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:

$\longleftarrow$ stack grows this way

buf	ebp	return addr	str

memory grows this way $\longrightarrow$

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

$\longleftarrow$ stack grows this way

buf	ebp	return addr	str
nop-sled code	garbage	ptr	···

memory grows this way $\longrightarrow$

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:

$\longleftarrow$ stack grows this way

C local vars	**B** arrays	canary	ebp	ret addr	**A** args

memory grows this way $\longrightarrow$

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

$\longleftarrow$ stack grows this way

buf	ebp	ret addr	···	
garbage	garbage	g1	g2	g3

memory grows this way $\longrightarrow$

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, $\mathrm{enc}_{k_e}(m)$ to encode plaintext m with encryption key k_e, and $\mathrm{dec}_{k_d}(c)$ 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 *perfect* 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 $\mathrm{enc}_k(m_b)$ the probability A outputs b is at most $\frac{1}{2} + \mathrm{neg}(n)$ (neglible).