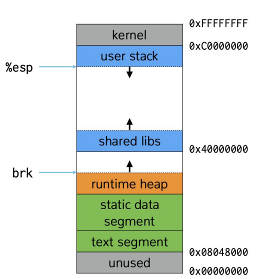
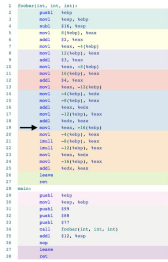
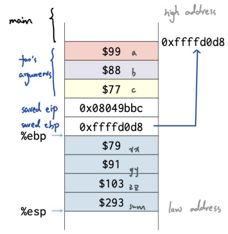
Stack Overflow







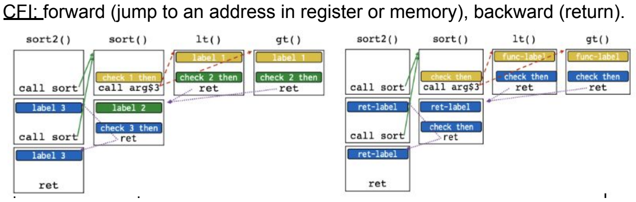
Avoid unsafe funct	Good idea in general	Requires manual code rewrite Non-lib funct vulnerable No guarantee everything found Alternatives also error
Stack canary	No code changes Only recompile	Performance penalty per return Only protects against stack smashing Fails if can read memory
ASLR	No code changes or recompile	32-bit arch get limited protection Fails if can read memory Load-time overhead
W^X (Data Extraction Prevention)	No code changes or recompile	Requires hardware support Defeat by ROP; Not protect JIT code;
CFI	No code changes or Hardware support Protect many vulns	Performance overhead; data-only attacks Requires smarter compiler Need all code available (see)

Stack Canary:

-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

Separate Control Stack: control stack, safe stack (return, register, local vars), shadow stack (%ssp) then compares %esp.

CFI: forward (jump to an address in register or memory), backward (return).



Defeats: (above: left fine-grained, right coarse-grained)

ROP: x86 instruction variable length, start on any byte boundary. 0xc3 means ret.

UAF: overwrite the vtable of freed object so entry points to attacker's code

Stack canary	1. pointer subterfuge (Fix: buffer closer to canaries; args copied to top of stack, pointers loaded into register before strcpy()); 2. Use memcpy other than strcpy null; 3. Chained exploited / Brute Force servers to learn the canary (fork process and guess).
Separate stack	Find a function ptr and overwrite it to point to shellcode; Put buffers, &var, and function pointers on the **user stack** such that overwrite function pointers when c programs compile to WebAssembly
W^X	Jump to existing code; Inject code with JIT, JIT spraying on heap overflow pointer.
ASLR	1. Find base through guess to usleep() [base + offset], max try: 2^16 = 65,536 tries. sleep, succeed; crash, next guess; 2. Call system() with &buf "/bin/sh".
CFI	Jmp to funct that has the same label, then return to more sites
IntOverflow	Truncation (assign 64 to 32); arithmetic overflow (0xffffffff + 2); Signedness bugs (0xffffffff = -1 > some num)

Isolation & SideChannel:

Separate into isolated least privileged compartments. Units: physical → **virtual machine** → **OS processes** → library → function (coarse → fine grained).

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: root = 0; setUID: EUID, setEGID: GID of file, sticky bit: on (only owner and root can rename or remove), off (then only if user has *w* permission)

Examples: Android apps has its own process ID, co limited using UNIX domain sockets + ref monitor checks permission; OKWS each server runs with unique UID, communication limited to structured RPC; modern browsers process; Qubes OS, trusted domain.

Memory Isolation: ACL, namespace (partition kernel), syscall filtering (scccomp)

Translation: MMU translate VA to PA, Page size = 2^{12} → multilevel page table. Each process/kernel has its own tree, context switch changes root.

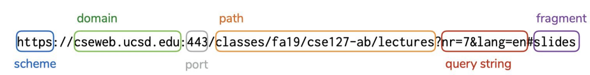
Translation LookAside Buffer: cache for address and access control. PCID for process context in the cached buffer. Extended nested page table entries (VPID for VMs).

ACL: page descriptors contain access control information (R, W, X).

Cache Side Channel: evict & time; prime & probe (time, slower means evicted); flush & reload (flush the cache, faster means evicted)

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.

Web Security Model



HTTP/2: allows pipelining requests for multi objects; multiplexing multiple requests over one TCP connection; header compression; server push

Cookies: a small piece of data server sends to browser, browser updates and sends it back with subsequent requests.

SOP: origin: isolation unit/trust boundary (*scheme, domain, port*). Isolate content of different origins;

SOP for DOM: each frame has its own origin; can only access data with the same origin; communication using postmessage API

SOP for HTTP responses: prevents code from directly inspecting HTTP responses; documents: can load cross origin but not inspect or modify frame content; scripts: can load cross origin, exe with same privilege of the page; images, fonts, css: can render cross origin but not inspecting each pixel

SOP for cookies: origin (*scheme, domain, path)* browser makes cookie available to given domain + sub-domains; path + child-path

Cookie 1:	Cookie 2:	Cookie 3:
name = mycookie value = mycookievalue domain = login.site.com path = /	name = cookie2 value = mycookievalue domain = site.com path = /	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

CSRF: 1. use attacker's domain to interact with banks url with user's cookie; submit transfer form from attacker's site with user's cookie. 2. Cookies can also be inspected through HTTP communication. → state-changing requests (authenticate POST).

Attacks: drive-by pharm hack routers; native apps run local servers. Login CSRF.

CSRF Defense: 1. Secret Token Validation (session-dependent identifier or token so attacker cannot retrieve due to SOP - attacker's site has different origin); 2. Referer/Origin validation: includes url of the previous web page; 3. 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 method; None: send cookies from any context; Secure: encrypted requests only; HttpOnly: cookie not in the document

Injection: Command injection: execute command on system bypassing unsafe data into shell (./head10 "myfile.txt; rm -rf /home");

Code injection: eval function

SQL injection: take user input and add it to SQL string; Defense: never build SQL commands by user.. Use parameterized (AKA ***prepared***, faster because of cached) SQL; ORM (object relational mappers) (provide interface between obj & DBs)

XXS: attacker inject scripting code into pages generated by a web application.

Reflected: the attacker script is reflected back to the user as part of a page from the victim site. E.g. paypal

Stored: the attacker stores the malicious code in a resource managed by the web application, such as a database. E.g. Samy, CSS for JS

DOM XSS: only allow sanitized TrustedHTML type values to end up in document.

CSP: whitelist valid domains of sources of executable scripts, as HTTP header or meta HTML object. E.g. Content-Security-Policy: default-src 'self'; img-src *; media-src: media.com.

frame-ancestors	Specify valid parents that may embed a page E.g. 'none' = X-Frame-Options: deny
upgrade-insecure-requests	Rewrite HTTP url to HTTPS
block-all-mixed-content	Don't load any content over HTTP

IFrame Sandbox: whitelist privileges. → privilege separate pages into multiple frames

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

HTTP Strict Transport Security: never visit site over HTTP again; strict-transport-security: max-age=n;
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 resources; 2. CSP directive with integrity-policy directive set to report (report but may render or execute)
Cross-Origin Resource Sharing (CORS): 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. DON'T use insecure JSONP.
Extension: different heap from main page; privilege separate between core extension script and content script; explicitly

<u>Others</u>

Parameterized SQL allows you to pass in query separately from arguments

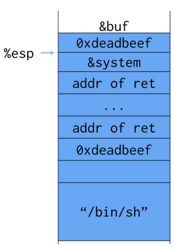
```
sql = "SELECT * FROM users WHERE email = ?"
cursor.execute(sql, ['nadiah@cs.ucsd.edu'])

sql = "INSERT INTO users(name, email) VALUES(?,?)"
cursor.execute(sql, ['Deian Stefan', 'deian@cs.ucsd.edu'])
```

Values are sent to server separately from command. Library doesn't need to try to escape

Benefit: Server will automatically handle escaping data

Extra Benefit: parameterized queries are typically *faster* because server can cache the query plan

Break ASLR, Stack Overflow to exe libc system "/bin/sh" Explained:



Browser Execution

Windows: load content, parse HTML, JS, fetch resources, respond to events.
Nested execution: frame as rigid visible division; iframe: floating inline frame. Delegate screen content from another soue.

SOP MORE

DOM	Each frame in a window has its own origin Frame can only access data with the same origin - DOM tree, local storage, cookies, etc.
HTTP	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.
Document	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.
Cookie	([scheme], domain, path). 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.

More CSRF

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.cookie

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.