CSCI4130 Introduction to Cyber Security

Ryan Chan

March 29, 2026

Abstract

This is a note for **CSCI4130 Introduction to Cyber Security**.
Contents are adapted from the lecture notes of CSCI4130, prepared by Wei Meng, as well as some online resources.
This note is intended solely as a study aid. While I have done my best to ensure the accuracy of the content, I do not take responsibility for any errors or inaccuracies that may be present. Please use the material thoughtfully and at your own discretion.
If you believe any part of this content infringes on copyright, feel free to contact me, and I will address it promptly.
Mistakes might be found. So please feel free to point out any mistakes.

Contents

Chapter 1

Introduction

Cyber Security is about keeping computing systems working as intended, keeping data accessed only as desired, and keeping the computing resources secured. All of these need to be done in the presence of an adversary and on a budget. We first introduce some concepts in this chapter.

Threat Model

A threat is a potential violation of security by an attacker. Since it is not possible to protect against all possible attackers, we need risk management. Computer security is always related to a specific threat model.

Threat modelling works to identify, communicate and understand threats and mitigations within the context of protecting something of value. A threat model includes:

- What to protect

- From whom to protect

- Assumptions (can be wrong)

- A list of the (most important) relevant threats

- What actions are to be taken for each threat

Security Policy

This is a statement on what is and what is not allowed.

Security Mechanism

This is a method, tool or procedure for enforcing a security policy. A security policy can be implemented by a variety of security mechanisms. For a security mechanism to work,

1. the security policy correctly describes the security requirement of a system

2. the security mechanism is implemented correctly

3. the security mechanism is deployed correctly

4. the security mechanism cannot be bypassed

Prevention

Prevention aims to prevent something bad from happening.

Detection

Detection is about knowing that something is wrong. There are two special cases, which are false positive, meaning this is a wrong alarm, and FALSE NEGATIVE, meaning there is an unnoticed attack.

Response

Response is needed when we detect something is going wrong.

Mitigation

Mitigation can be more economical and practical than prevention.

Chapter 2

Control Hijacking

Application software is computer software that performs a group of pre-defined functions or tasks. It can provide services locally or remotely, and its behavior is determined by the data/input being processed, the code being executed, and the environment in which it runs.

Compromising the security properties, i.e. **Confidentiality**, **Integrity**, and **Availability**, is the goal of attackers. But how can they achieve this?

2.1 Introduction

Control hijacking attacks are a class of attacks that hijack the control flow of an application. Since application code is loaded into memory, attackers may corrupt memory through techniques such as buffer overflows, integer overflows, and format string exploitation. Before diving into the details, we first look at several related concepts.

2.1.1 Control Flow Graph

Applications may invoke external commands to perform specific tasks. For example, `system(<string>)` executes a command specified in a string by calling `/bin/sh -c <string>`. Therefore, once users control the string content, they can inject arbitrary commands.

It is helpful to understand the behavior of an application, i.e. its control flow. This describes the order in which code is executed or interpreted, and it can represent the functionality or behavior of a program to some extent.

We can use a **Control Flow Graph (CFG)** to graphically represent the control flow of a program. It shows all possible paths that might be traversed during the execution of the program.

A Control Flow Graph contains nodes, which are blocks of code without branches, connected by directed edges. A CFG should have an entry node and an exit node. A path is a sequence of nodes connected by edges. Note that not all paths are feasible.

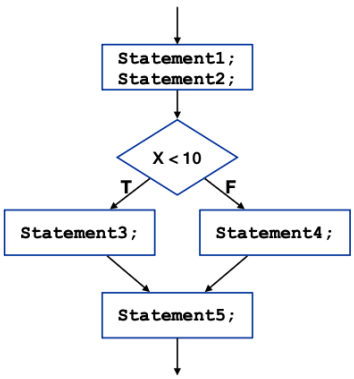

Figure 2.1: Control Flow Graph

2.1.2 Program State

A program state can be described by the values of processor registers and memory, such as the stack and heap.

Processor registers provide several quickly accessible storage locations. They hold data used in arithmetic and logical operations, and data is copied between registers and main memory as needed.

For memory, the stack usually grows downward (toward lower memory addresses). The stack pointer `ESP` points to the top of the stack. The `push` instruction decrements `ESP` and stores a value at the address pointed to by `ESP`. The `pop` instruction retrieves the value at the top of the stack into a register and then increments `ESP`.

The stack is composed of frames. Frames are pushed onto or popped from the stack when a function is invoked or returns. Each frame contains the function's actual parameters, the return address, a pointer to the previous frame, and the function's local variables (as shown in Figure 2.2). The stack base pointer `EBP` stores the address of the current frame.

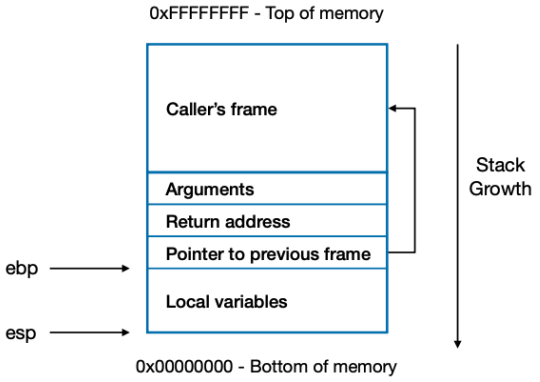

Figure 2.2: Stack Frame Layout

A simple walkthrough of a function call can be illustrated using the following example:

```
#include <ctype.h> // tolower
#include <string.h> // strcmp
#include <stdio.h> // fgets, fputs

void reveal_secret() {
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name) {
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main() {
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

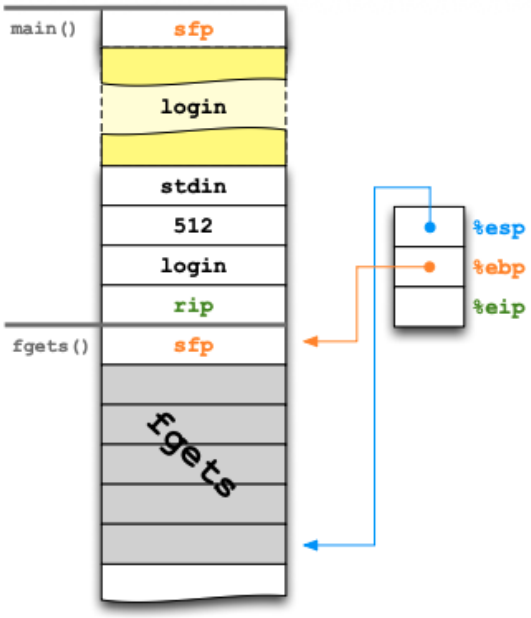

Figure 2.3: Stack - Function Call

Consider the entry point of this program at `int main()`. The saved frame pointer (SFP) points to the old frame pointer (OFP), which is pushed onto the stack, causing `ESP` to decrement. Since this is now the current frame, `EBP` will point to this position and will remain unchanged unless another function is called.

Next, `login[512]` is allocated on the stack. The program then reaches the first function call, `fgets`. The arguments of the function are pushed onto the stack in reverse order, i.e. `stdin`, `512`, and then `login`. After that, the return instruction pointer (RIP) is pushed onto the stack so that the program knows where to resume execution after the function call returns.

These steps repeat for subsequent function calls until the program finishes executing all function calls.

2.2 Attacks

Overflow is one of the most popular classes of attacks, as it is exploitable both locally and remotely, and it can modify both the data and the control flow of an application.

More specifically, we will discuss stack overflow. A function's local variables (buffers) are statically allocated on the stack, and data may be copied or written without boundary checking. Therefore, data can overflow a buffer and overwrite other important data, e.g. the return address or the saved `EBP`. With a carefully crafted input, it is possible to overwrite security-sensitive data, redirect execution to user-defined code, or even reuse existing code. We will examine three cases.

2.2.1 Overflow – Crash

Consider the example above. Suppose the user inputs a value of 500 bytes. After the `fgets` function call finishes, the program proceeds to the `verify(login)` function call. Inside `int verify`, the program iterates through the input and calls `tolower`. The resulting characters are then stored in the array `user[256]`.

Since the length of `name` is 500, writing the data into `user[256]` causes the buffer to overflow. As a result, the content beyond `user[256]` overwrites other data on the stack, including the saved frame pointer and the return address (instruction pointer / program counter, `EIP`). When the function returns, the program attempts to jump to the corrupted return address. This typically redirects execution to an invalid or unknown address, causing the program to crash.

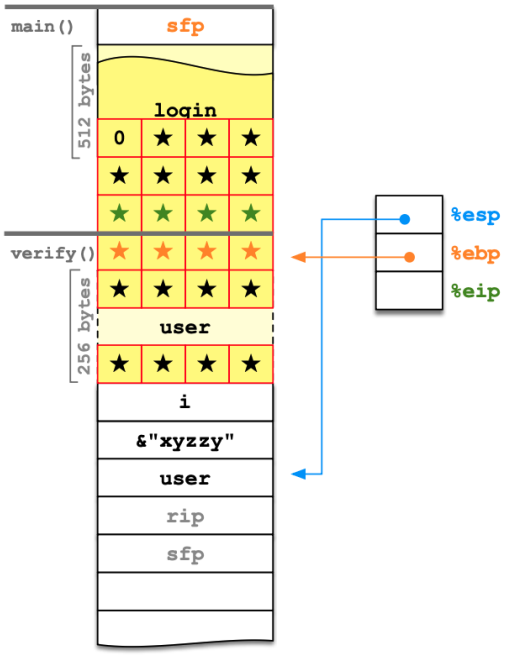

Figure 2.4: Stack Overflow - Crash

2.2.2 Overflow – Inject

Assume the attacker provides input containing machine code (shellcode), e.g.

```
a0c2d782
ffa86db2
307abba9
ad7c....
```

When this input is processed by the program (`user[i] = tolower(name[i])`), the bytes are copied into the buffer `user`. If the input exceeds the buffer size, a carefully crafted overflow can overwrite the saved frame pointer (SFP) and the saved return address (RIP). By replacing the saved return address with the address of the injected shellcode in the buffer, the attacker can cause the program to jump to the shellcode when the function returns. The shellcode can then execute a system call such as `execve("/bin/sh")` to spawn a shell.

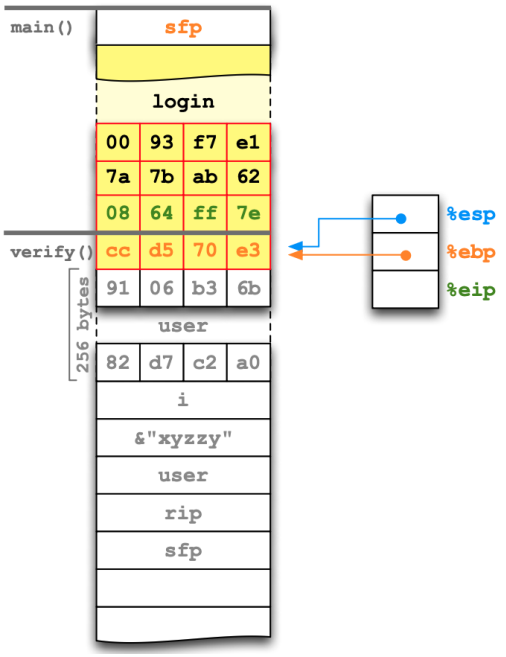

Figure 2.5: Stack Overflow - Inject

2.2.3 Overflow - Divert

This type of overflow is similar to the code injection. Assume the address of `reveal_secret` is `0xc8f4f2a9`. Similar to the previous case, with a carefully crafted overflow, the user can overwrite the return instruction pointer (RIP) with `0xc8f4f2a9`. Then, when the function call ends, the program will jump to `reveal_secret` and display the message 'SUPER SECRET = 42'.

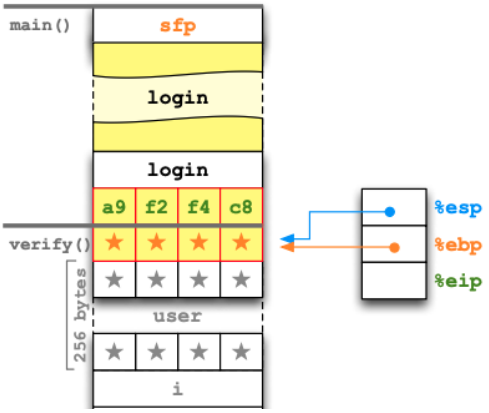

Figure 2.6: Stack Overflow - Divert

There are several other vulnerable functions in C/C++, e.g. `fgets()`, `strcpy()`, `strcat()`, `sprintf()`, `vsprintf()`, `scanf()`, `sscanf()`, etc. Any reference to a value that can be overwritten may represent a security vulnerability. For example, we can overwrite the following:

1. Data and data pointers
2. Return address
3. Stack frame pointer
4. Exception handler
5. Function pointer
6. Jump destination

Therefore, safer versions of memory access functions should be used to ensure that important data structures, i.e. the stack, heap, and others, cannot be overwritten. However, there are still some possible overflows.

2.2.4 Integer Overflow

Integer overflow is caused by unexpected results when comparing, casting, or performing arithmetic operations on integers. There are several cases:

1. A large signed integer can become a negative value.
2. A large unsigned integer can become a small unsigned integer.
3. A negative signed integer can become a large unsigned integer.

Therefore, integers of different sizes or types should be assigned with great care, and assertions or similar checks can be used to avoid overflows.

2.2.5 Index Overflow

Index overflow exploits the lack of boundary checks when referencing a value in an array using an index. This allows direct writes to memory locations. Depending on the situation, it may only be possible to modify memory values that conform to the type of elements in the array.

For example, consider an array declared as `array[10]`. If the program reads input without checking the bounds and writes to a position beyond indices 0 to 9, it causes an overflow.

> **Remark.** In C++, objects can be dynamically allocated on the heap. Each object is associated with a pointer pointing to the virtual function table (vtable), which is a table of pointers to the definitions of the object's methods. A variable overflow can be used to overwrite this pointer so that it points to a fake virtual function table specified by the attacker.

The goal of attackers who attempt to corrupt the memory of a process is usually to take control of it such that they can run with the privileges of the process and execute attacker-specified code. These attacks are made possible by mixing data and control in memory and allowing users to overwrite control data.

Therefore, we need to look at the defenses against such attacks.

2.3 Defenses

We want to make sure that for memory accesses, including read, write, and execute, there is no access to 'undefined' memory. Note that 'undefined' is with respect to the semantics of the programming language. To make sure the code executes safely, we utilize a module-by-module analysis.

2.3.1 Conditions

Here we discuss two conditions: preconditions and postconditions.

Preconditions describe what must hold for a function or statement to operate correctly. Formally, a precondition is an assertion that must hold about the inputs of a function for it to work correctly. The caller must satisfy the precondition, and the callee may assume that the precondition is met.

Postconditions describe what holds after the function returns or the statement is executed. A postcondition is an assertion that must hold after a function returns. The caller may assume that the postcondition holds, while the callee must ensure that the postcondition holds. For consecutive statements, the postcondition of statement 1 should logically imply the precondition of statement 2.

There are also **invariants**, which are conditions that always hold at a given point in a function.

For example, consider

```
int deref(int *p) {
  return *p;
}
```

The precondition that must hold is that the pointer `p` is not `NULL` and is valid. For

```
void *my_malloc(size_t n) {
  void *p = malloc(n);
  if (!p) {
    perror("malloc");
    exit(1);
  }
  return p;
}
```

The postcondition here is that the return value is not `NULL` and `p` is a valid pointer.

In general, the correctness proof strategy for memory safety is to identify each memory access point, write down the precondition it requires, and propagate the requirement up to the beginning of the function.

In cases with more complicated loops, induction might be used. The base case corresponds to the first entrance into the loop, and the induction step shows that the postcondition of the last statement in the loop, together with the loop test condition, implies the invariant.

2.3.2 Approaches

To defend against attacks due to memory corruption, we have two approaches.

Prevention

We aim to write bug-free programs, or rewrite programs in a memory-safe language that automatically checks boundaries and does not allow unsafe pointer arithmetic. We can also analyze programs for potential vulnerabilities before execution, prevent attack code execution, and make exploitation harder.

Detection

We detect attacks by performing security checks during program execution, such as detecting abnormal sequences of executed instructions and checking for integrity violations.

We now look at four types of defense.

2.3.3 Canary

Canaries are placed in stack frames to prevent and detect overwriting of the return address on the stack. StackGuard inserts a canary before the return address in each stack frame, such that when the function returns, the canary value is verified against the stored value. This acts as a runtime check for stack integrity.

There are different types of canaries:

1. **Terminator Canary**

This type of canary uses `NULL` (`0x00`), `CR` (`0x0d`), `LF` (`0x0a`), and `EOF` (`0xff`). String functions will not copy beyond a terminator, so attackers cannot use such functions to overwrite the return address.

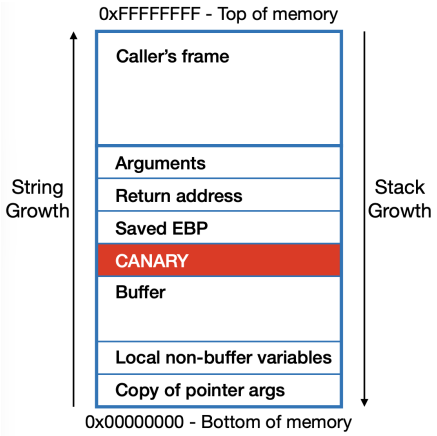

Figure 2.7: Stack with canary

However, terminator canaries are not completely random. For example, there are always fixed zero bytes, making it easier to guess the canary and potentially brute-force it.

2. **Random Canary**

A random value is chosen at program startup and stored in a location known only to the validation code, often protected using unmapped pages.

3. **XOR Canary**

This uses `random value` $\oplus$ `return address`.

There are also limitations to canaries. Canaries do not provide full protection because they can be learned. For example, there might be format string vulnerabilities or information leakage, and canaries can also be overwritten. Moreover, other pointers can still be overwritten. For example, a function pointer in the stack frame can be overwritten.

Canaries can also be bypassed by overflowing a pointer used as the destination of a `strcpy()`-like function. Some stack-smashing techniques can overwrite the return address without touching the canary (note that the XOR canary was introduced to protect against this attack). Also, enabling canary protection requires recompiling the program, which can be expensive.

2.3.4 Non-Executable Memory

In order to prevent attacker-supplied code from executing, the operating system can mark the stack and heap as non-executable.

> **Remark.** Linux leverages the NX (No-eXecute) bit in the page table entry to mark certain memory areas as non-executable. Windows implements this mechanism using Data Execution Prevention (DEP), which can emulate the mechanism if the NX bit is not supported by the processor. OpenBSD implements W $\oplus$ X, which enforces that a page is either writable or executable, but not both.

These mechanisms aim to ensure that no memory region is both writable and executable. However, some applications may require writable memory that is also executable. In addition, attackers can still hijack the control flow of a vulnerable application even if the memory is non-executable, for example, via return-to-libc or return-oriented programming (ROP).

If the memory is non-executable, attackers cannot launch code-injection attacks. However, they can still hijack the control flow by using existing functions or code snippets in libraries and chaining them together to perform the desired tasks.

2.3.5 Address Space Layout Randomization (ASLR)

This defense maps the stack, heap, code, and shared libraries to random locations in the process memory. The library code needs to be position-independent; otherwise, runtime mapping may incur overhead. This makes jumping to a library function more difficult.

In Linux, ASLR is enabled by default. It is implemented by the kernel in collaboration with the ELF loader.

To exploit ASLR, brute-forcing can be used if the entropy is low, i.e. if it is possible to limit the variation space.

> **Remark.** In security, entropy refers to how many possible values something can take. High entropy means many possibilities, making it hard to guess. Low entropy means fewer possibilities, making it easier to guess via brute-force.

2.3.6 Control Flow Integrity

Previously, we discussed the control flow graph (CFG). Programs have a CFG, and control-hijacking attacks usually result in execution paths that do not exist in the CFG. Therefore, control flow integrity (CFI) ensures that the application follows only the control flows specified by the CFG.

To enforce CFI, the CFG is built at compile time, and the application is instrumented to check at runtime the validity of each control transfer.

For example, in Windows 10, Control Flow Guard (CFG) prevents jumping to invalid locations through indirect calls by checking against a bitmap of all valid function entry points in the executable. However, it does not prevent an attacker from jumping to a valid but unintended function, so it is difficult to statically build a fully accurate CFG.

Chapter 3

Cryptography

Cryptography is a powerful tool that enables secure communication over insecure communication channels. It forms the basis for many security mechanisms, e.g. online browsing. It is not a solution to all security problems, but it is reliable if implemented and used properly.

Cryptography provides mechanisms to achieve these goals:

1. **Confidentiality**: preventing adversaries from learning the content of our private data

2. **Integrity**: preventing adversaries from altering our data

3. **Authenticity**: ensuring the data is created by the expected user

3.1 Basic Concepts

3.1.1 Terminologies

1. **Encryption**: converting plaintext into ciphertext

2. **Decryption**: converting ciphertext into plaintext

3. **Key**: a randomly chosen value used in encryption and decryption

4. **Cryptanalysis**: breaking the encryption/decryption scheme by analyzing the algorithm or implementation

3.1.2 Types of Cryptographic Algorithms

1. **Symmetric Algorithms**: use a single shared secret key

2. **Asymmetric Algorithms**: use a pair of keys (public/private). It is computationally impractical to derive one key from the other.

3.1.3 Cryptographic Functions

A **one-way function** is easy to compute but difficult to invert.

A **cryptographic hash function** produces a fixed-length output and satisfies:

- pre-image resistance

- second pre-image resistance

- collision resistance

Cryptographic algorithms can be computationally secure, since the cost of breaking the cipher is unacceptably high and the time required to break it exceeds the useful lifetime of the encrypted information.

3.2 Threat Model, Attacks and Security

3.2.1 Threat Model and Attacks

Before constructing secure systems, we define attacker capabilities:

1. **Ciphertext-Only Attack (COA)**: attacker sees ciphertexts only
2. **Known Plaintext Attack (KPA)**: attacker knows some plaintext-ciphertext pairs
3. **Chosen Plaintext Attack (CPA)**: attacker can choose plaintexts to be encrypted
4. **Chosen Ciphertext Attack (CCA)**: attacker can choose ciphertexts to be decrypted

A secure scheme must remain secure even under strong attack models (e.g. CPA).

3.2.2 Security Definitions

Pseudo-Random Permutation (PRP)

Encryption should behave like a random permutation to any efficient attacker.

> **Intuition.** The attacker can query inputs and observe the outputs from two boxes, one implementing E_K and one implementing a random permutation. Although the same input always produces the same output, the attacker cannot find any pattern to distinguish which box is the real encryption. Hence, the cipher behaves like a random shuffling, achieving PRP security.

We say an encryption scheme is PRP secure if for any polynomial-time attacker, the probability of winning this security game is not significantly greater than 50%, i.e.,

$$\mathbb{P}[\text{Attacker wins}] \leq \frac{1}{2} + \varepsilon \quad (\text{e.g., } \varepsilon = \frac{1}{2^{128}})$$

However, the attacker may still learn whether two plaintexts are equal, since $E_K(M_i) = E_K(M_j)$ implies $M_i = M_j$, even though the actual values of M_i and M_j remain unknown.

Indistinguishability under Known Plaintext Attack (IND-KPA)

Even with known plaintexts, attacker cannot distinguish encryptions.

> **Intuition.** The attacker has access to some plaintext–ciphertext pairs from previous observations. Even with this knowledge, they should not be able to gain any additional information about new ciphertexts or recover the underlying plaintexts. Thus, past knowledge does not help the attacker break the encryption, which corresponds to IND-KPA security.

An encryption scheme is IND-KPA secure if and only if, for any polynomial-time attacker, the probability of winning the IND-KPA game is not significantly greater than 50%, i.e.,

$$\mathbb{P}[\text{Attacker wins}] \leq \frac{1}{2} + \varepsilon \quad (\text{e.g., } \varepsilon = \frac{1}{2^{128}})$$

Indistinguishability under Chosen Plaintext Attack (IND-CPA)

Even if attacker can choose plaintexts, they cannot distinguish which message was encrypted.

> **Intuition.** The attacker can choose arbitrary plaintexts and obtain their corresponding ciphertexts. Despite this strong capability, when given a ciphertext of one of two chosen messages, the attacker still cannot determine which message was encrypted with probability better than random guessing. This means the encryption reveals no useful information even under active attack, achieving IND-CPA security.

An encryption scheme is IND-CPA secure if and only if, for any polynomial-time attacker, the probability of winning the IND-CPA game is not significantly greater than 50%, i.e.,

$$\mathbb{P}[\text{Attacker wins}] \leq \frac{1}{2} + \varepsilon \quad (\text{e.g., } \varepsilon = \frac{1}{2^{128}})$$

Key insight: IND-CPA requires **randomized encryption**, so for the same plaintext it outputs a different ciphertext each time.

3.3 Symmetric Key Cryptography

In symmetric key encryption, a random nonce[1] is sometimes used in the cipher.

3.3.1 Basic Model

Let message M, key K, ciphertext C:

$$C = E_K(M), \quad M = D_K(C)$$

Here, the goal is first to ensure confidentiality. There are three algorithms involved:

1. `KeyGen()` $\rightarrow K$
2. $E(M, K) = E_K(M) = C$
3. $D(C, K) = D_K(C) = M$

Correctness requires:

$$D_K(E_K(M)) = M$$

Note that ciphertext should reveal **no information** about plaintext (except length).

A secure scheme ensures:

$$\mathbb{P}(\text{guessing any bit correctly}) = 0.5$$

Any advantage > 0.5 implies information leakage.

> **Remark.** The 'slightly better than 50%' matters because, for a secure encryption scheme, every bit should appear random. Thus, an attacker guessing a bit should have
>
> $$\mathbb{P}(\text{correct}) = 0.5.$$
>
> If the probability is greater than 0.5, the encryption leaks information.

3.3.2 Simple Cipher

A simple cipher is the Caesar cipher, e.g.

$$E(M, K) = M + K \mod 26$$

This cipher is insecure due to:

- brute force
- frequency analysis
- CPA/KPA attacks

According to **Kerckhoffs's Principle**, *security must rely only on secrecy of the key.* It is not sufficient to say that an attacker cannot decrypt the ciphertext; such a system should still be considered insecure.

[1] A cryptographic nonce is an arbitrary, unique number employed in cryptographic protocols to ensure communication freshness and security.

3.3.3 One-Time Pad (OTP)

A random key is chosen for each message M, and the key is used only once. The key is as long as the message. Then we use $E(M, K) = M \oplus K$ and $D(C, K) = C \oplus K = (M \oplus K) \oplus K = M$.

OTP is secure against one-time eavesdropping, and the ciphertext reveals no information about the plaintext except for the length.

To prove OTP is IND-KPA secure, we assume the adversary knows some pairs $(M_0, C_0), (M_1, C_1), \ldots$. Since each encryption uses a fresh independent key, there is no relation between the old and new pairs. So for any M_0, M_1,

$$\mathbb{P}(C^* \mid M_0) = \mathbb{P}(C^* \mid M_1)$$

so

$$\mathbb{P}(M_0 \mid C^*) = \mathbb{P}(M_1 \mid C^*)$$

To prove it is IND-CPA secure, since the attacker can query $E(M_0)$, $E(M_1)$, we obtain a challenge ciphertext $C^* = E(M_b)$. For each encryption, a new random key is used, so the ciphertext looks completely random every time, giving $\mathbb{P}(\text{correct}) = \frac{1}{2}$.

However, there are several problems regarding the one-time pad.

1. Key Generation: it needs truly random and independent keys.

2. Key Distribution: it needs a secure way to distribute the random key for each communication; however, if the key can already be shared securely, such a method could be used directly to send the secret message.

3. Key Length: the key needs to be as long as the message.

3.4 More on Symmetric Key Cryptography

3.4.1 Block Cipher

Consider the building blocks as follows:

1. A function $E : \{0,1\}^n \times \{0,1\}^k \to \{0,1\}^n$

2. Fixing the key K: $E_K : \{0,1\}^n \to \{0,1\}^n$

3. Encryption $E_K(M) = E(M, K)$

with the following properties:

1. Correctness: E_K is a permutation (bijective)

2. Efficiency: Computation is very fast

3. Security: the permutation looks random without knowing K

For an unknown key, E_K behaves like a random permutation (PR). The attacker then cannot learn the plaintext given only the ciphertext, and thus block ciphers provide some confidentiality. However, with the same key, a block cipher produces the same output for the same input; thus, this deterministic behavior makes it not IND-CPA secure.

3.4.2 Modes of Operation

To ensure that we can encrypt messages of arbitrary lengths and achieve IND-CPA security even if the same key is reused, we apply different modes of operation.

Similar to OTP, a block cipher works on fixed-length input. If the message is shorter than the block length, padding can be added. If the message is longer, the block cipher can be applied repeatedly. This is called a block cipher mode, a rule for how to use a block cipher to encrypt long messages securely.

Chaining block ciphers in certain modes of operation invokes the block cipher multiple times on inputs related to other blocks. This requires some initial randomness, i.e. an Initialization Vector (IV). This

prevents the encryption scheme from being deterministic.

3.4.3 Electronic Code Book (ECB) Mode

This is the simplest mode. It splits a long message into n-bit blocks $M_1, M_2, \ldots$. Each block is encrypted or decrypted independently using the same key, i.e. $C_i = E_K(M_i), M_i = D_K(C_i)$. The output is produced by concatenating the encrypted or decrypted blocks (Figure 3.1).

However, this mode is insecure because it leaks patterns (Figure 3.2).

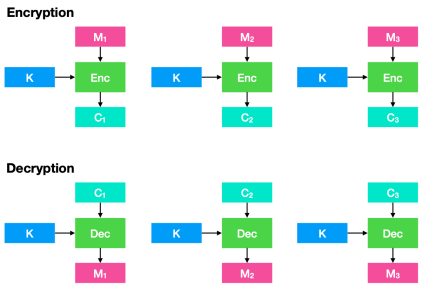

Figure 3.1: ECB Mode

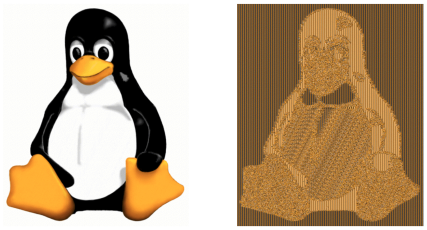

Figure 3.2: Encrypted Image by ECB Mode

3.4.4 Cipher Block Chaining (CBC) Mode

A better mode is to include a varying and known nonce. Again, a long message is split into blocks. Then a one-time random IV is chosen, and the block ciphers are chained by including the output of the previous block as input to the current block. The final output is the IV concatenated with all ciphertext blocks (Figure 3.3).

This can achieve IND-CPA security, and thus it is popular and still widely used. However, it uses sequential encryption and is hard to parallelize.

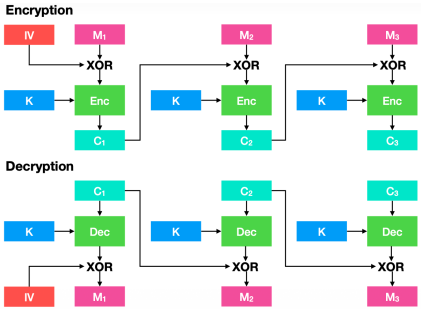

Figure 3.3: CBC Mode

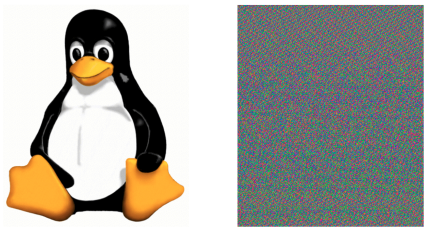

Figure 3.4: Encrypted Image by CBC Mode

3.4.5 Counter (CTR) Mode

For CTR mode, it supports parallel encryption and decryption, with provable security. This mode also selects a random IV (or nonce), and then the counter value for each block is incremented.

Note that CTR decryption uses the block cipher's encryption function, not the decryption function.

Both CTR and CBC achieve IND-CPA security if there is no reuse of the IV. Also, both modes require roughly the same amount of computation. However, CBC is parallelizable only for decryption, while CTR is parallelizable for both encryption and decryption.

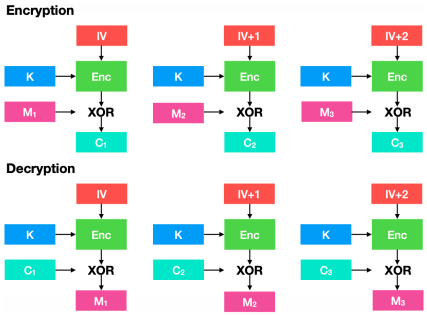

Figure 3.5: CTR Mode

3.4.6 Stream Ciphers

Block ciphers are fixed-size, stateless, and require modes to securely process longer messages. However, a stream cipher keeps state from previous messages and generates a one-time pseudo-random keystream K.

For a Pseudo-Random Number Generator (PRNG), given a seed, it outputs a sequence of seemingly random numbers and arbitrarily many random bits, while maintaining internal state. However, it cannot be truly random; instead, a cryptographically strong PRNG appears truly random to an attacker, since they cannot distinguish its output from a truly random sequence.

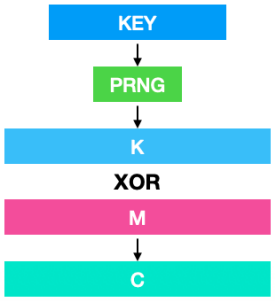

Figure 3.6: Stream Ciphers

Thus, the key is used together with a state. Using a PRNG, we generate a keystream K that is used to encrypt the message. After choosing a random IV (initial state), we have

$$E(M, K) = \text{PRNG}(K, IV) \oplus M,$$

and

$$D(C, K) = \text{PRNG}(K, IV) \oplus C.$$

This is similar to the OTP, and it can produce any number of pseudo-random bits. It can also encrypt multiple messages.

> **Remark.** Block ciphers are stateless, meaning each encryption is independent and does not depend on previous inputs, so modes are needed to securely handle long messages. In contrast, stream ciphers maintain internal state and generate a pseudo-random keystream sequentially.

After confidentiality, we also need to defend against active attacks.

3.4.7 Integrity and Authentication

Confidentiality alone is insufficient against active attacks. For an active attacker, they can inject new messages (ciphertexts), replay previous ciphertexts, and cause messages to be reordered or discarded. Therefore, for a message, we need to prove that it is not altered and is indeed sent from the sender. The approach to provide integrity is to use cryptographically strong hash functions.

There are properties that need to be satisfied:

1. One-way (pre-image resistance): ensure that given y, it is difficult to find x such that $\text{Hash}(x) = y$.
2. Collision resistance: ensure it is difficult to find x and x' such that $\text{Hash}(x) = \text{Hash}(x')$.

3. Second pre-image resistance: ensure that given x, it is difficult to find x' such that $\text{Hash}(x) = \text{Hash}(x')$.

For example, SHA-256 and SHA-384 are two variants of the SHA-2 hash algorithm, returning 256-bit or 384-bit hashes. However, they are vulnerable to length-extension attacks. For example, suppose S is a secret, and the attacker knows $\text{len}(S)$ and $\text{Hash}(S)$. Then they can compute $\text{Hash}(S \,\|\, M)$ for some M, as the internal state can be derived from $\text{Hash}(S)$ and $\text{len}(S)$.

Since cryptographically strong hash functions alone are not sufficient for authentication (because attackers can also compute valid hash values), we introduce the Message Authentication Code (MAC).

3.4.8 Message Authentication Code (MAC)

This provides message integrity and authentication. It does not provide confidentiality and requires a shared secret key K; the message itself can be in plaintext.

For example, sender A computes $T = \text{MAC}(M, K)$, then sends $\{M, T\}$. Receiver B receives $\{M', T'\}$, computes $T'' = \text{MAC}(M', K)$, and compares T' with T''.

Here, integrity is provided if the attacker cannot find M^* and T^* such that $T^* = \text{MAC}(M^*, K)$, or find M^* such that $T = \text{MAC}(M^*, K)$. This requires a combination of a strong hash function and a shared secret key.

3.4.9 Hash-MAC (HMAC)

To build a secure MAC, we consider the following components:

1. H, which is a cryptographically strong hash function
2. $\text{pad}_i, \text{pad}_o$, which are publicly known strings
3. K, a secret key

Then we have

$$\text{HMAC}(M, K) = H\big((K \oplus \text{pad}_o) \,\|\, H((K \oplus \text{pad}_i) \,\|\, M)\big)$$

which is believed to be secure even if H has certain weaknesses.

MACs in general do not provide any guarantee on confidentiality of the message, so it is recommended to compute the MAC using a secret key and apply it properly with encryption. For example, consider K_E as the encryption key and K_I as the MAC key:

1. SSL (MAC-then-Encrypt): $C = E(M \,\|\, \text{MAC}(M, K_I), K_E)$
2. SSH (Encrypt-and-MAC): $C = E(M, K_E) \,\|\, \text{MAC}(M, K_I)$
3. IPsec (Encrypt-then-MAC): $C = E(M, K_E) \,\|\, \text{MAC}(E(M, K_E), K_I)$

In summary, symmetric cryptography provides strong confidentiality but weak integrity and authenticity. Therefore, we introduce asymmetric cryptography.

3.5 Asymmetric Key Cryptography

To solve the problem of secret key management and distribution in symmetric encryption, we consider asymmetric cryptography.

3.5.1 Basic Model

In asymmetric cryptography, each party has a pair of keys $\{PK, SK\}$. The public key is known to everyone, while the private key is kept secret.

A message from A to B is encrypted using B's public key, and B uses the private key to decrypt it.

Let message M, public key PK, and private key SK:

$$C_i = E(M_i, PK_B), \quad M_i = D(C_i, SK_B)$$

This model relies on the following building blocks:

1. **Key generation algorithm**: the generated public key cannot be used to easily compute the secret key, and vice versa.

2. **Trapdoor permutation**: a one-way permutation that is easy to invert with special (secret) information.

3. **Encryption/Decryption**:

$$C = F(PK, M), \quad F^{-1}(SK, C) = M$$

3.5.2 Euler's Totient Function

A one-way function f is a function such that, given x, it is easy to compute $f(x)$, but it is difficult to find any x for a given y such that $f(x) = y$. The difficulty is measured with respect to polynomial-time computation.

A key tool in RSA is Euler's totient function $\phi(n)$, which counts the number of positive integers k that are relatively prime to n, i.e. $\gcd(k, n) = 1$.

Important properties:

$$\phi(p) = p - 1 \quad \text{if } p \text{ is prime}$$

$$\phi(pq) = \phi(p)\phi(q) \quad \text{if } p, q \text{ are relatively prime}$$

Euler's theorem states that if a and n are co-prime:

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

This property forms the foundation of RSA.

To find a large prime p:

1. Generate a large random number p.

2. Choose random a, $1 < a < p - 1$.

3. Test whether $a^{p-1} \bmod p = 1$.

If not, find another p. Otherwise, repeat with different a. Some composite numbers (Carmichael numbers) may pass, but they are rare.

3.5.3 RSA Crypto-System

RSA Key Generation

This key generation algorithm follows these steps:

1. Find large prime numbers p, q.

2. Compute $n = p \cdot q$.

3. Compute $\phi(n) = (p - 1)(q - 1)$.

Note that computing $\phi(n)$ from n alone is difficult.

4. Choose an integer e such that $\gcd(e, \phi(n)) = 1$ and $2 < e < \phi(n)$.

5. Compute $d = e^{-1} \bmod \phi(n)$, which is the multiplicative inverse of e modulo $\phi(n)$.

$$\text{PK} = \{n, e\}, \quad \text{SK} = \{d\}.$$

Encryption and Decryption

For $M < n$:

$$C = E(M, PK) = E_{\{n,e\}}(M) = M^e \mod n$$

$$D(C, SK) = D_{\{d\}}(C) = C^d \mod n = M^{ed} \mod n = (M^{ed-1})M \mod n$$

Since $e \cdot d = 1 \mod f(n)$ gives $e \cdot d - 1 = k \cdot f(n)$ for some k, then

$$\begin{aligned}(M^{ed-1})M \mod n &= (M^{kf(n)})M \mod n \\ &= ((M^{f(n)})^k)M \mod n \\ &= (1^k)M \mod n \\ &= M \mod n \\ &= M\end{aligned}$$

Remark. If $M \geq n$, we typically encrypt a symmetric key using RSA and use symmetric encryption for the message.

However, this scheme is deterministic and not IND-CPA secure.

3.5.4 Diffie-Hellman Key Exchange

Diffie-Hellman is used for secure key exchange.

Two parties agree on a prime p and base (generator) g, where g is a primitive root modulo p[2].

- A chooses secret a, sends $A = g^a \mod p$

- B chooses secret b, sends $B = g^b \mod p$

Both compute:

$$s = g^{ab} \mod p$$

The computation of modular exponentiation can be done efficiently even for large numbers. However, given only g, p, and $A = g^a \mod p$, it is very difficult to find a if p is very large (Discrete Logarithm Problem). This makes it infeasible for an attacker to learn the secret even with the fastest computers.

$s = g^{ab} \mod p$ is used as the basis for a session key. When the communication stops, the secrets are discarded, providing forward secrecy. They also do not retain any information that later attackers could use to decrypt messages exchanged using s.

MITM Attacks

An active attacker can replace A, B with A', B', establishing separate keys with both parties. Thus, Diffie-Hellman alone does not provide authentication.

This can be prevented using digital signatures and certificates.

3.5.5 Digital Signatures

To bind a document to its author, the signature must depend on the document.

$$S = \text{sign}(M, SK) = F^{-1}(H(M), SK)$$

$$\text{verify}(M, S, PK) = \text{accept if } F(S, PK) = H(M)$$

In practice, A will generate a public-private key pair $\{n, e\}$ and $\{d\}$, then sign a message M where

$$S = \text{sign}_{\{d\}}(M) = RSA_{\{d\}}(H(M)) = H(M)^d \mod n$$

This can be validated using

$$\text{verify}_{\{n,e\}}(M, S) = \text{true} \Leftrightarrow H(M) = S^e \mod n$$

This provides authentication and non-repudiation.

[2] A primitive root $g \mod p$ is a number that can generate all possible non-zero values mod p by exponentiation.

3.6 A Better Asymmetric Key Encryption

Asymmetric encryption alone ensures confidentiality, but not authentication.

A common approach:

- Sign with sender's private key

- Encrypt with receiver's public key

This provides confidentiality, integrity, authentication, and non-repudiation. However, it is still vulnerable to MITM attacks during key exchange.

MITM attacks occur when an attacker replaces public keys. This is solved using certificates, which bind a public key to an identity.

A Certificate Authority (CA) verifies identities and signs certificates:

$$\text{Cert} = \text{Sign}_{SK_{CA}}(\text{identity}, PK)$$

The CA publishes its own public key, allowing anyone to verify certificates.

Chapter 4

OS Security

Operating system security aims to protect system resources while ensuring correct and controlled access. The fundamental goals remain the same: confidentiality, integrity, and authentication. Since many system resources are sensitive, mechanisms are required to prevent unauthorized access and misuse.

4.1 Access Control

4.1.1 Fundamentals

To enforce security goals, we first need a way to regulate how resources are accessed. Access control is the selective restriction of access to resources according to a policy.

The key components are:

- subject (principal): the entity requesting access to resources

- object: a resource that can be accessed

- policy: rules that define how subjects can access objects

- authentication: identifying who is making the request

- authorization: determining what actions the subject is allowed to perform

- audit: logging actions performed by subjects

- accountability: holding subjects responsible for their actions

We formalize access decisions as:

$$\text{access}(\text{Subject}, \text{Object}) = \text{true/false}$$

4.1.2 Access Control Matrix

A natural way to represent access control policies is through the access control matrix, which maps subjects to objects and specifies allowed actions.

	`/user/alice`	`/user/bob`	`/etc/password`	`/share/bob/b.txt`	...
Alice	True	False	False	True	
Bob	False	True	False	True	
root	True	True	True	True	
...					

Table 4.1: Access Control Matrix

In practice, permissions are not simply binary. Instead, we use finer-grained access rights such as Read (R), Write (W), and Execute (X).

	`/user/alice`	`/user/bob`	`/etc/password`	`/share/bob/b.txt`	...
Alice	{R, W}	{}	{}	{R}	
Bob	{}	{R, W}	{}	{R, W}	
root	{R, W}	{R, W}	{R, W}	{R, W}	
...					

Table 4.2: Access Control Matrix

However, directly storing this matrix is inefficient in real systems. This motivates practical mechanisms for implementing access control.

4.2 Security Mechanisms

There are two primary ways to implement the access control matrix:

1. Access Control List (ACL)
2. Capability-based Security

4.2.1 Access Control List

ACL is an object-centric mechanism. Each object maintains a list specifying which subjects have what permissions.

When access is requested:

- the system identifies the subject,
- looks up the subject in the object's ACL,
- and determines whether the requested action is allowed.

4.2.2 Capability-based Security

Capability-based security is a subject-centric mechanism. Each subject holds capabilities, which are unforgeable tokens that reference objects along with permitted operations.

When access is requested:

- the subject presents a capability,
- the system verifies the capability,
- and grants access if the required permission is included.

4.2.3 Reference Monitor

Regardless of the mechanism used, all access decisions must be enforced by a reference monitor.

A reference monitor mediates every access to objects and must satisfy:

1. unbypassable
2. tamper-proof
3. verifiable

Thus, subjects cannot access objects directly; all requests must pass through the reference monitor.

4.2.4 Trusted Computing Base

The Trusted Computing Base (TCB) consists of all components that must be correct to enforce security. Since all trust depends on the TCB, it should be:

- minimal

- unbypassable

- tamper-proof

- verifiable

4.2.5 DAC vs. MAC

Different systems adopt different policy models.

- Discretionary Access Control (DAC): object owners define access policies.

- Mandatory Access Control (MAC): a centralized authority enforces system-wide policies.

4.2.6 Attribute-Based Access Control

Attribute-Based Access Control (ABAC) generalizes access control by assigning attributes to subjects and objects. Policies are defined based on these attributes.

Role-Based Access Control (RBAC) can be viewed as a special case of ABAC.

4.3 UNIX Security Model

We now examine how these concepts are applied in a real system.

In UNIX:

1. subjects: users

2. objects: files, directories, devices, sockets, etc.

3. operations: read, write, execute

Users may belong to multiple groups, enabling group-based (role-like) access control.

4.3.1 File Access Control

Each file has an owner and a simple ACL based on:

1. owner

2. group

3. others

Permissions are set by the owner or root.

> **Remark.** If a user has write permission on a directory but not on a file, the ability to rename or delete the file depends on the sticky bit.
>
> - If the sticky bit is off: users can delete or rename files they do not own.
>
> - If the sticky bit is on: only the file owner, directory owner, or root can do so.

Each process maintains:

1. real UID/GID: identity of the process owner

2. effective UID/GID: used for permission checks

3. saved UID/GID: allows privilege dropping and restoration

If an executable has the SUID bit set, executing it temporarily grants the process the privileges of the file owner.

4.4 Privilege and Isolation

Traditional UNIX distinguishes between:

- privileged processes (e.g., root)

- unprivileged processes

To reduce risk, modern systems decompose root privileges into smaller capabilities.

To limit damage from compromise:

- minimize the TCB

- enforce the principle of least privilege

- isolate untrusted components

4.4.1 Sandbox

A sandbox isolates a process from the rest of the system. If some vulnerable code is from an untrusted source or takes input from potential attackers, running it in a sandbox aims to prevent exploits from spreading outside of the sandbox.

It restricts access to:

- file system

- network

- system resources

This prevents exploits from affecting the broader system.

4.4.2 Android Process Isolation

Android enforces sandboxing as follows:

1. Isolation: each app runs with a unique UID in its own VM

2. Interaction: permissions checked via a reference monitor

3. Least privilege: permissions granted explicitly by users

4.4.3 Chrome Security Model

Chrome uses a multi-process architecture:

- a privileged browser process

- multiple sandboxed renderer processes

Renderer processes handle untrusted web content with restricted permissions. If compromised, the damage is contained.

4.4.4 Information Flow Tracking

Information flow tracking monitors how data propagates through a system. Objects are labeled with security tags, which are updated during computation. This helps enforce confidentiality and integrity constraints.

Chapter 5

Security Principles

To design secure software and systems, we follow a set of fundamental security principles. These principles guide how systems should be built, how access should be controlled, and how users interact with security mechanisms.

Understanding Threat Model

To defend against attacks, we must first understand what we are protecting and what the potential threats are. Therefore, threat modeling should be performed both before and after deploying any security mechanisms. This ensures that defenses remain effective as the system evolves.

Economy of Security Mechanism

There is often a tradeoff between the cost of security and the level of protection provided. The goal is to increase the cost or difficulty for an attacker such that the reward does not justify the effort. At the same time, defenders must consider the cost of implementing and maintaining security mechanisms.

Open Design

Security should not rely on obscurity. As stated in Kerckhoffs's principle: "A crypto-system should be secure even if everything about the system, except the key, is public knowledge," and Shannon's maxim: "the enemy will immediately gain full familiarity with the system."

It is easier to protect and revoke a key than to keep a system design secret. Open design allows public scrutiny, which helps identify flaws in both design and implementation.

Fail-safe Defaults

Access decisions should be based on permission rather than exclusion. Without explicit permission, the default action is to deny access. Mechanisms that attempt to identify only unsafe conditions are less reliable and not recommended.

Least Privilege

Each system module should have only the minimum privileges necessary for its intended function. This principle relies on compartmentalization and isolation, where the system is divided into separate components with limited access rights.

Separation of Privilege

When possible, multiple independent privileges should be required to perform sensitive actions. This reduces the risk that a single compromised component can lead to a full system breach.

Defense in Depth

Security should not rely on a single mechanism. Instead, multiple layers of protection should be used so that even if one fails, others remain effective. These mechanisms should be diverse to avoid common points of failure.

For example, two-factor authentication combines:

- something you know (e.g. password),

- something you have (e.g. security token),

- something you are (e.g. biometrics).

Monolithic Design

A monolithic design builds the entire application as a single unified program. It is simple to develop, test, and deploy, and does not require communication between components. However, a single successful attack can compromise the entire system, since components are not isolated.

Component Design

In component-based design, the system is divided into separate modules with well-defined interfaces. This improves structure and reduces accidental vulnerabilities. However, since components still reside within the same application, a compromise may still propagate across components.

Complete Mediation

Every access to every resource must be checked for authorization. This is typically enforced using a reference monitor, through which all access requests must pass.

Human Factors

Security mechanisms must account for human limitations. Since users cannot reliably store high-quality cryptographic keys, systems rely on tools to securely manage credentials.

Two important considerations are:

Psychological acceptability: systems should be easy to use securely, so users do not bypass or misuse security mechanisms.

Human factor assumptions: security mechanisms should not rely on unrealistic assumptions about user behavior, even when users are not intentionally acting against the system.

Chapter 6

Web Security

6.1 Fundamentals of Web

Code	Status	Description
1xx	Informational	The request was received, continuing process
200	OK	Success
301	Moved Permanently	Permanent redirection
307	Temporary Redirect	Try URI in the Location field this time
400	Bad Request	Malformed request
401	Unauthorized	Require user authentication
403	Forbidden	Refuse to fulfill the request
404	Not Found	
500	Internal Server Error	Something unexpected (Error) happened
501	Not Implemented	Does not support the requested functionality
503	Service Unavailable	Unable to handle the request due to overloading or maintenance

Table 6.1: Common HTTP Response Status Code

6.2 Same-Origin Policy

6.3 Session

6.4 Injection

6.5 XSS, UI-based Attacks

Chapter 7

Network Security

7.1 Fundamentals

7.2 Local Area Network Attacks