SQL injections: a server making SQL query by client input can be exploited. For example the query `select person from people where name= '$name'` where `$name` is user input, the user can set `$name = "' or 1=1 -- "`. The query then will list every person in people: `select person from people where name='' or 1=1 -- .`

Similarly the user can set `$ = "'; drop table people --` , which will delete the table.

Similarly the user can use `union` to get information from other tables.

Blind SQL injections: suppose there is no visual representation of the results of the query. Instead we can create queries which ask true/false questions, and wait a certain amount of time upon a positive answer. For example, the following code checks if a password starts with ASCII 50, if so it waits 5 seconds.

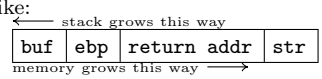
```
select if(substring(pswd,1,1) = char(50), benchmark(5000000,
encode('msg','by 5 seconds')), null) from users where id=1;
```

Prevention: sanitize user input (escape quotation marks, etc.). Use parameterized SQL queries, where the parameters are of a set datatype. So injecting something like `$name = "' or 1=1 -- "` will match *literally* when `name` is `' or 1=1 --`.

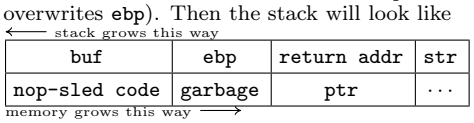
Buffer overflow: suppose we have code

```
void func(char* str) {
    char buf[126];
    strcpy(buf, str);
}
```

The stack will look like:



So if `str` is longer than 126 bytes, then it will begin to overwrite `ebp` and `return addr`. Suppose we could find an address in `buf`, `ptr`. Then we could write malicious code `code` and make `str = nop-sled code garbage ptr` (`garbage` overwrites `ebp`). Then the stack will look like



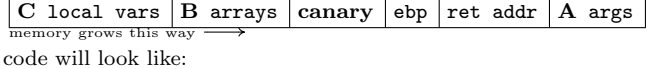
Then when the function returns it will return to where `ptr` points, which is hopefully inside `nop-sled` (a stream of `nop`s), and will eventually execute `code`.

Problems we may encounter are as follows: firstly we need to find an address inside `buf` for `ptr`. Secondly, our shellcode cannot contain any nul bytes, as these will cause `strcpy` to stop copying. We can also write it so that the return address is overwritten to point to existing code in the system.

A similar exploit can be found when we have a local variable which is a function pointer: we can use the buffer overflow to alter the function pointer to point where we want it.

Prevention: [1] type safe languages (e.g. Python, Rust), [2] safer functions (e.g. `strncpy`), [3] static source code analysis (use a tool to check static code for vulnerabilities), [4] canaries, [5] ASLR (place memory in a random location: makes getting the address for shellcode harder), [6] non-executable stack (so shellcode can't be put into buffer).

Canaries: we place a random number into memory called a **canary** to protect the return address from buffer overflow. Before returning check that the canary has not changed. Split frame into three (four, whatever) parts:



Our code will look like:

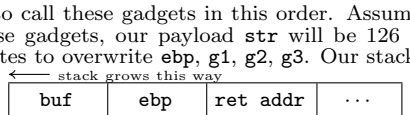
```
...  func(args...)  {
    int canary = XXX;
    // arrays
    // local vars
    /* function logic */
    if (canary != XXX) // exit logic
    return ...;
}
```

Return into libc: using what is called **DEP** we can make it so that memory is only either written to or executed (not both). But this can be overcome by **return into libc**: where an attacker overwrites `ret addr` to point into libc and execute code there. For example make it point to `system`, which runs its argument as a shell script.

Return oriented programming (ROP): the issue (seemingly) with ret2libc is that the attacker can only use functions in libc, and only directly. But this can be overcome. Libc has many **gadgets** which are small snippets of code ending in `ret` which are used by larger functions (e.g. `add $8, %esp; ret`). If we take control of the stack (e.g. through buffer overflow), we can chain these gadgets together to form malicious code. Suppose we have gadgets `g1, g2, g3` and the code

```
void func(char* str) {
    char buf[126];
    strcpy(buf, str);
}
```

and we want it to call these gadgets in this order. Assuming we know the addresses of these gadgets, our payload `str` will be 126 garbage bytes, 4 more garbage bytes to overwrite `ebp`, `g1`, `g2`, `g3`. Our stack will look like



Since we overwrote `ret addr` with `g1`, the function will return into `g1`, which then returns and since the top of the stack is `g2` it returns there. And similarly then returns into `g3`.

Double free attack: suppose we have code

```
p = malloc(100); q = malloc(100);
free(p); free(q);
p = malloc(200);
strncpy(p, str, 200);
free(q);
```

Instead of freeing `p` at the end, `q` was freed. `q` still points to the original area of memory, which was overwritten by `str`. `malloc` works by storing metadata on the allocated area of memory before the pointer, so the line allocating 200 bytes to `p` will have overwritten the metadata and `free(q)` will utilize this erroneous metadata.

As a basic model of malloc, suppose the metadata stored are two pointers: one to the left (previous) chunk, and one to the right (next) chunk (in total eight bytes). Since the data is 8 byte-aligned, the last three bytes of each pointer can be used as a flag, in particular we can have a *free* flag denoting if the chunk is not utilized. `free` will set this flag, and will merge the chunk with the left/right chunks if they are free. Excplicitly, `free(q)` will do `(q->lptr)->rptr=q->rptr` if `q->lptr` is free, and `(q->rptr)->lptr=q->lptr` if `q->rptr` is free.

Let us focus on `(q->lptr)->rptr=q->rptr`. We overwrite the metadata so `lptr` points inside the stack to `ebp`, and `rptr` points to the shellcode (potentially in the payload `str`). Then `(q->lptr)->rptr` is `ret addr`, and setting this to `q->rptr` sets `ret addr` to point to the shellcode.

Heap spraying: when ASLR is enabled, the previous attacks are now useless (since they require getting some address). Even if we use a nop-sled, the heap is so large that the nop-sled must be gigabytes long. Instead:

```
nopblock = "0c0c0c0c"
sled = nopblock * 256kb;
spray = new [sled + shellcode for i in range(1000)];
```

This will fill the heap up with 256mb of heap spray. `0c0c0c0c` is generally always in the heap, and since it corresponds to the `nop` instruction, if we use buffer overflow to write this into a `ret addr`, jumping to it will *likely* jump to a nop-sled in the heap leading to the shellcode. But heap spraying is not reliable, and can fail. Making it more reliable requires more memory being used in the spray, which will slow down the machine.

Heap Feng Shui: first fill up the heap with blocks the size of some object. Then free some of these blocks at the end, and place instances of the object in the other. Now write `0c0c0c0c` many times in the free blocks, so that they overflow and overwrite the instance of the object's vtable. Then the object's vtable will point to `0c0c0c0c`, and we can control the flow.

Encryption schemes: an encryption scheme has three algorithms: gen to generate a key, $\text{enc}_{k_e}(m)$ to encode plaintext m with encryption key k_e, and $\text{dec}_{k_d}(c)$ to decrypt ciphertext c with decryption key k_d.

One-time pad: Alice and Bob choose a random k, and encrypt plaintext m by $c = m \oplus k$. Then decrypt by $m = c \oplus k$. Call a cipher a **perfect cipher** if for a uniform distribution of messages, $\mathbb{P}(M = m \mid C = c) = \mathbb{P}(M = m)$, i.e. knowledge of the ciphertext doesn't give any knowledge of plaintext. One-time pad is perfect. But if we use OTP twice, notice that $c_1 = m_1 \oplus k, c_2 = m_2 \oplus k$ and so $c_1 \oplus c_2 = m_1 \oplus m_2$ and so we have gained knowledge of the plaintexts.

Secure schemes: a scheme is *secure* against an efficient adversary if when running in polynomial time, they can break the scheme with only negligible probability (smaller than $p(n)^{-1}$ for any polynomial $p(n)$).

Eavesdropping: a scheme is secure against eavesdropping if for any two messages m_1, m_2 no polynomial adversary can distinguish between encryptions of the messages. I.e. given $\text{enc}_k(m_b)$ the probability A outputs b is at most $\frac{1}{2} + \text{neg}(n)$ (neglible).

Pseudorandom generator (PRG): is a function mapping a short random seed (key) to a longer output such that if an adversary does not know the seed, it cannot distinguish the output from a truly random stream. Formally, G is a PRG iff for any polynomial time D the difference between the probability that D outputs "pseudorandom" when the input is from G and when the input is truly random, is negligible.

OTP with PRG is secure against eavesdropping: $\text{enc}_k(m) = G(k) \oplus m$ and $\text{dec}_k(c) = G(k) \oplus c$.

Chosen plaintext attack (CPA): the adversary outputs a pair of lists of messages $\vec{m}_i = (m_{i,1}, \ldots, m_{i,t})$ for $i = 0, 1$. Receives $\vec{c}_b = \text{enc}_k(\vec{m}_b)$, and must guess what b is. If a scheme is secure against CPA it must be nondeterministic: otherwise set $\vec{m}_0 = (0, 0)$ and $\vec{m}_1 = (0, 1)$ then if $\vec{c}_b = (c_0, c_1)$ output $c_0 \neq c_1$.

Pseudorandom function (PRF): is an effectively computable keyed function $F_k(\cdot)$: $\{0,1\}^n \to \{0,1\}^\ell$ such that an adversary cannot distinguish between it and a truly random function sampled from $\{0,1\}^n \to \{0,1\}^\ell$.

We can form a scheme using a PRF: $\text{gen}(1^n) \in \{0,1\}^n$, $\text{enc}_k(m) = (r, F_k(r) \oplus m)$ where r is chosen randomly, $\text{dec}_k(r, s) = F_k(r) \oplus s$. This is CPA-secure.

ECB: suppose we want to encode a long message, then assuming that F_k is efficiently invertible given k, we can then do

$$\text{enc}_k(m_1 \cdots m_\ell) = (F_k(m_1), \ldots, F_k(m_\ell)).$$

But this is deterministic and thus not CPA-secure.

CBC: instead what we can do is start with some initial vector $m_0 = \text{IV}$ and encrypt as follows: $c_i = F_k(m_i \oplus c_{i-1})$, and then transmit $\text{IV}, c_1, \ldots, c_\ell$. Decrypting is just $m_i \oplus c_{i-1} = F_k^{-1}(c_i)$ so $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$. Benefits of this is that: [1] the receiver can start decrypting at any ciphertext block, [2] errors in one ciphertext block only propogate to the next block, [3] conceals plaintext pattern (same block, different ciphertext block), [4] if IV chosen randomly and F_k invertible PRF then CBC is CPA-secure.

MAC authentication: suppose Alice sends Bob an encrypted, but Eve intercepts it and alters the contents. No matter how secure the encryption, it is susceptible to such attacks. A MAC scheme has three algorithms: $k \leftarrow$

$\text{gen}(1^n)$ generates a key, $t \leftarrow \text{MAC}_k(m)$ generates a tag t, $b \leftarrow \text{verify}_k(m,t)$ verifies if a message and tag pair are valid.

The adversary plays the following game: A is given pairs $(m_i, \text{MAC}_k(m_i))$ where m_i may be chosen by A (chosen plaintext). A then outputs (m,α) where $m \neq m_i$. A wins if $\alpha = \text{MAC}_k(m)$. The scheme is secure if A wins with negligible probability.

A fixed-length MAC is as follows: generate $k \in \{0,1\}^n$, $\text{MAC}_k(m) = F_k(m)$, and $\text{verify}_k(m,t)$ outputs if $t = F_k(m)$. This is only secure when the messages are n bits long.

CBC-MAC: suppose $m = m_1 \cdots m_\ell$, then $\text{MAC}_k(m) = t_\ell$ where $t_0 = 0^{|m|}$ and $t_i = F_k(t_{i-1} \oplus m_i)$. It is necessary to prepend $0^{|m|}$ since otherwise a CPA would be: get $(m_1, t_1), (t_1 \oplus m_2, t_2)$, then output $(m_1 \| m_2, t_2)$.

Hashing: a hash function is a function $h\colon X \longrightarrow Y$ where $|X| > |Y|$ such that (**weak collision resistance**) for any $x \in X$ it is hard to find $x' \neq x$ such that $h(x) = h(x')$. Or (**strong collision resistance**) it is hard to find a pair x, x' such that $h(x) = h(x')$. Instead of MAC on m, MAC on $h(m)$ (requires at least weak collision resistance).

Authenticated encryption: generate $k = (k_e, k_m)$, $c = \text{enc}_{k_e}(m), t = \text{MAC}_{k_m}(c)$, send (c,t). Verify $\text{verify}_{k_m}(c,t)$ and decrypt $m = \text{dec}_{k_e}(c)$. It is necessary to MAC the ciphertext, since the tag may reveal information about the message.

Hard problems in cyclic groups: DL (discrete log): given $g, h \in G$ find n st $g^x = h$. **CDH** (computational Diffie-Hellman): given $g^{n_1}, g^{n_2} \in G$ compute $g^{n_1 \cdot n_2}$. **DDH** (decisional Diffie-Hellman): given $g^{n_1}, g^{n_2}, h \in G$ return if $h = g^{n_1 \cdot n_2}$. These problems are not hard in all cyclic groups, but they are believed to be hard in cyclic groups of prime order. Note that $\mathbb{Z}_p^\times$ is not of prime order, but it is cyclic.

Diffie-Hellman key exchange: take a generator $g \in G$ and randomly choose an integer $n \in \mathbb{Z}_q$ ($q = |G|$), compute $h_A = g^n$. Then Alice sends Bob $(G, g, q), h_A$. Bob chooses $m \in \mathbb{Z}_q$ and computes $h_B = g^m$ and sends Alice h_B. Alice sets $k_A = h_B^n$, and Bob sets $k_B = h_A^m$. Note $k_A = g^{mn} = k_B$. If DDH is hard in G, then this protocol is secure in eavesdropping.

Public-key encryption (PKE): $\text{gen}(1^n) = (k_s, k_p)$, $\text{enc}_{k_p}(m) = c$, $\text{dec}_{k_s}(c) = m$. PKE must be nondeterministic, as we cannot have an adversary distinguish between two messages.

ElGamal encryption: $k_p = (G, g, q, h)$ where $h = g^n$ for some $n \in \mathbb{Z}_q$ and set $k_s = n$. $\text{enc}_{k_p}(m)$, choose a random $r \in \mathbb{Z}_q$ and set $c = (g^r, h^r \cdot m)$. $\text{dec}_{k_s}(c_1, c_2) = \frac{c_2}{c_1^{k_s}}$. Note that

$$\frac{c_2}{c_1^{k_s}} = \frac{h^r \cdot m}{c_1^n} = \frac{g^{nr} m}{g^{nr}} = m$$

Alternatively, use a hash function H and encrypt using $(g^r, H(h^r) \oplus m)$ and decrypt by $H(c_1)^n \oplus c_2$.

Textbook RSA: in general given $N = pq$, it is hard to factor and find p, q. RSA works as follows: $\text{gen}(1^n) = (k_p, k_s)$ where $k_p = (N, e)$ and $k_s = d$ st $ed \equiv 1 \bmod \varphi(N)$. $\text{enc}_{k_p}(m) = m^e \bmod N$ and $\text{dec}_{k_s}(c) = c^d \bmod N$. The issue is that while e may be chosen to be small, d may be large and so decryption expensive. Note though that $x^y \equiv x^{y \bmod \varphi(N)} \bmod N$. And using CRT, $y \equiv a \bmod p, y \equiv b \bmod q$ and so can compute $a^d \bmod p, b^d \bmod q$ (can use $d \bmod p-1, q-1$), and from that get $y^d \bmod N$. But textbook RSA is deterministic and thus not CPA secure.

Padded RSA: if messages are $|N| - L$ bits, choose a random string r of L bits and encrypt with $(r \| m)^e \bmod N$. After decrypting, discard the first L bits. Not proven to be secure.

Digital signature: allows someone to sign something, anyone can verify that the signature is valid, but only the sender can sign. A digital signature must have three algorithms: $(k_p, k_s) \leftarrow \text{gen}(1^n)$, $\sigma \leftarrow \text{sign}_{k_s}(m)$ and output (m, σ), $\text{verify}_{p_k}(m, \sigma)$ verifies that σ is a valid signature of m.

Consider the game where the adversary A has k_p and an oracle which signs messages. A then outputs (m, σ) where m was not queried to the oracle. A wins if σ is a valid signature. The scheme is **existentially unforgeable** under chosen message attack if A wins with negligible probability.

For example using RSA: have p, q, N, e, d as in RSA, $k_p = (N, e)$ and $k_s = d$. Then sign by $s = m^d \bmod N$. Verify by checking that $s^e \equiv m \bmod N$.

This is not secure: an attacker needs to generate (m, s) st $m \equiv s^e \bmod N$. So simply choose s and compute $s^e \bmod N$. And suppose it wants to sign m, choose a random r and sign $m^* \equiv mr^e \bmod N$, then $s^* \equiv (mr^e)^d \equiv m^d r \bmod N$, then compute $s \equiv s^*/r \equiv m^d \bmod N$. So (m, s) is valid signature.

Instead choose a hash function H, and have $\text{sign}(m) = H(m)^d \bmod N$. This protects against the previously mentioned attacks.

PK authority: note that Diffie-Helman is vulnerable to MITM. Can use digital signatures. Choose from a dictionary of public keys called a **PK authority**. But what if the PK authority is tampered with? We use **certificates**: suppose Charlie knows that k_p^B is Bob's public key, and Charlie has (k_s^C, k_p^C), then Charlie generates a certificate:

$$\text{cert}_{C \longrightarrow B} = \text{sign}_{k_s^C}(\text{“Bob's key is } k_p^B\text{”}).$$

If Bob wants to talk to Alice, and Alice knows Charlie, then Bob can send her $\text{cert}_{C \longrightarrow B}$. But what if Alice trusts neither Bob nor Charlie? Create a chain of **trusted certificate authorities** where each CA certifies others, then Alice can verify Bob's CA by checking that it is verified in some chain from her CA. Certificates must be able to be revoked, and have expiry dates for security.

SSL/TLS: these are the protocols used for secure communication with a server. A simplified model of an SSL handshake is as follows: [1] the client sends the ciphers it supports to server as well as a random R_C, [2] server sends back certificate using k_p^{server}, the cipher it chooses, and R_S, [3] client computes the master key $MK = f(S, R_C, R_S)$ and sends $\text{enc}_{k_p^{server}}(S), \text{MAC}_{MK}(\text{transcript})$, [4] the server decrypts S and computes MK, verifies MAC, then sends $\text{MAC}_{MK}(\text{transcript})$ which is verified by client.

URLs: a url has the following structure:
`<protocol>://<host>:<port>/<path>;<params>?<query>#<fragment>`

HTTP: HTTP is a protocol for sending requests to servers. A request has three parts: [1] the request itself (protocol, type of request, url), [2] headers, [3] contents. A response has three parts: [1] the status of the response, [2] headers, [3] contents. HTTP messages are written in readable text. HTTP is stateless, it cannot save states. To overcome this, we use **cookies** which are small key-value files that a browser stores for each website. One common cookie to store is `session_id`, which is a unique identifier for the current session on a website. The server stores information keyed by the session id.

CSRF: suppose Alice uses a bank which transfers money using a GET request, like so:

`https://bank.com/transfer?amount=5000&account=1234`

The server will check the `session_id` cookie sent and transfer this amount to account 1234. Now suppose an attacker wants to transfer \$5000 to account 666. Assuming Alice has an active session with the bank currently (and is thus signed in), the attacker may trick her into opening an innocent-looking website `https://nice.com` which has an image tag

`<img src="https://bank.com/transfer?amount=5000&account=666">`

The browser will open this link automatically, expecting an image, without Alice's consent. Since Alice is signed into the bank, it will send the session id, and the bank will follow through with this request.

Same-origin policy (SOP): only allow you to read HTML objects which originate from the same **origin**, which is a thruple: [1] protocol (e.g. https), [2] address (must fully agree, cannot be a subdomain), [3] port. The opposite of this, which is disabled by default usually, is **cross origin resource sharing (CORS)**.

CSRF token: consider now the previous example with Alice in CSRF. Now suppose the bank serves a **CSRF token** to each session, which is a large random number. To complete a transaction, you must include this token in the URL. Since the attacker cannot know this token, it cannot complete its malicious request.

SameSite cookies: a cookie can have the following SameSite attributes which dictate when the cookie is sent upon a cross-site request: [1] none (the cookie is always sent), [2] lax (the cookie is sent only on a safe method (e.g. GET, not POST), and only when the change in site is top-level (changes the URL in the top bar, e.g. not iframe)), [3] strict (the cookie is never sent upon cross-site request).

Clickjacking: the attacker hides malicious HTTP objects at the top of the page (so they are clicked and not some other object; hide by setting opacity to zero). Place at the same location some misleading text (win a million dollars!) so they click it, but really they click the malicious HTTP object. For this reason, many websites don't allow for them to be viewed through an iframe.

XSS: inject malicious code into a website which doesn't properly sanitize user input. [1] Stored XSS: the script is stored on the server, [2] reflected XSS: the script comes from the URL, [3] DOM-based XSS: the script comes from the DOM, i.e. input from the user.