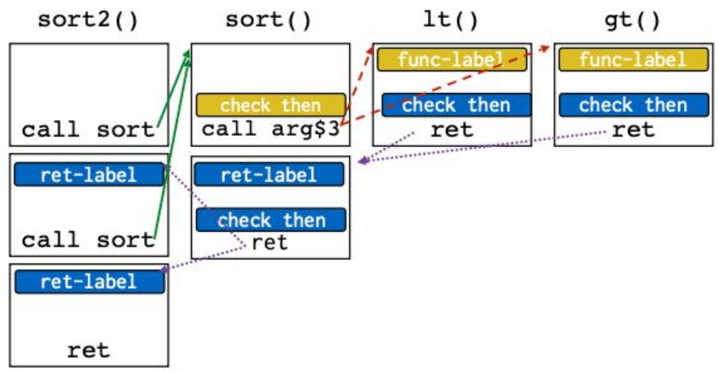
-

- **Restrict jumps to CFG (*Fine Grained CFI - Abadi et al.*)**
    - Assign **labels** to all indirect jumps and their targets
    - Validate that label before jumping
    - Need **hardware** support



-

- **Restrict jumps to CFG** (***Coarse-grained CFI - bin-CFI***)
    - Label for **destination** of indirect calls, rets and indirect jmp



-

-

Lecture 3

Defeat Buffer Overflow Protections

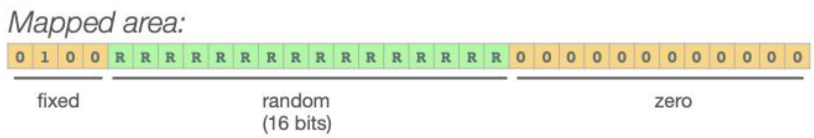
- **Stack canaries**
    - Assume it is impossible to subvert control flow without corrupting the canary
    - Attack
        - Targeted write gadge
        - Pointer subterfuge (trick - skip canary)
        - Overwrite function pointer elsewhere on the stack or heap
        - memcpy buffer overflow with fixed canary
        - Brute forcing in **forked process (same mem layout → guess & try)**

- **Separate control stack**
    - Need to compile c/c++ to WebAssembly
    - Put buffers, &var, and function pointers on the **user stack** such that it will overwrite function pointers when c programs compiled to WebAssembly
    - Shadow stack defeat
        - Find a function pointer and overwrite it to point to a shellcode

- **W ^ X: write XOR execute**
    - Still write to stack, and **jump to the existing code**
    - Search executable for code that does what you want (*system("/bin/sh"), libc ...*)
    - Find system call, replace the arguments → "**/bin/sh**"

- **ASLR**
    - Older Linux allows local attacker read the stack start address **"/proc/<pid>/stat"**
    - Each region has random offset, but layout is fixed

- Brute force for 32-bit binaries
- Heap spray for 64-bit binaries
- Derandomizing ALSR
    - Call **system()** with attacker argument
    - Target: [apache daemon]() **(a background process that handles requests for services, dormant when not required)**
    - Attack steps
        - Find base of mapped region



            - Layout is fixed
            - Guess return pointer to **usleep()** with **non-negative argument**
            - **65,536 tries maximum**
            - No need to derandomize the stack base
        - Call **system()** with attacker arguments (command string)
            - Overwrite saved return pointer with the address of **ret** instruction in **libc**
            - Repeat until the address of buf looks like argument to **system()**
            - Append address of **system()**

- **CFI**
    - Imprecision can allow for control-flow hijacking
        - Jump to functions that have the same label
        - Can then return to many more sites

- **Integer overflow attacks**
    - Example-1

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

        -

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

```
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';
}
```

- Three flavors (kinds) of integer overflows
  - Truncation bugs (assign 64 to 32)
  - Arithmetic overflow bugs (adding huge unsigned number - **ex2**)
  - Signedness bugs (treating signed number as unsigned **- ex1**)

Slide 4

Return-Oriented Programming

- Idea: make shellcode out of existing code
- Trick: code sequences ending in ret instruction
  - Overwrite saved eip on stack to pointer to first gadget, then second…
- Where to find those **ret** instructions
  - End of function
  - Any sequence of executable memory ending in **0xc3**
- Can express arbitrary programs
- Simple implementation
  - Write the instruction address on stack

- When return, esp subtracts
- Pop eip from the stack and get the next instruction

<u>Heap-based attacks</u>

- What if the attacker can cause the program to use **freed objects**
- Heap corruption
    - Bypass security checks (isAuthenticated, buffer_size, isAdmin, etc.)
    - Overwrite function pointers (especially **vtables**)
        - Each object contains a pointer to vtable
        - Vtable is an array of function pointers
        - Call looks up entry in vtable

<u>Use After Free (UAF)</u>

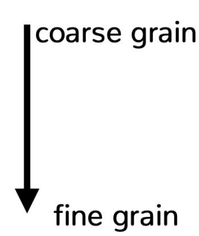
- Victim: free object: free(obj)
- Attacker: overwrite the vtable of the object so entry (obj → vtable[0]) points to the attacker gadget (*that was freed*)
- **<u>Temporal memory violation</u>**

<u>Slide 5</u>

<u>Principles of secure design</u>

- High-level idea
    - Separate the system into isolated least-privileged compartments
    - Mediate interaction between compartments based on security policy

- Unit of isolation
    - Physical machine
    - Virtual machine (**popular**)
    - OS process (**popular**)
    - Library
    - Function
    - …



- **The Process Abstraction**
    - Each process is memory isolated from each other
    - Each process has its own UIDs (read/write privilege)
    - Each file has ACL (access control list → owner, group, other)
    - Process UIDs

        - *Real user ID - RUID: parent's UID, who started the process*
        - *Effective user ID - EUID: determines permission for process*
        - *Saved user ID - SUID: save and restore EUID*
    - SetUID
        - Superuser root ID = 0, can access any file
        - Fork & execution system calls: inherit 3 IDs from parent
        - SetUID system call lets you **change EUID**
        - 3 bits
            - Setuid: set EUID
            - Setgid: set EGroupID
            - Sticky bit:
                - On: only file owner, directory owner, and root can rename or remove file in the directory
                - Off: if user has write permission on directory, rename or remove files, even if **not** owner

- **Mechanism**
    - ACL → restrict which process can access files (OS)
    - Namespaces → partition kernel resources between processes (Linux)
    - Syscall filtering → allow/deny system calls and filter on their arguments (Seccomp-bpf)
    - Common & necessary: **memory isolation**
        - Each process gets its own virtual address space
        - **Memory addresses used are virtual addresses not physical (VA, !PA)**
        - When & how to translate
            - Whenever there is a memory access performed (load, store, fetch)
            - CPU's memory management unit (**MMU**)
                - Page: basic unit of translation: 4Kb = 2^{12}
                - Use multi-level page tables: sparse tree of page mappings
                - Each process gets its own tree (**page table walking**)
                - **Kernel has its own tree**
        - **Access control**
            - Not everything within a process' VA is accessible
            - Page descriptors contain access control information
            - Example: kernel's VM(emory) is mapped into every process but inaccessible in **user mode**
        - TLB (translation lookaside buffer)
            - Small cache of **recently translated addresses**
            - Gives physical page corresponding to virtual page
            - Tells if page mapping allows the access control
            - When context switch
                - Flush the TLB
                - If has process-context identifiers (PCID), no need to flush

        - **Memory isolation in VM(emory)**
            - Isolate VM of one process from that of the other
            - Modern hardware supports **extended/nested page table entries**
            - TLB also tagged with VM ID(**VPID**, PCID) → address lookup
            - VMM is isolated from guest VMs: VMM is assigned VPID = 0

- **Key limitations**
    - Defeat VM/process isolation
        - Find a bug in the kernel or hypervisor
        - Find a hardware bug
        - Exploit OS/hardware **side-channels (cache based)**
            - Cache: smaller & faster
            - Kick out when collision
            - Shared system resource: Not isolated by process, VM, privilege level
    - Threat model: co-located VM
        - Attacker & victim are isolated **but on the same physical system**
        - Attacker is able to invoke functionality exposed by the victim
    - **Side channel**
        - Many algorithms have **memory access patterns**
        - Evict & time
            - Time the victim code
            - Evict parts of the cache & time it, repeat
            - Denote if slower, **then cache lines evicted must have been used by the victim**
        - Prime & probe
            - Prime the cache (access many memory locations so that previous cache contents are replaced)
            - Let the victim code run
            - Time access to different memory locations, <u>**slower** means **evicted**</u>
        - Flush & reload
            - Flush the cache
            - Let code run
            - Time access to different memory locations, <u>**faster** means **evicted**</u>

<u>**Slide 6**</u>

<u>**Malware**</u>

- Virus: code propagates by arranging itself to **eventually** be executed
- Worm: self-propagates by arranging itself to **immediately** be executed

- Malicious behavior: Runs with some user privileges
    - Malice
        - Can pop up messages
        - Trash files
        - Damage hardware

    - Espionage
        - Extract information
        - Keylogging, Screen capture, audio, etc

    - Economics
        - Botnet
        - Spam
        - Click Fraud
        - Extortion attacks
        - Steal credentials
        - Blackmail
- How does it run
    - Attack a network-accessible vulnerable service
    - Vulnerable client connects to remote system that sends over an attack “driveby”
    - Trick the user into running or installing (fake antivirus)
    - Attacker with local access downloads or runs directly

- Countermeasures
    - Signature-based detection
        - Look for bytes corresponding to virus code
        - Antivirus software is a multibillion dollar industry
    - Anti-virus arms race
        - Virus writers change viruses to evade detection
        - Virus encrypts its code; static code detection works less well
    - Cleanup
        - Rebuild from original media or backups
        - Some malware contains rootkits (hide its presence)
    - Analysis
        - Run in VM(achine) or sandboxed environment
        - Modern malware tries to detect if it runs in VM or fresh install and acts less maliciously

Slide 7

HTTP protocol

- Fetching resources from the internet (HTML documents)
- Resources have a uniform resource location (URL)

domain path fragment id

`https://cseweb.ucsd.edu:443/classes/fa19/cse127-ab/lectures?nr=7&lang=en#slides`

scheme port query string

-
- Clients & servers communicate by exchanging individual messages

Anatomy of a Request

method path version

- `GET /index.html HTTP/1.1`

headers

```
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
```

-

body (empty)

-

Anatomy of a Response

status code

headers

```
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: ...
Content-Length: 2543
```

body

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

-

Many HTTP methods

- GET: get 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

HTTP/2

- Major revision of HTTP released in 2015
- No major changes in how applications are structured. Major changes:
    - Allows **pipelining** requests for multiple objects
    - Multiplexing multiple requests over one TCP connection
    - Header compression
    - Server push

Cookies

- Small piece of data that a server sends to the browser
- The browser then stores it and sends it back with subsequent requests
- Useful
    - Session management: logins, shopping carts, etc
    - Personalization: user preferences, themes, etc
    - Tracking: recording and analyzing user behavior

- **Setting 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>
```

- **Sending cookie 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
```

Basic browser execution model

- Loads content → parse HTML & runs JS → fetch sub resources (imgs, CSS, …) → respond to events like onClick, onMouseover, etc.
- Nested execution model
    - Windows may contain frames from different sources
    - Frames provide **isolation**

- DOM (document object model)
    - JS uses DOM to manipulate objects or items in HTML

Attacker Models

- Network attacker
- Web attacker
- 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
- Pages share the same cookies/HTML5 local storage

<u>Same Origin Policy</u>

- Origin: isolation unit/trust boundary on the web (**scheme, domain, port**)
- SOP goal: isolate content of different origins
    - Script contained in **evil site** should not be able to read data in **bank.ch page**
    - Script from the **evil site** should not be able to modify the content of bank.ch

- SOP for DOM
    - Each frame has its own origin
    - Frame can only access data with the **same origin**
    - Communication between frames
        - Postmessage API

- SOP for HTTP responses
    - SOP prevents code from **directly** inspecting HTTP responses
    - **Documents**
        - Can load cross-origin HTML in frames, but not inspect or modify frame content
    - **Scripts**
        - Can load scripts from across origins
        - Scripts execute with the **same** privilege of the page
        - Page can see source thru `func.toString()`
    - **Images (similar for fonts & CSS)**
        - Browser renders cross-origin images
        - SOP prevents page from inspecting individual pixels though
        - Page can **only** see img.width
    - **Cookies**
        - Cookies use a separate definition of origins
        - DOM SOP: origin is (**scheme, domain, port**)
        - Cookie SOP: origin is (**scheme, domain, path**)
        - Browser will make a cookie available to the **given domain + sub-domains**
        - **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

- Cross-site request forgery attack (CSRF)
- Same Site cookies: sent **only when request is from the same site (top-level domain)**
- **Cookies are always sent**
    - Network attacker can steal cookies if server allows unencrypted HTTP traffic
    - Web attackers **DO NOT** need to wait for users to go to the site. Can make cross-origin requests
- Secure cookies: **sent only with <u>an encrypted request</u>**
- Finner grained isolation?
    - **NO.** cookies SOP does not allow domains to access the cookies of other domains of the same level, but DOM SOP does allow so

- **SOP does not prevent leaking data (document.cookie)**
- HTTPOnly cookies
    - Do not expose cookie in document.cookie

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; HttpOnly;
```

<u>Slide 8</u>

<u>Cross Site Request Forgery (CSRF)</u>

- HTTP methods related:
    - **GET**: retrieving data
    - **POST**: submit an entity, cause a change in state or side effects on the server

- Process
    - GET: use attacker's domain to interact with bank's URL, with user's own cookie
    - Attacker cannot see the result of GET but money all gone
    - POST: submit transfer form from attacker's site with user's cookie

- Defenses
    - Ensure that **POST is authentic (coming from a trusted page)**
    - **Secret Token validation**
        - Includes a secret value in every form submitted so that server can validate if the form is coming from a trusted page
        - Note that static token does not provide protection
        - **Use session-dependent identifier or token so attacker cannot retrieve due to SOP (attacker site and trusted site have different ORIGIN)**
    - **Referer or Origin validation**

- Referer request header includes URL of the previous web page from which a link to the currently requested page
- Referer header sends the full URL

https://bank.com	->	https://bank.com	✓
https://attacker.com	->	https://bank.com	X

- **SameSite cookies**
  - Strict: never send cookie in any cross-site browsing context
  - Lax: allowed when following a navigation link but blocks it in CSRF-prone request methods
  - None: send cookies from any context

Injection

- **Command injection**
  - Execute arbitrary command on the system
  - Pass unsafe 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);
}
```

  - **Normal Input:**

```
./head10 myfile.txt -> system("head -n 100 myfile.txt")
```

Adversarial Input:

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

- **Code injection**
  - Most high-level languages have safe ways of calling out to a shell, *eval* (don't use it)
  - **Incorrect:**

```
var preTax = eval(req.body.preTax);
var afterTax = eval(req.body.afterTax);
var roth = eval(req.body.roth);
```

Correct:

```
var preTax = parseInt(req.body.preTax);
var afterTax = parseInt(req.body.afterTax);
var roth = parseInt(req.body.roth);
```

- **SQL Injection (SQLi)**
    - Take user input and add it into the SQL string
    - Could possibly drop some table in SQL
    - Prevention
        - Never build SQL commands by yourself
        - Use parameterized (AKA prepared) SQL instead (**allows to pass in query separately from arguments**)
        - ORMs (Object Relational Mappers) (**provides interface between native objects and relational databases**)

__**Cross Site Scripting (XSS)**__

- When application takes untrusted data and sends it to a web browser without proper validation or sanitization

- 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>
```

-

- **Reflected XSS:** script is reflected back to the user as part of a page from the victim site
- **Stored XSS:** stores the malicious code in a resource managed by the web app (DB)
- **Defense**
    - Old times: filtering malicious content (cons: really hard)

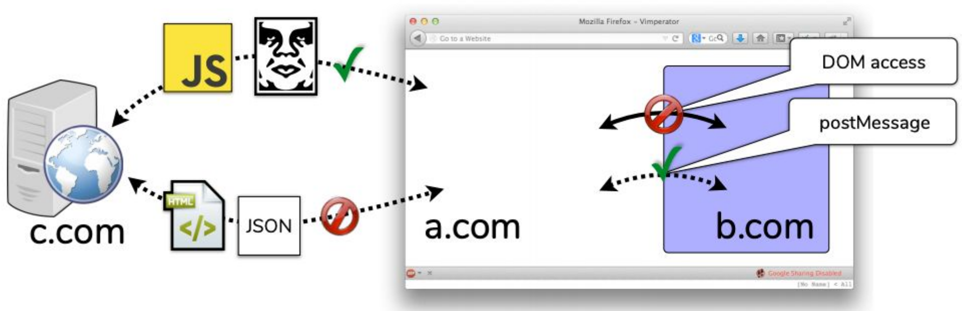
- **Content security policy (CSP)**
    - Need to specify the domains that the browser should consider to be valid sources of executable scripts
    - Examples
        - Content can only be loaded from the same domain, no inline script
        - Content-Security-Policy: default-src 'self'
        - Allow images from any origin
        - Restrict audio or video media to trusted providers
        - Only allow scripts from a specific server that hosts the trusted code, no inline scripts
        - Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com; script-src userscripts.example.com
    - Set up in **HTTP header, meta HTML object**
- **Trusted types**
    - Only allow values that have been **sanitized or filtered (type TrustedHTML)**

Using Untrusted or Vulnerable components

- SOP for Frames is a Lax (allowed when following a navigation link but blocks it in CSRF-prone request methods)

Slide 9

Recall: SOP

- Isolate content from different origins
- 

- Not strict enough
    - Third-party libs run with the **same privilege of the page**
    - Code within page can arbitrarily leak data
    - Iframe isolation is limited
- Not flexible enough

- Cannot read cross-origin responses

<u>Modern Mechanism</u>

- **Iframe sandbox**
  - Restrict actions iframe can perform
  - Whitelisting privileges

    allow-scripts: allows JS + triggers (autofocus, autoplay, etc.)

    allow-forms: allow form submission

    allow-pointer-lock: allow fine-grained mouse moves

    allow-popups: allow iframe to create popups

    allow-top-navigation: allow breaking out of frame

    allow-same-origin: retain original origin
  -
  - Run content in iframe with least privilege
  - Privilege separate page into multiple iframes

- **CSP**
  - Consider running library in sandboxed iframes (**desired guarantee: checker cannot leak password**)
  - **Problem:** sandbox does not restrict exfiltration
  - Restrict resource loading to a whitelist

- **HTTP strict transport security (HSTS)**
  - Attackers can force you to go to HTTP vs. HTTPS
  - HSTS: never visit site over HTTP again

- **Subresource integrity (SRI)**
  - CSP + HSTS can be used to limit damages but cannot really defend against malicious code
  - Idea: page author specifies hash of (sub)resource they are loading; browser checks integrity
  - When check fails
    - 1. Browser reports violation and does not render or execute resource
    - 2. CSP directive with integrity-policy directive set to report (report but may render or execute)

- **Cross-origin resource sharing (CORS)**

- **Recall: SOP is not flexible**
- Problem: cannot fetch cross-origin data
- Solution: cross-origin resource sharing (CORS)
    - Data provider explicitly whitelists origins that can inspect responses
    - Browser allows page to inspect response if its origin is listed in the header
- How it works
    - Browser send origin header with XHR request
    - Server can inspect origin header and respond with access-control-allow-origin header
    - CORS XHR may send cookies + custom headers

<u>COWL</u>

- Provide means for associating security label with data
- Ensure code is confined to obey labels by associating labels with browsing contexts
- **Confining the checker with COWL**
    - Express sensitivity of data (checker only receive pw if its context label is as sensitive as the pw)
    - Use postMessage to send labeled pw (at time of sending source, specify the sensitivity of the data)