Computer Security

Christian Rossi

Academic Year 2023-2024

Abstract

This course provides a comprehensive introduction to information security, beginning with foundational concepts in understanding what information security is, alongside definitions of vulnerabilities, risks, exploits, and attackers, with a focus on managing security as a risk-based practice.

The course then covers cryptography, providing a brief history and discussing paradigm shifts, perfect and computational confidentiality, data integrity, Message Authentication Codes (MACs), cryptographic hash functions, and asymmetric cryptographic tools like key agreement, key exchange, and digital signatures. Students will explore Public Key Infrastructure (PKI) and critically assess digital signature schemes and PKI engineering pitfalls.

Authentication is discussed in terms of the three factors of authentication, multifactor methods, and evaluation of authentication technologies, including ways to bypass these controls. This leads into authorization and access control, covering discretionary (DAC) and mandatory (MAC) access control, multilevel security, and applications in sensitive contexts like military information management.

Next, the course examines software vulnerabilities, addressing design, implementation, and configuration errors, memory issues like buffer overflows, web application security, and code-injection vulnerabilities such as cross-site scripting and SQL injections.

The course also addresses secure networking architectures, exploring network protocol attacks, firewall technologies, secure network setups (like DMZ and multi-zone networks), VPNs, and secure protocols like SSL/TLS.

Finally, malicious software and its evolution are discussed, from early threats like the Morris worm to modern malware, botnets, and the underground economy. Techniques in malware analysis, antimalware strategies, and rootkit detection and mitigation are also covered.

Contents

CHAPTER 1

Introduction

1.1 Basic security requirements

The fundamental security principles, known as the CIA paradigm for information security, outline three key requirements:

- *Confidentiality*: only authorized entities can access information.
- *Integrity*: information can only be modified by authorized entities in authorized ways.
- *Availability*: information must be accessible to all authorized parties within specified time limits.

It's worth noting that the availability requirement can sometimes conflict with the other two, as higher availability exposes the system for longer durations.

1.2 Definitions

Definition (*Vulnerability*)**.** A vulnerability is a flaw that can be exploited to violate one of the constraints of the CIA paradigm.

Definition (*Exploit*)**.** An exploit is a specific method of leveraging one or more vulnerabilities to achieve a particular objective that breaches the constraints.

Definition (*Asset*)**.** An asset is anything of value to an organization.

Definition (*Threat*)**.** A threat is a potential event that could lead to a violation of the CIA paradigm.

Definition (*Attack*)**.** An attack is a deliberate use of one or more exploits with the aim of compromising a system's CIA.

Definition (*Threat agent*)**.** A threat agent is any entity or factor capable of causing an attack.

Definition (*Hacker*)**.** A hacker is an individual with advanced knowledge of computers and networks, driven by a strong curiosity and desire to learn.

Definition (*Black hats*)**.** Malicious hackers are commonly referred to as black hats.

1.3 Ethical hacking

White hats, also known as security professionals or ethical hackers, are tasked with:

- *Identifying vulnerabilities.*
- *Developing exploits.*
- *Creating attack-detection methods.*
- *Designing countermeasures against attacks.*
- *Engineering security solutions.*

Since no system is invulnerable, it's crucial to assess its risk level. This involves evaluating the potential damage due to vulnerabilities and threats through the concept of risk:

Definition (*Risk*)**.** Risk is a statistical and economic evaluation of potential damage resulting from the presence of vulnerabilities and threats:

$$\text{Risk} = \text{Asset} \times \text{Vulnerabilities} \times \text{Threats}$$

Assets and vulnerabilities can be managed, but threats are independent variables.

To ensure system security, a balance must be struck between cost and reducing vulnerabilities and containing damage. The costs of securing a system can be categorized as direct and indirect. Direct costs include management, operational, and equipment expenses, while indirect costs, which often form the larger portion, stem from:

- *Reduced usability.*
- *Slower performance.*
- *Decreased privacy* (due to security controls).
- *Lower productivity* (as users may be slower).

It's important to note that simply spending more money on security may not always resolve the issue.

In real-world systems, setting boundaries is essential, meaning that a portion of the system must be assumed as secure. These secure parts consist of trusted elements determined by the system developer or maintainer. For example, the level of trust in a particular system can be determined at the software, compiler, or hardware level.

CHAPTER 2

Cryptography

2.1 Introduction

Definition (*Cryptography*)**.** Cryptography refers to the field of study concerned with developing techniques that enable secure communication and data storage in the presence of potential adversaries.

Cryptography offers several essential features, including:

- *Confidentiality*: ensures that data can only be accessed by authorized entities.
- *Integrity/freshness*: detects or prevents tampering or unauthorized replays of data.
- *Authenticity*: certifies the origin of data and verifies its authenticity.
- *Non-repudiation*: ensures that the creator of data cannot deny their responsibility for creating it.
- *Advanced features*: includes capabilities such as proofs of knowledge or computation.

2.1.1 History

Cryptography has a history as ancient as written communication itself, originating primarily for commercial and military purposes. Initially, cryptographic algorithms were devised and executed manually, using pen and paper.

The early approach to cryptography involved a contest of intellect between cryptographers, who devised methods to obscure messages, and cryptanalysts, who sought to break these ciphers.

A significant development occurred in 1553 when Bellaso pioneered the idea of separating the encryption method from the key.

In 1883, Kerchoff formulated six principles for designing robust ciphers:

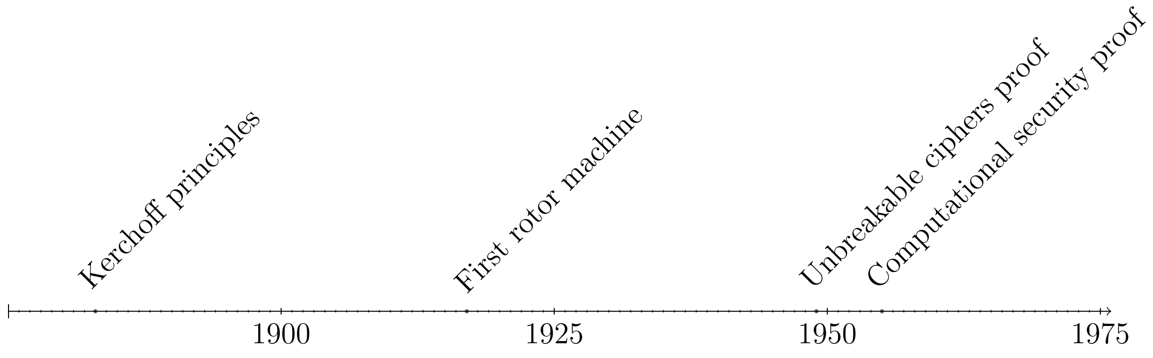
1. The cipher should be practically, if not mathematically, unbreakable.
2. It should be possible to disclose the cipher to the public, including enemies.

3. The key must be communicable without written notes and changeable at the discretion of correspondents.
4. It should be suitable for telegraphic communication.
5. The cipher should be portable and operable by a single person.
6. Considering the operational context, it should be user-friendly, imposing minimal mental burden and requiring a limited set of rules.

The landscape of cryptography underwent a significant transformation in 1917 with the introduction of mechanical computation, exemplified by Hebern's rotor machine, which became commercially available in the 1920s. This technology evolved into the German Enigma machine during World War II, whose encryption methods were eventually deciphered by cryptanalyst at Bletchley Park, contributing significantly to the Allied victory.

After World War II, in 1949 Shannon proved that a mathematically secure ciphers exists.

Following World War II, in 1949, Shannon demonstrated the existence of mathematically secure ciphers. Subsequently, in 1955, Nash proposed the concept of computationally secure ciphers, suggesting that if the interaction of key components in a cipher's determination of ciphertext is sufficiently complex, the effort required for an attacker to break the cipher would grow exponentially with the length of the key ($\mathcal{O}(2^\lambda)$), surpassing the computational capabilities of the key owner ($\mathcal{O}(\lambda^2)$) for sufficiently large key lengths.



2.1.2 Definitions

Definition (*Plaintext space*)**.** A plaintext space P is the set of possible messages $ptx \in P$.

Definition (*Ciphertext space*)**.** A ciphertext space C is the set of possible ciphertext $ctx \in C$.

It's worth noting that the ciphertext space C may have a larger cardinality than the plaintext space P.

Definition (*Key space*)**.** A key space K is the set of possible keys.

The length of the key often correlates with the desired level of security.

Definition (*Encryption function*)**.** An encryption function $\mathbb{E}$ is a mapping that takes an element from the plaintext space P and a key from the key space K, and produces an element from the ciphertext space C:

$$\mathbb{E} : P \times K \to C$$

Definition (*Decryption function*)**.** A decryption function $\mathbb{D}$ is a mapping that takes an element from the ciphertext space C and a key from the key space K, and yields an element from the plaintext space P:

$$\mathbb{D} : C \times K \to P$$

2.2 Computational security

The objective of ensuring confidentiality is to prevent unauthorized individuals from comprehending the data. Various methods can compromise confidentiality:

- Passive interception by an attacker.
- Knowledge of a set of potential plaintexts by the attacker.
- Data manipulation by the attacker to observe the reactions of an entity capable of decryption.

Definition (*Perfect cipher*)**.** In a perfect cipher, for any plaintext ptx in the plaintext space P and any corresponding ciphertext ctx in the ciphertext space C, the probability of the plaintext being sent is equal to the conditional probability of that plaintext given the observed ciphertext:

$$\Pr(ptx_{sent} = ptx) = \Pr(ptx_{sent} = ptx|ctx)$$

In other words, observing a ciphertext $c \in C$ provides no information about the corresponding plaintext it represents.

Theorem 2.2.1 (Shannon 1949)**.** *Any symmetric cipher $\langle P, K, C, \mathbb{E}, \mathbb{D}\rangle$ with $|P| = |K| = |C|$, achieves perfect security if and only if every key is utilized with equal probability $\frac{1}{|K|}$, and each plaintext is uniquely mapped to a ciphertext by a unique key:*

$$\forall\, (ptx, ctx) \in P \times C, \exists! k \in K \text{ such that } \mathbb{E}\,(ptx, k) = ctx$$

Example:
Let's consider P, K, and C as sets of binary strings. The encryption function selects a uniformly random, fresh key k from K each time it's invoked and computes the ciphertext as $ctx = ptx \oplus k$.

Gilbert Vernam patented a telegraphic machine in 1919 that implemented $ctx = ptx \oplus k$ using the Baudot code. Joseph Mauborgne proposed utilizing a random tape containing the key k.

Combining Vernam's encryption machine with Mauborgne's approach results in a perfect cipher implementation.

It's crucial to understand that while a cipher may achieve perfect security, this doesn't necessarily mean it's practical or user-friendly. Managing key material and regularly changing keys can be exceptionally challenging.

In practice, perfect ciphers often face vulnerabilities due to issues such as key theft or reuse. Additionally, the generation of truly random keys has historically been problematic, leading to potential vulnerabilities and breaches.

In practical terms, ensuring the security of a cipher involves ensuring that a successful attack would also require solving a computationally difficult problem efficiently. The most commonly utilized computationally hard problems for ciphers include:

- Solving a generic nonlinear Boolean simultaneous equation set.
- Factoring large integers or finding discrete logarithms.

- Decoding a random code or finding the shortest lattice vector.

These problems cannot be solved faster than exponential time. However, with some hints, they can become easier to solve within polynomial time.

At this juncture, proving computational security involves the following steps:

1. Define the ideal attacker's behavior.
2. Assume a specific computational problem is difficult.
3. Prove that any non-ideal attacker would need to solve the difficult problem.

The attacker is typically represented as a program capable of accessing given libraries that implement the cipher in question. The security property is defined as the ability to respond to a specific query. The attacker succeeds if it breaches the security property more frequently than would be possible through random guessing.

2.3 Pseudo-Random number generators

To expand the key for use in a Vernam cipher with a finite-length key, we require a Pseudo-Random Number Generators (PRNG). We assume that the attacker's computational capability is limited to polynomial computations.

Definition (*Cryptographically Safe Pseudo-Random Number Generators*)**.** A Cryptographically Safe Pseudo-Random Number Generator (CSPRNG) is a deterministic function:

$$\text{PRNG} : \{0,1\}^{\lambda} \to \{0,1\}^{\lambda+I}$$

where I is an expansion factor, such that the output of the PRNG cannot be distinguished from a uniformly random sample $\{0,1\}^{\lambda+I}$ with computational complexity $\mathcal{O}(\text{poly}(\lambda))$.

In practice, CSPRNG are considered as candidates because there is no conclusive evidence supporting the existence of a definitive PRNG function. Demonstrating the existence of a CSPRNG would imply $\mathcal{P} \neq \mathcal{NP}$.

Developing a CSPRNG from scratch is feasible but not the usual approach due to inefficiency. Typically, they are constructed using another fundamental element called Pseudo-Random Permutations (PRP), which are derived from Pseudo-Random Functions (PRF).

To randomly select a function, we start by considering the set:

$$F = \left\{ f : \{0,1\}^{in} \to \{0,1\}^{out}, in, out \in \mathbb{N} \right\}$$

A uniformly randomly sampled $f \xleftarrow{\$} F$ can be represented by a table with 2^{in} entries, each entry being *out* bits wide:

$$|F| = \left(2^{out}\right)^{2^{in}}$$

Example:
For instance, if $in = 2$ and $out = 2$, the function set $F = \{f : \{0,1\}^2 \to \{0,1\}^1\}$ consists of the 16 Boolean functions with two inputs. Each function is represented by a 4-entry truth table. The total number of functions is 16, corresponding to the $2^4 = 16 = (2^1)^{2^2}$ tables.

2.3.1 Pseudo-Random function

Definition (*Pseudo-Random Function*)**.** A Pseudo-Random Function (PRF) is denoted as:

$$prf_{seed} : \{0,1\}^{in} \to \{0,1\}^{out}$$

Where it takes an input and a λ-bit seed.

Consequently, prf_{seed} is entirely determined by the seed value. It cannot be distinguished from a random function:

$$f \in \left\{f : \{0,1\}^{in} \to \{0,1\}^{out}\right\}$$

within polynomial time in λ. In other words, given $a \in \{f : \{0,1\}^{in} \to \{0,1\}^{out}\}$, it is computationally infeasible to determine which of the following is true:

- $a = prf_{seed}(\cdot)$ with seed $\overset{\$}{\leftarrow} \{0,1\}^{\lambda}$.
- $b \overset{\$}{\leftarrow} F$, where $F = \{f : \{0,1\}^{in} \to \{0,1\}^{out}\}$.

2.3.2 Pseudo-Random Permutation

Definition (*Pseudo-Random Permutation*)**.** A Pseudo-Random Permutation (PRP) is a bijective PRF defined as:

$$prf_{seed} : \{0,1\}^{len} \to \{0,1\}^{len}$$

It is characterized solely by its seed value and cannot be distinguished from a random function within polynomial time. This permutation represents a rearrangement of all possible strings of length *len*. In practical terms:

- It operates on a block of bits and yields another block of equal size.
- The output appears unrelated to the input.
- Its behavior is entirely determined by the seed, akin to a key in conventional cryptography.

However, there is no formally proven PRP because its existence would imply $\mathcal{P} \neq \mathcal{NP}$. Construction of such a PRP typically involves three steps:

1. Compute a small bijective Boolean function f with input and key.
2. Compute f again between the previous output and the key.
3. Repeat the second step until satisfaction.

PRP selection Modern PRP often emerge from public competitions, where cryptanalytic techniques help identify and eliminate biases in their outputs, ensuring robust designs.

These PRPs are commonly known as block ciphers. A block cipher is considered broken if it can be distinguished from a PRP with less than 2^{λ} operations. The key length λ is chosen to be sufficiently large to render computing 2^{λ} guesses impractical. For different security levels:

- Legacy-level security typically employs λ around 80.
- For a security duration of five to ten years, λ is set to 128.
- Long-term security requires λ of 256.

2.3.3 Standard block ciphers

The Advanced Encryption Standard (AES) operates on a 128-bit block size and offers three key lengths: 128, 192, and 256 bits. Chosen as a result of a three-year public competition by NIST on February 2, 2000, AES emerged as the preferred standard out of 15 candidates and has since been standardized by ISO.

The predecessor to AES, known as the Data Encryption Standard (DES), was established by NIST in 1977. DES operated with a relatively short 56-bit key length, leading to security concerns. It was bolstered through triple encryption, effectively achieving an equivalent security level of $\lambda = 112$. Although still present in some legacy systems, DES has been officially deprecated.

2.4 Plaintext encryption

Encrypting plaintexts with a length less than or equal to the block size using a block cipher is effective. This method can be expanded by employing multiple blocks with a split-and-encrypt approach, also known as Electronic CodeBook (ECB) mode.

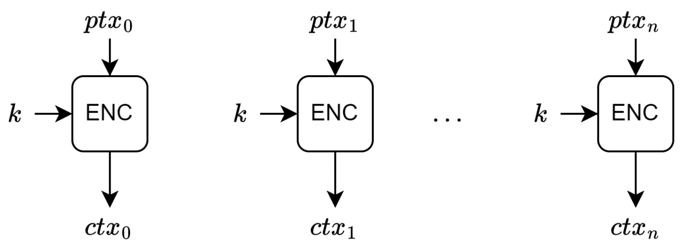


Figure 2.1: ECB encryption mode

However, this technique becomes problematic when there is redundancy within plaintext segments, as the resulting ciphertext may still reveal patterns. This vulnerability arises from the deterministic nature of ECB encryption, where identical plaintext blocks produce identical ciphertext blocks, making it susceptible to certain cryptographic attacks.

To address the issue of pattern visibility in ciphertexts caused by redundancy in plaintext segments, we can employ a counter to differentiate the strings submitted to each block during encryption. This counter, unique for each block, helps mitigate the predictability inherent in traditional encryption modes.

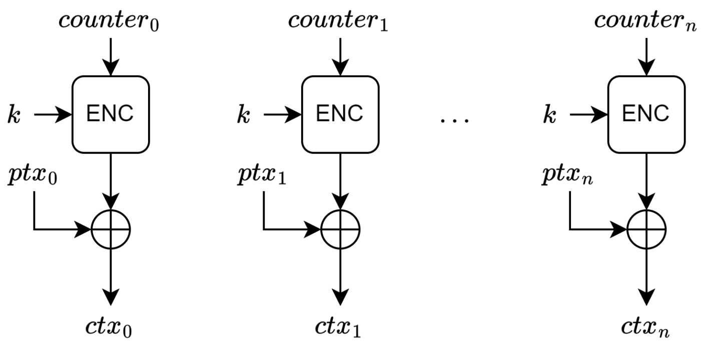


Figure 2.2: CTR encryption mode

This method ensures that even if plaintext blocks are repeated, the resulting ciphertext blocks are different due to the unique counter values assigned to each block.

2.4.1 Chosen-Plaintext Attacks

Now, let's consider a scenario where the attacker has access to both the ciphertext and a portion of the plaintexts. In this type of attack, the attacker is familiar with a series of plaintexts that undergo encryption, and their objective is to determine the specific plaintext being encrypted.

In an ideal situation, the attacker should not be able to distinguish between two plaintexts of equal length when provided with their encrypted versions. Such scenarios frequently occur in contexts like managing data packets within network protocols and discerning between encrypted commands sent to a remote host.

The counter (CTR) mode of operation is vulnerable to Chosen-Plaintext Attacks (CPA) due to its deterministic encryption process. To enhance security and achieve decryptable non-deterministic encryption, we can implement the following steps:

1. *Re-keying*: change the encryption key for each block using a mechanism like a ratchet, ensuring that each block's encryption is independent and unpredictable.

2. *Randomize the encryption*: introduce (removable) randomness into the encryption process by altering the mode of employing PRP. This randomization enhances the unpredictability of the ciphertext, making it more resistant to cryptanalysis.

3. *Nonce usage*: utilize numbers used once (NONCEs) to introduce additional variability into the encryption process. In the case of CTR mode, a NONCE is chosen as the starting point for the counter. This NONCE can be public, adding an extra layer of unpredictability to the encryption.

By implementing these measures, we can significantly enhance the security of the encryption process and mitigate vulnerabilities associated with deterministic encryption modes like CTR.

Re-keying The term ratcheting is derived from the mechanical device called a ratchet, which allows movement in one direction while preventing backward movement. Similarly, in symmetric ratcheting, the encryption keys are ratcheted forward in a manner that prevents an attacker from decrypting past messages even if they compromise the current key.

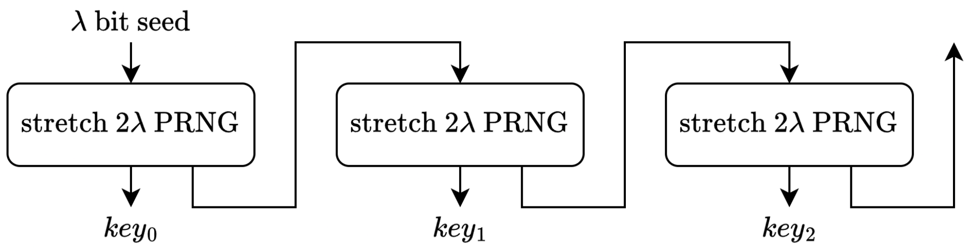


Figure 2.3: Ratcheting

Symmetric ratcheting ensures that even if an attacker manages to compromise the current encryption key, they cannot decrypt past messages or predict future messages due to the frequent key updates. This technique effectively limits the impact of key compromise and strengthens the security of encrypted communication over time.

CPA secure encryption Secure encryption schemes are designed to withstand CPA by ensuring that an attacker cannot gain any useful information about the encryption key or plaintexts, even if they have access to ciphertexts for chosen plaintexts. This is accomplished by utilizing the NONCE in conjunction with the CTR mode of operation.

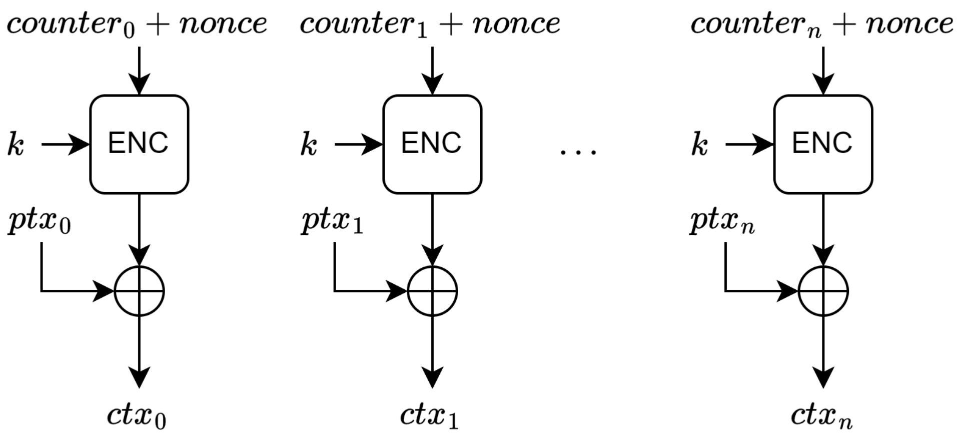


Figure 2.4: Secure ctr

2.5 Data integrity

Malleability refers to the ability to make alterations to the ciphertext, without knowledge of the encryption key, resulting in predictable modifications to the plaintext. This characteristic can be exploited in various ways to launch decryption attacks and manipulate encrypted data. However, malleability can also be leveraged as a desirable feature, as seen in homomorphic encryption schemes.

To mitigate malleability, it is crucial to design encryption schemes that are inherently non-malleable and incorporate mechanisms to ensure data integrity against attackers. While current encryption schemes primarily provide confidentiality, they do not detect changes in the ciphertext effectively.

To address this limitation, a small piece of information known as a tag can be added to the encrypted message, allowing for integrity testing of the encrypted data itself. Simply adding the tag to the plaintext before encryption is not sufficient, as Message Authentication Codes (MAC) are required for proper data authentication.

2.5.1 Message Authentication Codes

A MAC consists of a pair of functions:

- `compute_tag(string, key)`: generates the tag for the input string.
- `verify_tag(string, tag, key)`: verifies the authenticity of the tag for the input string.

In an ideal attacker model, the attacker may possess knowledge of numerous message-tag pairs but should be unable to forge a valid tag for a message they do not already know. Additionally, tag splicing from valid messages should also be prevented.

CBC-MAC Cipher Block Chaining MAC (CBC-MAC) is a method for generating a fixed-size authentication tag from variable-length messages using a block cipher in CBC mode. Here's how CBC-MAC works:

1. *Initialization*: CBC-MAC operates on fixed-size blocks of data, so if the message is not a multiple of the block size, padding is applied to make it fit. The MAC is initialized with a zero or an initial value.
2. *Block Encryption*: the message is divided into blocks of equal size. Each block is encrypted using the block cipher in CBC mode. The ciphertext of each block is then XORed with the next plaintext block before encrypting the next block.
3. *Finalization*: once all blocks are encrypted, the last ciphertext block becomes the MAC.

CBC-MAC possesses several noteworthy characteristics. It is computationally efficient, requiring only a single pass through the message. The MAC generates a fixed-length authentication tag determined by the block size of the underlying block cipher. Additionally, CBC-MAC offers collision resistance, making it extremely difficult to find two different messages that produce the same MAC.

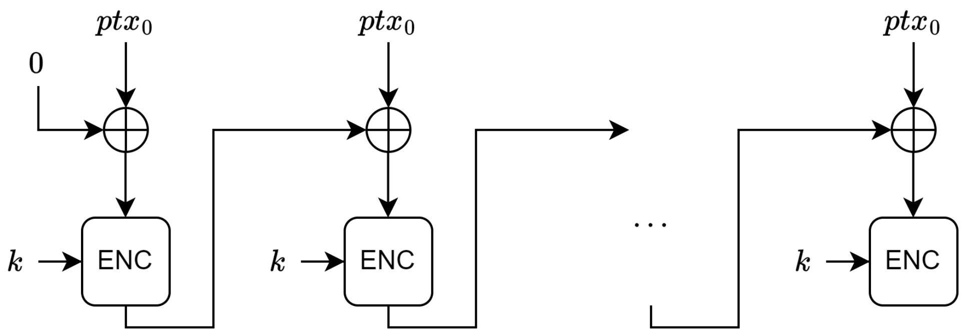


Figure 2.5: CBC encryption mode

CBC-MAC is widely used in practice for message authentication in various cryptographic protocols and applications, including network protocols, file authentication, and secure messaging systems. However, it is important to use CBC-MAC correctly and securely to avoid potential vulnerabilities.

2.5.2 Cryptographic hashes

Ensuring the integrity of a file typically involves either comparing it bit by bit with an intact copy or reading the entire file to compute a message authentication code. However, it would be highly advantageous to verify the integrity of a file using only short, fixed-length strings, regardless of the file's size, thereby simplifying the process and reducing computational overhead. Unfortunately, a significant obstacle arises due to the inherent lower bound on the number of bits required to accurately encode a given content without any loss of information. This limitation presents a challenge when attempting to devise a method for efficiently testing the integrity of files.

A cryptographic hash function, denoted as $H : \{0,1\}^* \rightarrow \{0,1\}^l$, is designed such that the following computational problems are difficult to solve:

1. Given a digest $d = H(s)$, determining the original input s (first preimage).
2. Given both an input s and its corresponding digest $d = H(s)$, finding another input r (where $r \neq s$) that produces the same digest $(H(r) = d)$ (second preimage).
3. Finding two distinct inputs r and s (where $r \neq s$) that yield the same digest $(H(r) = H(s))$ (collision).

In an ideal scenario, the performance of a concrete cryptographic hash function can be summarized as follows:

1. Finding the first preimage requires approximately $O(2^d)$ hash computations, involving guessing potential inputs s.
2. Finding the second preimage similarly demands around $O(2^d)$ hash computations, involving guessing potential inputs r.
3. Discovering a collision involves approximately $O(2^{2d})$ hash computations.

The resulting output bit-string of a hash function is commonly referred to as a digest.

Hash functions For preferred cryptographic hash functions, consider utilizing SHA-2 and SHA-3. SHA-2, developed privately by the NSA, offers digest sizes of 256, 384, and 512 bits. SHA-3, on the other hand, emerged from a public design contest akin to AES and boasts digest sizes ranging from 256 to 512 bits. Both SHA-2 and SHA-3 are currently unbroken and enjoy wide standardization.

Conversely, it's advisable to steer clear of SHA-1 and MD-5. SHA-1, with its fixed 160-bit digest size, has been compromised for collisions, achievable in around 2^{61} operations. MD-5, which is known to be severely broken, allows for collisions with just 2^{11} operations, with public tools readily accessible for generating collisions. MD-5 is particularly vulnerable to collisions with arbitrary input prefixes, achievable in approximately 2^{40} operations.

Usage Hash functions serve various purposes, including:

- *Pseudonymized matching*: employed in scenarios like signal's contact discovery, where hashes of values are stored and compared instead of the actual values themselves.
- *MAC construction*: hash functions are integral in generating MACs, where a tag is produced by hashing both the message and a secret string. Verification involves recomputing the same hash and comparing it with the original tag. Hash-based MAC (HAMC) utilizes a generic hash function as a plug-in, denoted as HMAC-hash name.
- *Forensic applications*: hash functions are crucial in forensic investigations. For instance, only the hash of a disk image obtained can be documented in official reports, ensuring data integrity and facilitating verification processes.

2.6 Asymmetric cryptosystems

Desirable features include the ability to establish a short secret agreement over a public channel, send messages confidentially over an authenticated public channel without sharing secrets with recipients, and authenticate actual data.

The solution lies in asymmetric cryptosystems, which revolutionized cryptography. Before 1976, methods relied on human carriers or physical signatures. Then, innovations such as the Diffie-Hellman key agreement in 1976, public key encryption in 1977, and the introduction of digital signatures in the same year paved the way for modern cryptographic solutions.

2.6.1 Diffie-Hellman key agreement

The objective of the Diffie-Hellman key agreement protocol is to enable two parties to securely share a secret value using only public messages. Assuming an attacker model where interception of communications is possible, but tampering is not, and relying on the Computational Diffie-Hellman (CDH) assumption, the protocol operates under the following principle:

- Let $(\mathsf{G}, \cdot) \equiv \langle g \rangle$ represent a finite cyclic group, with two randomly sampled numbers a and b from the set $\{0, \ldots, |\mathsf{G}|\}$, where the length of a ($\lambda = \text{len}(a)$) is approximately logarithmic to the size of G.
- Given g^a, g^b, the computational complexity of finding g^{ab} is significantly greater than polynomial in the logarithm of the size of G.
- The most effective current attack strategy involves solving either for b or a, known as the discrete logarithm problem.

Example:
Let's consider two users, A and B:

- User A selects a random number a from the set $\{0, \ldots, |\mathsf{G}|\}$ and sends g^a to user B.
- User B selects a random number b from the set $\{0, \ldots, |\mathsf{G}|\}$ and sends g^b to user A.
- User A receives g^b from user B and computes $\left(g^b\right)^a$.
- User B receives g^a from user A and computes $(g^a)^b$.

Because the finite cyclic group $(\mathsf{G}, \cdot)$ is commutative, we can observe that $\left(g^b\right)^a = (g^a)^b$.

In practical implementations, a subgroup $(\mathbb{Z}_N^*, \cdot)$ is chosen from $(\mathsf{G}, \cdot)$, where $\mathbb{Z}_N^*$ denotes the set of integers modulo n

In this scenario, breaking the computational Diffie-Hellman assumption requires a minimum of:

$$\min\left(O\left(e^{k \log(n)^{\frac{1}{3}} \cdot \log(\log(n))^{\frac{2}{3}}}\right), O\left(2^{\frac{\lambda}{2}}\right)\right)$$

2.6.2 Public key encryption

In public key encryption, distinct keys are utilized for decryption and encryption purposes. It is computationally challenging to accomplish two tasks: decrypting a ciphertext without access to the private key and computing the private key solely from the public key.

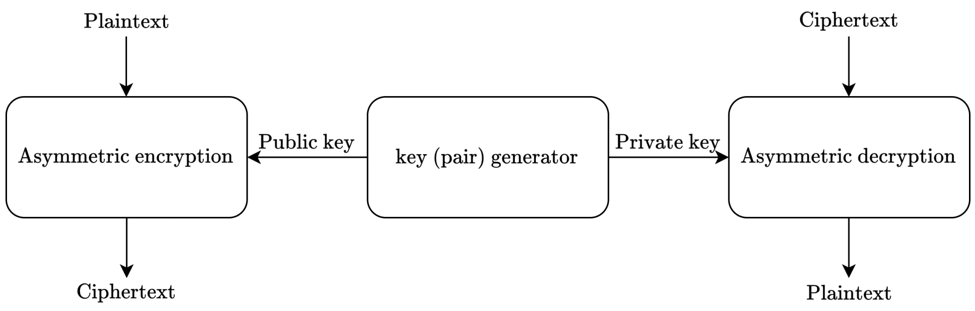


Figure 2.6: Key encryption

Algorithm Rivest, Shamir, Adleman (RSA) is a groundbreaking encryption algorithm introduced in 1977. It supports message and key sizes ranging from 2048 to 4096 bits, ensuring robust security against modern computational threats. Originally patented, RSA's intellectual property rights have since expired, fostering widespread adoption and further development within the cryptographic community. One notable advantage of RSA is its capability to encrypt messages without expanding ciphertext, ensuring efficient transmission and storage of encrypted data. Additionally, RSA encryption with a fixed key exhibits PRP properties, enhancing its versatility and applicability in various cryptographic scenarios.

The ElGamal encryption scheme, conceived in 1985, offers a versatile cryptographic solution characterized by its flexibility and patent-free nature. ElGamal encryption accommodates keys spanning either the k-bit range or hundreds of bits, contingent upon the chosen variant, allowing for tailored security configurations to suit various applications. Free from patent encumbrances, the ElGamal scheme has gained traction as a viable alternative to RSA, particularly in scenarios where patent restrictions were a consideration. A distinctive attribute of ElGamal encryption is its ciphertext, which typically spans twice the size of the plaintext. Despite this expansion, its widespread adoption attests to its effectiveness in safeguarding sensitive data and communications.

Usage Key encapsulation is a cryptographic technique used to securely transmit secret keys between parties over an insecure communication channel. In this method, the secret key is encapsulated or wrapped within another encryption layer using a public key algorithm. The recipient, possessing the corresponding private key, can then decrypt and extract the encapsulated key.

Example:
Let's consider a scenario where there exists a public channel between users A and B, and the attacker can only observe but not alter the communication. Subsequently, user B randomly selects a bit-string s from the set (k_{pri}, k_{pub}) encrypts it using k_{pub}, and forwards the resulting ciphertext to user A. User A, possessing the corresponding private key k_{pri}, decrypts the ciphertext and retrieves the bit-string s.

The process is then repeated with the roles of users A and B swapped. Consequently, both users obtain separate secrets. Although user B alone determines the value of the shared secret s, combining the two secrets derived from the exchanged messages yields analogous security guarantees to those of a conventional key agreement protocol.

Using an asymmetric cryptosystem, user B encrypts a message for user A without the requirement of pre-shared secrets. Theoretically, user B and user A could rely solely on an asymmetric cryptosystem for their communication needs. However, in practice, this method would prove highly inefficient. Asymmetric cryptosystems operate significantly slower compared to their symmetric counterparts, with performance degradation ranging from 10 to 1000 times.

Modern encryption Hybrid encryption schemes represent a strategic blend of asymmetric and symmetric cryptography techniques. In this approach, asymmetric algorithms are utilized for key transport or agreement, facilitating secure key exchange between parties. Meanwhile, symmetric algorithms are employed to encrypt the bulk of the data, ensuring efficient and swift encryption of large volumes of information. This concept serves as the cornerstone for all contemporary secure transport protocols, embodying a harmonious integration of both cryptographic methodologies to deliver robust and effective encryption for secure communication.

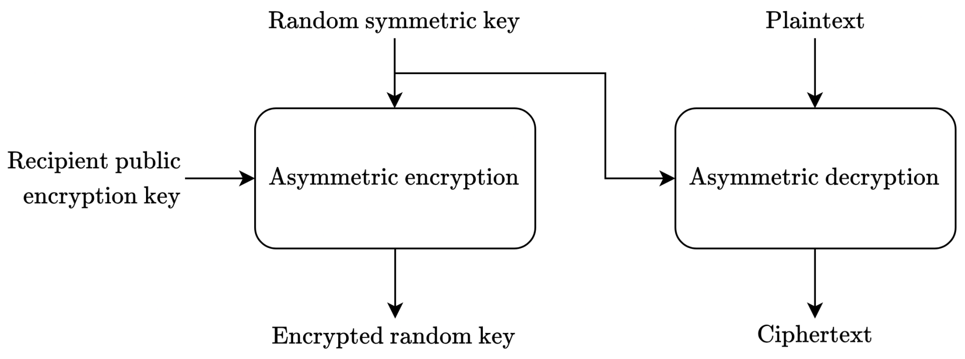


Figure 2.7: Modern encryption

2.7 Message authentication

Authenticating data serves as a crucial aspect in the establishment of secure hybrid encryption schemes. It ensures that the public key utilized by the sender corresponds accurately to the intended recipient. Additionally, the ability to verify the authenticity of data without relying on a pre-shared secret is highly desirable. Digital signatures play a pivotal role in achieving data authentication objectives:

- They offer robust evidence linking data to a specific user, enhancing data integrity.
- Verification of digital signatures does not necessitate a shared secret, simplifying the authentication process.
- Properly generated digital signatures cannot be repudiated by the user, ensuring accountability.
- Asymmetric cryptographic algorithms underpin digital signatures, providing a solid foundation for their security.
- It has been formally demonstrated that achieving non-repudiation without digital signatures is impractical, reinforcing their indispensable role in data authentication.

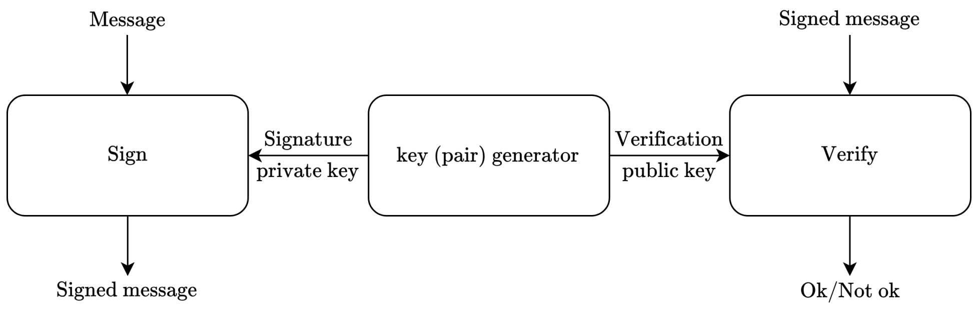


Figure 2.8: Digital signature

The computational challenges inherent in digital signatures encompass several key aspects:

- Signing a message without possessing the signature key, which includes attempting to splice signatures from unrelated messages.
- Computing the signature key when provided only with the verification key.
- Attempting to derive the signature key solely from signed messages, without access to additional information.

Algorithms In 1977, Rivest, Shamir, and Adleman (RSA) introduced a groundbreaking cryptographic method. This method employs a singular hard-to-invert function to craft both an asymmetric encryption scheme and a signature, with distinct message processing for each. Notably, the process of signing is significantly slower than verification, roughly around 300 times slower. This innovative approach has been standardized in NIST DSS (FIPS-184-4), underscoring its widespread adoption and importance in modern cryptographic practices.

The Digital Signature Standard (DSA) draws its foundations from adjustments made to signature schemes initially proposed by Schnorr and ElGamal. It, too, has been formalized in NIST DSS (FIPS-184-4), reflecting its establishment as a recognized cryptographic protocol. Notably, in DSA, the processes of signature creation and verification unfold at comparable speeds, distinguishing it from some other cryptographic methods.

Usages Digital signatures serve various purposes:

- *Authenticating digital documents*: in order to enhance efficiency, digital signatures frequently entail signing the hash of a document rather than the document itself. However, the assurance of the signature's reliability relies on the robustness of both the signature and hash algorithms.
- *Authenticating users*: digital signatures present an alternative approach to user authentication, serving as a viable replacement for traditional password-based logins. During this procedure, the server retains the user's public verification key, typically acquired during the account creation phase. Upon authentication requests, the server initiates the client to sign a lengthy, randomly generated bit-string, referred to as a challenge. Successful verification of the challenge signature by the client serves as compelling evidence of identity to the server.

2.8 Keys handling

The issue of securely binding public keys to user identities is paramount in both asymmetric encryption and digital signatures. Failure to authenticate public keys can lead to serious consequences, including susceptibility to Man-in-the-Middle attacks in asymmetric encryption and the potential for unauthorized signature generation by malicious actors.

To ensure the authenticity of public keys, an additional layer of verification, often in the form of another signature, is necessary. This additional signature serves as a guarantee of the legitimacy of the public-key and identity pairing. To facilitate this process, there is a requirement for a standardized format for distributing these signed pairs securely across systems.

2.8.1 Digital certificates

Digital certificates serve the purpose of associating a public key with a specific identity. This identity can be represented as an ASCII string for human interpretation or as either the Canonical Name (CNAME) or IP address for machine understanding. Additionally, these certificates outline the intended usage of the public key they contain, eliminating any potential ambiguities when a key format is suitable for both encryption and signature algorithms.

Furthermore, digital certificates include a designated time frame during which they are deemed valid.

2.8.2 Certification authorities

The certificates are signed by a trusted third party, known as the Certificate Authority (CA). This CA's public key is authenticated using another certificate. This process extends even to self-signed certificates, which must be trusted beforehand.

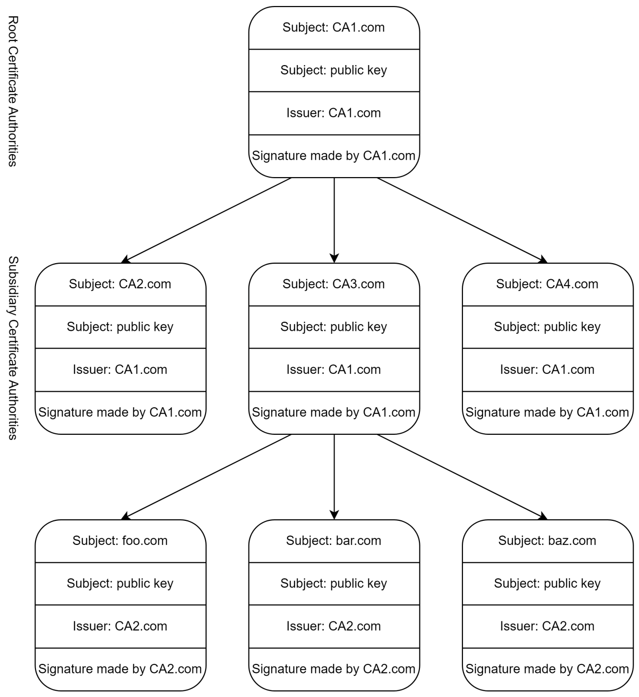


Figure 2.9: Certificate authorities hierarchy

2.9 State of the art

The contemporary secure communication protocols such as TLS, OpenVPN, and IPSec utilizes the following structure:

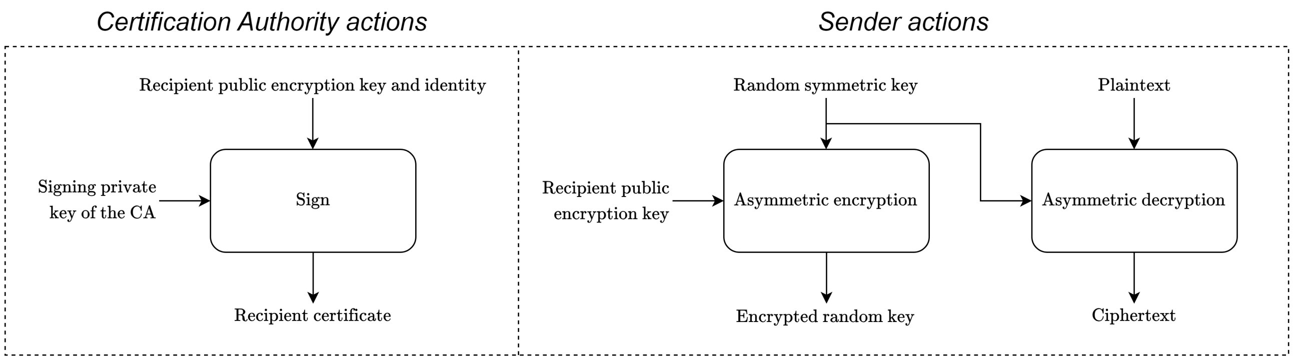


Figure 2.10: Secure communication protocols

Quantum computers With quantum computing some computationally challenging problems will become less hard, prompting a reassessment of their difficulty. There's a notable shift away from cryptosystems built upon factoring and discrete logarithm.

Compute on encrypted data Performing computations on encrypted data is feasible; however, it tends to be moderately to severely inefficient.

Physical access If the attacker gains physical access to the device executing the cipher (or can remotely measure it), it is essential to consider side-channel information within the attacker model.

2.10 Shannon's information theory

Shannon's information theory provides a mathematical framework for understanding communication and quantifying information.

Communication occurs between two endpoints:

- The sender comprises an information source and an encoder.
- The receiver consists of an information destination and a decoder.

Information is transmitted through a channel in the form of a sequence of symbols from a finite alphabet.

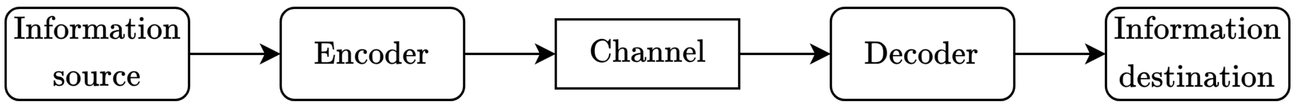


Figure 2.11: Shannon's communication structure

The receiver exclusively receives information through the channel. Until the symbol arrives, there remains uncertainty about what the next symbol will be. Consequently, we represent the sender as a random variable. Hence, obtaining information is akin to determining an outcome of a random variable $\mathcal{X}$, and the quantity of information relies on the distribution of $\mathcal{X}$. Encoding involves mapping each outcome to a finite sequence of symbols: more symbols are necessary when transmitting more information.

2.10.1 Entropy

We require a non-negative measure of uncertainty. The combination of uncertainties should correspond to adding entropies.

Definition (*Entropy*)**.** Let $\mathcal{X}$ be a discrete random variable with n outcomes in $\{x_0, \ldots, x_{n-1}\}$, where $\Pr(\mathcal{X} = x_i) = p_i$ for all $0 \leq 1 \leq n$. The entropy of $\mathcal{X}$ is given by:

$$H(\mathcal{X}) = \sum_{i=0}^{n-1} - \Pr_i \log_b \left(\Pr_i \right)$$

The unit of measurement for entropy is contingent on the base b of the logarithm, where the typical case for $b = 2$ is bits.

Example:
The random variable $\mathcal{X}$ represents a sequence of 6 uniform random letters (with 6^{26} combinations). In this case, the entropy is calculated as:

$$H(\mathcal{X}) = \sum_{i=0}^{6^{26}-1} -\frac{1}{6^{26}} \log_b \left(\frac{1}{6^{26}} \right) \approx 28.2b$$

On the other hand, if $\mathcal{X}$ represents a uniform selection from six-letter English words (with 6^6 combinations), the entropy is computed as:

$$H(\mathcal{X}) = \sum_{i=0}^{6^{6}-1} -\frac{1}{6^{6}} \log_b \left(\frac{1}{6^{6}} \right) \approx 12.6b$$

Theorem 2.10.1. *It is possible to encode the outcomes n of independent and identically distributed random variables, each with entropy $H(\mathcal{X})$, using at least $nH(\mathcal{X})$ bits per outcome. Encoding with fewer than $nH(\mathcal{X})$ bits will result in loss of information.*

Consequently, achieving arbitrary compression of bit-strings without loss is unattainable, necessitating cryptographic hashes to discard certain information.

Additionally, the task of guessing a piece of information (equivalent to one outcome of $\mathcal{X}$) is no less challenging than guessing a bit-string of length $H(\mathcal{X})$, disregarding momentarily the effort involved in decoding the guess.

2.10.2 Minimum entropy

Definition (*Min-entropy*)**.** The min-entropy is a measure of the most conservative assessment of the unpredictability of a set of outcomes. It is defined for $\mathcal{X}$ as:

$$H_\infty(\mathcal{X}) = -\log(\max_i \Pr_i)$$

In essence, it represents the entropy of a random variable with a uniform distribution, where each outcome has a probability of $\max_i p_i$.

It's worth noting that guessing the most common outcome of $\mathcal{X}$ is no less challenging than guessing a bit-string of length $H_\infty(\mathcal{X})$.

Example:
Consider the random variable $\mathcal{X}$ defined as:

$$\mathcal{X} = \begin{cases} 0^{128} & \text{with probability} \frac{1}{2} \\ a & \text{with probability} \frac{1}{2^{128}} \end{cases}$$

Here, $a \in 1\{0,1\}^{127}$. Predicting an outcome shouldn't be too difficult: just predict 0^{128}:

$$H(\mathcal{X}) = \frac{1}{2}\left(-\log_2\left(\frac{1}{2}\right)\right) + 2^{127}\frac{1}{2^{128}}\left(-\log_2\left(\frac{1}{2^{128}}\right)\right) = 64.5b$$

$$H_\infty(\mathcal{X}) = -\log_2\left(\frac{1}{2}\right) = 1b$$

Min-entropy indicates that guessing the most common output is as difficult as guessing a single bit string.

CHAPTER 3

Authentication

3.1 Introduction

Definition (*Identification*)**.** Identification refers to the action where an entity declares its identifier.

Definition (*Authentication*)**.** Authentication involves the process by which an entity provides evidence to confirm its claimed identity.

Authentication can take either a unidirectional or bidirectional (mutual) form.

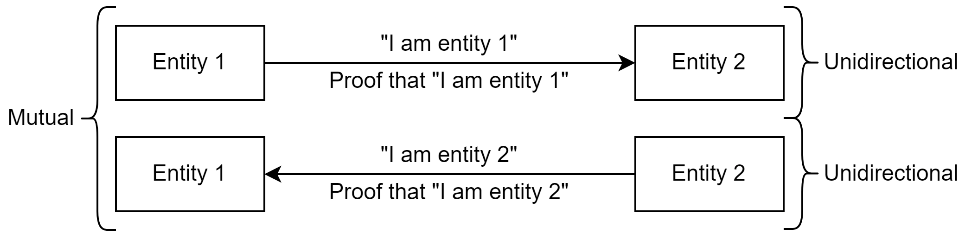


Figure 3.1: Authentication direction

This process can occur between various entities:

- Human to human.
- Human to computer.
- Computer to computer.

Authentication serves as a foundational step for subsequent authorization phases.

3.1.1 Factors of authentication

Authentication factors may include:

1. *Knowledge-based factors*: information that the entity knows (e.g., password or PIN).
2. *Possession-based factors*: items that the entity possesses (e.g., door key or smart card).

3. *Inherent factors*: characteristics that are unique to the entity (e.g., face or voice).

Multifactor authentication typically involves the use of two or three of these factors.

In human-centric scenarios, inherent factors are more commonly utilized than possession-based factors, and possession-based factors are more commonly used than knowledge-based factors. In machine-centric scenarios, knowledge-based factors are more frequently employed than possession-based factors, and possession-based factors are more commonly used than inherent factors.

3.2 Knowledge-based factor

The most commonly used knowledge-based factors are passwords and PINs.

Passwords offer several advantages: they are low-cost, easy to deploy, and have a low technical barrier for users. However, they also have significant drawbacks, primarily because they are susceptible to theft or snooping, guessing by unauthorized individuals, and being cracked through various methods.

To address these vulnerabilities, several countermeasures can be implemented:

- *Regularly changing or expiring passwords*: this reduces the risk of prolonged exposure if a password is compromised.
- *Utilizing lengthy passwords with a diverse range of characters*: this increases complexity and makes passwords harder to crack.
- *Ensuring passwords are not directly associated with the user*: this helps prevent predictable patterns that can be easily guessed.

Determining the most effective countermeasure requires anticipating the most probable attack in a given scenario. Once identified, prioritize countermeasures that users can feasibly follow and are likely to mitigate the identified threat effectively.

Countermeasures incur costs due to human limitations. Unlike machines, humans struggle to effectively safeguard secrets, find it challenging to remember complex passwords, and cannot adopt an unlimited number of countermeasures. The table below summarizes the effectiveness of different countermeasures against various attack types:

	Increase complexity	**Change password**	**Not being user-related**
Snooping	✗	✗	✗
Cracking	✓	∼	✗
Guessing	∼	∼	✓

3.2.1 Password complexity

User education is critical in addressing human weaknesses, which often serve as the weakest link in security. This involves implementing policies to enforce strong passwords and regular password changes or expiration. Additionally, employing password meters can help strike a balance between security and usability by guiding users in creating robust passwords.

For password complexity, it is essential to ensure that passwords contain a rich character set, including numbers, symbols, and both upper and lower-case letters. Moreover, passwords should be sufficiently long to resist brute-force attacks. Combining these elements enhances the strength of passwords and contributes to overall security.

3.2.2 Password exchange

Authentication involves sharing a secret, and several strategies can mitigate the risk of secrets being stolen: implement mutual authentication whenever feasible or utilize a challenge-response scheme, which involves exchanging random data to prevent replay attacks.

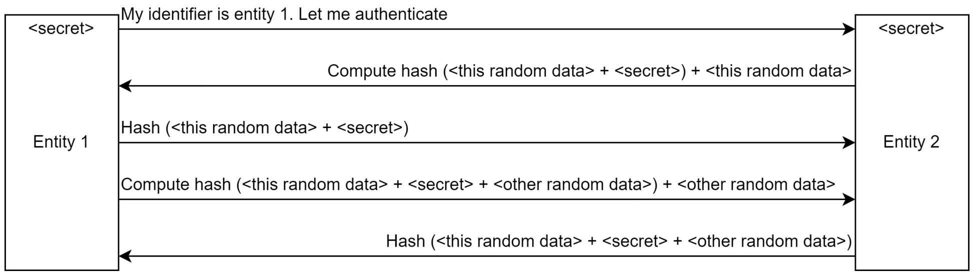

<secret>
Entity 1
<secret>
Entity 2
My identifier is entity 1. Let me authenticate
Compute hash (<this random data> + <secret>) + <this random data>
Hash (<this random data> + <secret>)
Compute hash (<this random data> + <secret> + <other random data>) + <other random data>
Hash (<this random data> + <secret> + <other random data>)

Figure 3.2: Countermeasure costs

3.2.3 Password storage

To mitigate the risk of secrets being stolen from a file containing usernames and passwords stored by the operating system, the following measures can be implemented:

- *Employ cryptographic protection*: ensure that passwords are never stored in clear text. Instead, use techniques such as hashing combined with salting to mitigate dictionary attacks.
- *Implement access control policies*: limit privileges for reading and writing to the password file to authorized users only.
- *Avoid disclosing secrets in password-recovery schemes*: ensure that password recovery mechanisms do not inadvertently reveal sensitive information.
- *Address caching problems*: be mindful of information stored in intermediate locations, as this can pose security risks. Regularly review and manage cached data to minimize potential exposure.

3.3 Possession-based factor

Possession-based factors commonly include tokens, smart cards, and smartphones. These factors offer several advantages, such as reducing the likelihood of users handing out keys, being relatively low-cost, and providing a good level of security. However, they also have drawbacks, including deployment challenges and the risk of being lost or stolen.

To mitigate these vulnerabilities, implementing a second factor alongside passwords or exploring alternative authentication methods can be effective. Possible solutions include:

- *One-Time Password generators* (OTP): these operate on the principle of a secret key synchronized with a counter on the host system. The client computes a MAC using the counter and key, which is then verified by the host system.
- *Smart cards*: these devices contain a CPU and non-volatile RAM housing a private key. During authentication, the smart card verifies its identity to the host system through a challenge-response protocol.

- *Static OTP*: these consist of sequences known to both the client and the host. The host selects challenges, typically random numbers or specific criteria, and the client responds, ideally transmitting the response over an encrypted channel.
- *Time-based OTP*: these replicate the functionality of password generators but differ in implementation. Password generators are typically embedded systems that operate on general-purpose software and hardware platforms.

3.4 Inherent factor

Inherent factors, most commonly utilized in biometric authentication, offer several advantages, including a high level of security and the absence of extra hardware requirements. However, they also have notable drawbacks, such as deployment challenges, probabilistic matching, invasive measurement techniques, susceptibility to cloning, changes in biometric characteristics over time, privacy concerns, and issues for users with disabilities.

To address these vulnerabilities, potential countermeasures may include:

- Regularly re-measuring biometric data.
- Securing the biometric authentication process.
- Providing alternative authentication methods for users who may face difficulties with biometric authentication.

Fingerprint authentication Fingerprint authentication involves several steps:

1. *Enrollment*: a reference sample of the user's fingerprint is acquired by a fingerprint reader. From this sample, features are derived. These extracted feature vectors are then securely stored in a database.
2. *Authentication*: a new fingerprint reading is taken. The features of this newly captured fingerprint are compared with the reference features stored in the database. Access is granted if the similarity between the captured and reference features exceeds a predefined threshold.

However, a main challenge in fingerprint authentication is the occurrence of false positives and false negatives.

3.5 Single Sign-On

Single Sign-On (SSO) addresses the complexity of managing and remembering multiple passwords. This issue often leads to password reuse across different sites, which can compromise security. Additionally, replicating password policies across various platforms can be costly and inefficient.

SSO offers a solution by establishing a single identity, typically supported by one or two authentication factors, and designating one trusted host. Users authenticate or sign on to this trusted host, and other hosts can verify a user's authentication status by querying the trusted host.

The main drawbacks of SSO include having a single point of trust: the trusted server. If this server is compromised, all affiliated sites are compromised as well. Implementing SSO correctly is often challenging for developers.

CHAPTER 4

Software security

4.1 Introduction

In the realm of software engineering, meeting requirements encompasses both functional and non-functional aspects:

- *Functional requirements*: the software must effectively execute its intended purpose.
- *Non-functional requirements*: usability (the software should be user-friendly and intuitive), safety (the software should maintain a secure environment for users and data), and security (robust measures should be in place to protect against breaches and unauthorized access).

Recognizing the significance of crafting inherently secure applications is pivotal for any proficient developer or software engineer. However, it's worth acknowledging that developing secure software presents formidable challenges.

Software must conform to the specified requirements. Not meeting a requirement results in a software bug. Failure to meet a security requirement creates a vulnerability. Exploiting a vulnerability to compromise the CIA of a system is termed an exploit.

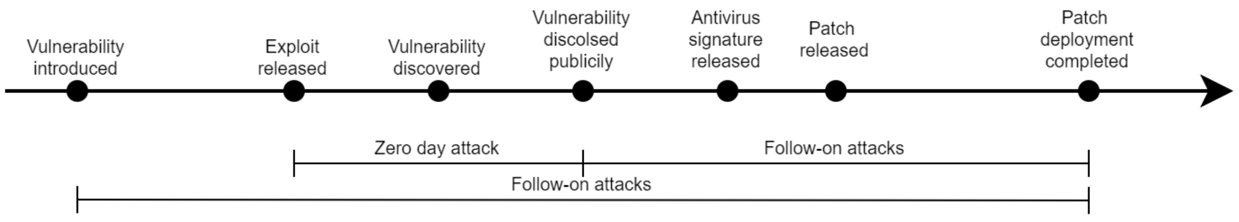


Figure 4.1: Vulnerability lifecycle

4.1.1 Design principles

1. *Minimize privileged access*: limit privileged access to the essential components.
2. *Keep it simple, stupid (KISS)*: embrace simplicity in design to reduce potential vulnerabilities.

3. *Immediate privilege discard*: promptly discard privileges to mitigate potential security risks.
4. *Open design*: rely on openness rather than obscurity for security, aligning with Kerckhoffs' principle.
5. *Address concurrency and race conditions*: be vigilant about handling concurrency and race conditions.
6. *Fail-safe and default deny*: implement fail-safe mechanisms and default deny policies to enhance security posture.
7. *Avoid shared resources and untrusted libraries*: refrain from using shared resources against incorporating unknown or untrusted libraries into the system.
8. *Input and output filtering*: implement robust input and output filtering mechanisms to mitigate potential attack vectors.
9. *Use established cryptographic primitives*: avoid developing custom cryptographic primitives, password, or secret management code; instead, rely on audited and trusted cryptographic libraries.
10. *Trustworthy random number generation*: utilize reliable random number generators to ensure the integrity of cryptographic operations and secure communications.

Bug-free software is an ideal that's practically unattainable. While not all bugs lead to vulnerabilities, achieving software devoid of vulnerabilities is a challenging task. It's important to remember that vulnerabilities can exist without any working exploits targeting them.

4.2 Binary files

Binary formats contain vital information regarding a file's organization on disk, memory loading procedures, file type, machine class, and sections. ELF (Executable and Linkable Format) binaries adhere to the following structure:

- *ELF header*: this component delineates the overarching structure of the binary, specifying the file type and demarcating the boundaries for section and program headers.
- *Program headers*: these headers explain how the file will be loaded into memory. They segment the data into distinct parts and establish mappings between sections and segments.
- *Section headers*: these headers provide a representation of the binary as it exists on disk, defining various sections, including:
  - `.init`: contains executable instructions responsible for initializing the process.
  - `.text`: holds the executable instructions of the program.
  - `.bss`: reserved for statically-allocated variables.
  - `.data`: reserved for initialized data.

Segment Segments represent the runtime view of a binary. They define how the binary will be loaded into memory, dividing the data into distinct segments and specifying the mapping between sections and segments. Segments play a crucial role in the execution of the program by providing the necessary information for the operating system to allocate memory and execute the binary.

Section Sections in an ELF binary contain linking and relocation information. They provide granular details about the various components of the binary, such as code, data, and symbols. Sections are essential for the linking process, aiding in resolving external symbols and determining memory layout. They also facilitate relocation, enabling the binary to be loaded and executed correctly in different memory locations. Sections serve as the building blocks for the organization and structure of the ELF binary.

4.2.1 Linux processes

Creation When a program is executed in Linux, it undergoes a series of steps to be mapped into memory and organized for execution:

1. *Creation of virtual address space*: the kernel initiates by creating a virtual address space dedicated to the program's execution, providing isolation and abstraction.
2. *Loading information from executable file*: the kernel, with the assistance of the dynamic linker, loads relevant information from the executable file into the newly allocated address space. This process involves loading the segments specified by the program headers of the ELF binary.
3. *Setup of stack and heap*: after loading the necessary information, the kernel sets up the stack and heap within the program's address space. The kernel then initiates execution by jumping to the designated entry point of the program.

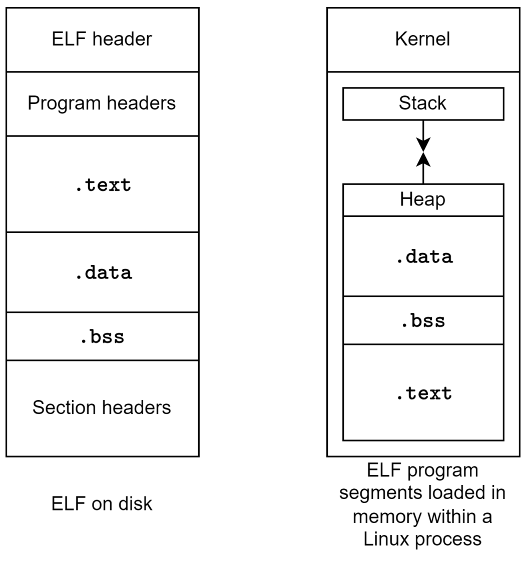


Figure 4.2: Process layout in Linux

Execution Once a program is correctly created, the virtual address space contains the following elements:

- *Argc, env pointer, and stack*: this region includes statically allocated local variables, environment variables, and function activation records. The stack grows downward, towards lower addresses, as more function calls and local variables are added.
- *Unallocated memory.*
- *Heap*: dynamically allocated data resides in this section. The heap grows upward, towards higher addresses, as more memory is dynamically allocated during program execution.
- `.data`, `.bss`, and `.text`.
- *Shared libraries.*

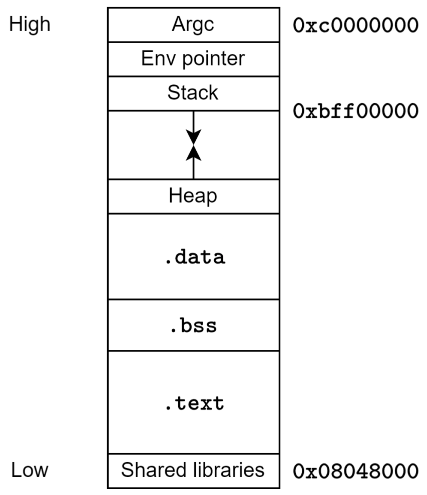


Figure 4.3: Code structure

Termination When the program completes its execution, the CPU needs to determine where to jump next. Typically, after a function finishes executing, the control flow returns to the point in the program from which the function was called. This return address is crucial for maintaining the program's execution flow.

To achieve this, the CPU saves the current instruction pointer (EIP) onto the stack before jumping to the called function. When the function completes its execution, the CPU retrieves the return address from the stack and jumps to that address, resuming execution from the point where the function was initially called.

4.3 Buffer overflow

Stack smashing, commonly seen in C programming, occurs due to unsafe practices when a buffer is filled with data exceeding its allocated size, leading to a buffer overflow vulnerability. Several standard C library functions are prone to causing stack smashing if not used carefully, including: `strcpy`, `strcat`, `fgets`, `gets`, `sprintf`, `scanf`.

Improper handling of these functions can result in stack smashing vulnerabilities, allowing attackers to execute arbitrary code, crash the program, or gain unauthorized access to sensitive information. Therefore, it's essential for programmers to use safer alternatives or apply proper input validation and buffer size checks to mitigate these risks.

4.3.1 Buffer address

Guessing the buffer address for executing arbitrary machine code in the overflowed buffer can be challenging due to several factors:

- *Proximity to ESP*: one common approach is to guess that the overflowed buffer is near the stack pointer (ESP). Debugging tools like gdb can help determine the approximate location of ESP at runtime.
- *Variability*: the exact address of the buffer may change with each execution or across different machines, making it difficult to predict reliably.
- *CPU precision*: CPUs are not designed to handle imprecise memory accesses gracefully.

The problem of precision arises because executing arbitrary machine code relies on accurately predicting the memory location of the buffer. However, due to the dynamic nature of memory allocation and the lack of precision in CPU operations, achieving reliable execution of arbitrary code in a stack overflow scenario is challenging. Security measures such as Address Space Layout Randomization (ASLR) further complicate this task by introducing additional randomness to memory addresses, making it even harder to predict the buffer's location.

In practical scenarios, obtaining the ESP value can be done using a debugger or reading from the process directly. However, it's important to note that certain debuggers, like gdb, may introduce an offset to the allocated process memory. Consequently, the ESP value obtained from gdb might differ slightly from the ESP value obtained by reading directly within the process.

Despite these methods, precision issues persist. Even with accurate ESP values, precisely determining the buffer's location remains challenging. This lack of precision can impede the reliable execution of arbitrary code in a stack overflow scenario.

nop sled To address the precision issue, we can utilize a nop sled. A nop sled is a sequence of nop (no operation) instructions, represented by the hexadecimal value `0x90` on x86 architecture. These instructions do nothing when executed and serve as a landing strip for the program's execution flow.

At the beginning of the buffer, we place a sequence of nop instructions, creating the nop sled. This sled acts as a safe landing zone for the program's execution flow. By jumping to anywhere within the nop sled range, we ensure that even if the precise location of the buffer is not accurately determined, the program will still land on valid instructions within the nop sled and continue execution until it reaches the desired executable code.

4.3.2 Process selection

Historically, attackers have often aimed to spawn a privileged shell on a local or remote machine.

Definition (*Shell code*)**.** Shell code refers to a sequence of machine instructions designed to open a shell, typically a privileged one.

While shell code can perform various actions, spawning a shell has been a common objective due to the elevated privileges it provides. To achieve this, attackers typically invoke the `execve` system call, which executes a program specified by its path name.

In Linux systems, invoking a system call involves executing a software interrupt using the `int` instruction with the value `0x80`.

Shell code Creating shell code usually starts with high-level code and involves translating it into machine instructions, often using assembly language. An alternative approach is to write the code in C and then extract the relevant instructions to compose the shell code. The process can be outlined as follows:

1. *Write high-level code*: begin by writing the desired functionality in high-level C code.
2. *Compile and disassembly*: compile the C code and then disassemble the resulting binary to obtain its assembly instructions.
3. *Analyze assembly*: analyze the disassembled code to identify and extract only the relevant instructions needed to achieve the desired functionality.
4. *Extract opcode*: identify the operation code for each relevant instruction.
5. *Create the shell code*: finally, assemble the extracted opcodes into a sequence of bytes to form the shell code.

4.3.3 Countermeasures

A layered defense strategy is effective in mitigating buffer overflow vulnerabilities:

1. *Source code level defenses*: address vulnerabilities within the source code to prevent buffer overflows. This involves educating developers on secure coding practices, conducting targeted testing, and using source code analyzers. Additionally, adopting safer libraries and programming languages with dynamic memory management, such as Java, can provide inherent protection against these vulnerabilities.
2. *Compiler level defenses*: utilize compiler-level techniques to make vulnerabilities harder to exploit. Compiler warnings can alert developers to potential issues before the code is executed. Techniques such as randomizing the order of stack variables can introduce variability that complicates exploitation. Incorporating stack protection mechanisms during compilation further enhances security.
3. *Operating system level defenses*: implement operating system-level measures to obstruct or complicate attacks. Non-executable stack configurations help prevent stack smashing attacks by distinguishing data from executable code. Despite non-executable stack defenses, attackers may bypass them by redirecting the return address to existing machine instructions, a technique known as code-reuse attacks. Despite these defenses, attackers may use techniques like code-reuse attacks to bypass protections. Address Space Layout Randomization (ASLR) adds further security by randomizing memory locations, including the stack, making it difficult for attackers to predict return addresses.

Stack canary The stack canary mechanism involves placing a sentinel value between local variables and control values to detect tampering. The integrity of this canary is checked during the function's epilogue, and if tampering is detected, the program is terminated. Various types of canaries enhance security:

- *Terminator canaries*: use terminator characters, such as `/0`, which are not copied by string-copy functions, making them resistant to overwriting during attacks.
- *Random canaries*: generated as random sequences of bytes at program execution.
- *Random XOR canaries*: randomly generated canaries that are XORed with a portion of the protected structure. This XOR operation adds complexity, making the canary harder to manipulate even if the buffer overflow does not occur.

4.4 Format string bugs

Definition (*Format string*)**.** A format string is used to craft output strings that include variables, allowing for customizable formatting as specified by the programmer.

Definition (*Variable placeholder*)**.** Variable placeholders dictate how data is formatted into a string when using format functions.

Common format printing functions include: `printf`, `fprintf`, `vfprintf` and `sprintf`.

In a stack examination, we often encounter a portion of our format string, as it is commonly stored on the stack. By retrieving what's placed on the stack, we can analyze it using the `%N$x` syntax to access the N-th parameter and perform straightforward shell scripting. This method can also be leveraged to discover valuable data in memory, highlighting a vulnerability known as information leakage.

4.4.1 Stack writing

The `%n` placeholder in format strings allows us to write the number of characters printed so far into a memory address specified by an argument. To determine the number of bytes printed so far, the `%c` placeholder can be used. This placeholder specifies the precision of the printed number and can be applied to both characters and strings.

To write an arbitrary number (e.g., `0x6028`) to a target address (e.g., `0xBFFFF6CC`), follow these steps:

1. *Include the target address*: push the target address onto the stack as part of the format string.
2. *Locate the target address*: use `%x` to find the target address on the stack, identified as `%N$x`, where `N` represents the position on the stack. Let's call this displacement `pos`.
3. *Write the value*: use `%c` and `%n` to write `0x6028` into the memory cell pointed to by the target address. Ensure to account for the parameter of `%c` and the length of the printed characters.

32 bit addresses To efficiently write to a 32-bit address and avoid processing excessive amounts of data, we can split the address into two 16-bit components and perform two separate write operations. Here's the procedure to write a 32-bit address in two stages:

1. *Identify target addresses*: determine the target memory addresses for the two write operations.
2. *Calculate displacements*: compute the positions (displacements) of these target addresses on the stack.
3. *Compute write values*: calculate the values to be written into these addresses. The sum of these values should equal the original 32-bit address.
4. *Prepare the format string*: push the two target addresses (one for each write operation) onto the stack as part of the format string.
5. *Locate target addresses*: use `%x` to find the first target address `<target_1>` on the stack (indicated as `%N$x`). The second target address `<target_2>` will be located at `pos+1`.
6. *Execute write operations*: use `%c` and `%n` to:

   - Write the lower 16-bit value to the memory cell pointed to by `<target_1>`.
   - Write the higher 16-bit value to the memory cell pointed to by `<target_2>`.

The final format string will become:

```
<tg><tg+2>%<low-8>c%pos$hn<high-low>c%pos+1$hn
```

4.4.2 Process selection

The choice of target address can vary depending on the exploit's context. Here are some common locations for the target address:

- *Saved return address* (saved EIP).
- *Global Offset Table* (GOT): used for dynamic relocations of functions in shared libraries.
- *C library hooks*: overwriting these hooks can alter program behavior.
- *Exception handlers*: overwriting exception handlers can allow control of program execution when exceptions occur.

4.4.3 Countermeasures

Here's an overview of countermeasures against format string vulnerabilities:

- *Memory error countermeasures*: implement safeguards such as stack canaries and ASLR to prevent exploitation.
- *Compiler warnings*: compilers help mitigate vulnerabilities by issuing warnings.
- *Library patches*: updated versions of the C standard library address format string vulnerabilities.

CHAPTER 5

Web Application security

5.1 Web Application security

Web Applications have become the primary method for delivering software across various environments, including corporate intranets, SaaS platforms, and Cloud services. These applications are often designed to be publicly accessible, similar to public web services, and operate on the stateless HTTP protocol. This protocol's stateless nature requires additional mechanisms to simulate state management for maintaining user sessions and data between requests. Moreover, HTTP's built-in authentication methods are relatively weak, necessitating the implementation of more robust authentication strategies to enhance security.

The key principle in web application security is that the client should never be trusted implicitly. It is crucial to rigorously filter and validate all data received from the client to mitigate risks associated with potentially malicious inputs or attacks.

Data filtering Filtering data is a complex task, but employing a variety of validation techniques can significantly improve security:

- *Allow listing*: only permit data that meets predefined criteria, effectively restricting input to expected and safe values.
- *Black listing*: exclude known malicious content in addition to using allow listing. This method is supplementary and helps catch specific threats.
- *Escaping*: convert special characters into safer forms to prevent them from being interpreted as executable code or commands.

A fundamental principle to follow is that allow listing generally provides stronger security compared to black listing, as it proactively controls what data is permitted rather than reacting to known threats.

5.2 Cross-Site Scripting

Consider a simple blog application where users can post comments. If the application displays comments directly without filtering, an attacker could insert malicious code, such as:

```
<script> alert('JavaScript Executed'); </script>
```

If this input is not properly sanitized, the script will execute on the screens of other visitors. This vulnerability is known as Cross-Site Scripting (XSS).

XSS is a security flaw where an attacker can inject client-side code into a web page. There are three primary types of XSS attacks:

1. *Stored XSS*: the malicious input is saved on the target server, such as in a database. When a victim accesses the affected content, the malicious code is retrieved and executed in the victim's browser.
2. *Reflected XSS*: the malicious input is sent to the Web Application in a request. The application includes this input in its response without sanitizing it, causing the malicious script to execute in the client's browser.
3. *DOM-based XSS*: the attack occurs entirely within the victim's browser. Malicious scripts are executed by client-side JavaScript code without the need to send data back to the server.

XSS can lead to severe consequences, including:

- *Cookie theft*: stealing session cookies or hijacking user sessions.
- *Session manipulation*: altering sessions and performing unauthorized actions.
- *Data theft*: eavesdropping on sensitive information.
- *Drive-by downloads*: initiating unwanted software downloads.
- *Bypassing Same Origin Policy*: overcoming restrictions intended to isolate content from different origins.

Same Origin Policy The Same Origin Policy (SOP) is a fundamental security feature enforced by web browsers. SOP ensures that client-side code loaded from one origin (domain) can only interact with data from the same origin. This policy helps prevent malicious scripts from accessing sensitive data on other origins. However, modern web technologies, such as Cross-Origin Resource Sharing (CORS) and various client-side extensions, can sometimes bypass these restrictions, leading to potential security concerns.

5.2.1 Content Security Policy

Content Security Policy (CSP) is a W3C specification designed to enhance web security by instructing browsers on which content sources are considered trustworthy. It acts as an extension of the Same-Origin Policy, providing more control and flexibility in defining what content can be loaded and executed. CSP is implemented through a set of directives sent from the server to the client via HTTP response headers.

CSP includes a variety of directives to control different aspects of web content, such as:

- `script-src`: specifies the sources from which JavaScript can be loaded.
- `form-action`: defines the valid endpoints to which forms can be submitted.
- `frame-ancestors`: lists the sources allowed to embed the page within frames or iframes.

- `img-src`: specifies the allowed origins for loading images.
- `style-src`: controls the sources from which stylesheets can be loaded.

The effectiveness of these directives depends on the browser's implementation, and while CSP is increasingly adopted, it comes with its own set of challenges:

- *Policy creation*: identifying who is responsible for writing and managing CSP rules.
- *Manual process*: the process of creating and maintaining CSP policies is largely manual and complex.
- *Limited automation*: there are few automated tools available to assist with policy creation, leaving much of the task to manual efforts.
- *Ongoing maintenance*: keeping CSP policies up-to-date can be difficult, especially as web pages often load resources from multiple sources, and both pages and resources can change dynamically.

5.3 SQL injection

SQL injection is a vulnerability that allows attackers to manipulate SQL queries by injecting malicious input. This often involves terminating a query prematurely, commenting out parts of the query, or adding additional SQL commands.

To terminate an SQL statement, a semicolon followed by a double dash (–) is used to comment out the remainder of the line. This technique can be exploited to alter the behavior of a query. For example:

```
' OR '1'='1';--
```

This injection alters the query to always return true, potentially bypassing authentication or other security checks.

Unions Union-based SQL injection allows attackers to retrieve data from different tables within a single query. For example:

```
SELECT name, phone, address FROM Users
WHERE Id='' UNION ALL SELECT name,creditCardNumber,CCV2
FROM CreditCardTable;--';
```

This query combines results from two different tables. It works only if the number and types of columns in the original and injected queries match.

Insertions Insert injection involves modifying an insert statement to insert arbitrary data or execute sub-queries. For example:

```
INSERT INTO results VALUES (NULL, 's.zanero',
    (SELECT password from USERS where user='admin')
    )--', '18')
```

Here, the injected sub-query retrieves the password of an admin user, which is then inserted into the results table.

Blind injections Blind SQL injection occurs when the SQL queries do not return data directly but still allow an attacker to infer information based on the application's behavior. For example, you may infer whether a condition is true or false based on the response or timing of the application.

5.3.1 Countermeasures

To protect against SQL injection attacks, consider the following measures:

- *Input sanitization*: validate and filter user inputs to ensure they do not contain harmful SQL syntax.
- *Prepared statements*: use parameterized queries instead of directly concatenating user inputs into SQL statements. Prepared statements use placeholders for variables, which prevents injection.
- *Avoid dynamic queries*: minimize the use of dynamic SQL and avoid using table names or other sensitive identifiers as field names to prevent information leakage.
- *Restrict database privileges*: implement least privilege principles by limiting the types of queries that different users can execute on the database. Ensure that users have only the permissions necessary for their roles.

5.4 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is an attack that tricks a user into performing unwanted actions on a Web Application where they are currently authenticated. This is achieved by leveraging the user's existing credentials, such as cookies, which are automatically included with each request made from their browser. Consequently, attackers can craft malicious requests that are processed by the vulnerable Web Application using the victim's credentials. Since the Web Application cannot differentiate between legitimate user actions and those initiated by an attacker, it is at risk of executing unauthorized commands.

5.4.1 Countermeasures

To protect against CSRF attacks, several effective countermeasures can be implemented:

- *Anti-CSRF tokens*: use a unique, random token associated with each user session. This token should be included in any form or request that performs sensitive operations. The server generates and stores this token, and it must match the token submitted with the request for the operation to proceed. Importantly, these tokens should not be stored in cookies to avoid exposure to cross-site requests.
- *SameSite cookies*: utilize the SameSite attribute for cookies to restrict how cookies are sent with cross-site requests. This attribute helps prevent session cookies from being included in requests originating from other sites:
  - SameSite=Strict: cookies are sent only in requests originating from the same site, providing strong protection against CSRF.

  - SameSite=Lax: cookies are sent with top-level navigation requests but not with cross-site post requests, images, or iframes, offering a balance between usability and security.

5.5 Other vulnerabilities

5.5.1 Cookies

HTTP operates in a stateless manner, meaning it lacks inherent memory of previous interactions. While HTTP predominantly facilitates data flow from client to server, it lacks a native ability for the server to persist information on the client side. However, cookies emerged as a solution, enabling client-side storage of information and providing a dependable mechanism for maintaining stateful data. Originally conceived for website customization purposes, cookies have been susceptible to misuse, leading to privacy infringements. Additionally, notions such as user authentication and session management, when implemented through cookies, can pose security risks if not handled properly.

The process of establishing a session using cookies involves the following steps:

1. The user submits their Web Server username and password.
2. The Web Server generates and stores a Session ID.
3. The Web Server responds by sending a cookie containing this Session ID.
4. The user's browser stores the session ID and includes it in subsequent requests to the Web Server.

Issues concerning session cookies include:

- Preventing the prediction of tokens received or to be received by the client (next token) to thwart impersonation and spoofing attacks and minimize the impact of session stealing.
- Ensuring that every token has a reasonable expiration period, although this should not be set in the cookies themselves.
- Employing cookie encryption for sensitive information and utilizing storage mechanisms such as Message Authentication Codes (MACs) to prevent tampering.

Session hijacking Due to the stateless nature of HTTP, session hijacking can occur through methods such as stealing a cookie via an XSS attack or brute-forcing a weak session ID parameter.

5.5.2 Information leakage

While detailed error messages enhance Human-Computer Interaction (HCI), they can inadvertently lead to security vulnerabilities. Information leakage can occur through various avenues, including active debug traces in production environments. Additionally, risks stem from the insertion of user-supplied data in errors, which may expose vulnerabilities like Reflected Cross-Site Scripting. Side channels are another potential source of data leaks.

5.5.3 Password security

The essentials of password security stay the same: passwords should never be stored in plain text in Web Applications to reduce the risk of exposure in case of a breach. Employing techniques like salting and hashing is crucial to prevent attacks like rainbow tables. It's crucial to handle password reset processes carefully: Typically, resetting a password involves providing an alternative login method. Common methods include sending a reset link to a registered email. Less secure practices include sending temporary passwords or relying solely on security questions for verification.

Brute-forcing protection Protecting against brute-force attacks requires thoughtful measures:

- A naive solution involves locking an account after a certain number of failed login attempts. However, this can lead to reverse brute-forcing, where attackers focus on other accounts.
- Making accounts non-enumerable can prevent attackers from cycling through usernames systematically.
- Blocking IP addresses may seem intuitive, but it's not foolproof due to proxies and NATs. Moreover, this approach can inadvertently lead to Denial of Service (DoS) attacks.

CHAPTER 6

Network protocol attack

6.1 Introduction

Networks encompass a diverse range of physical media, topologies, and protocols. To effectively manage this complexity, a layered approach is crucial.

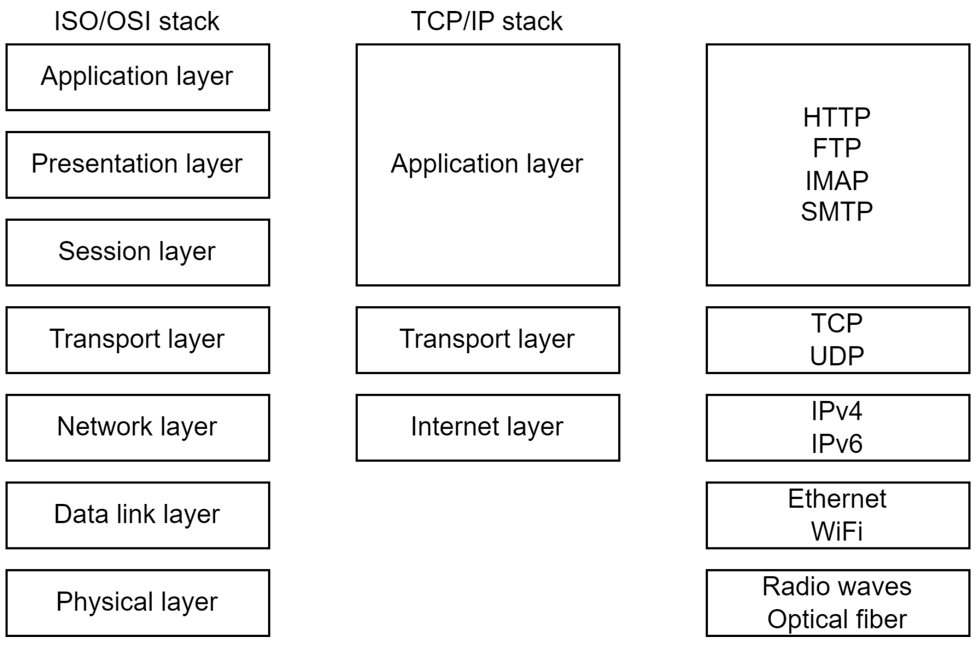


Figure 6.1: Layering and protocols

In network communications, hosts are uniquely identified by addresses, similar to phone numbers or postal addresses. Each layer in the network stack utilizes its own addressing scheme:

- *Data link layer*: utilizes MAC addresses (for Ethernet), which are globally unique identifiers embedded in the Network Interface Card (NIC). The Address Resolution Protocol (ARP) is used to map an IP address to a MAC address.

- *Internet layer*: employs IP addresses to identify network hosts globally. It includes both public and private addresses, with private addresses defined by RFC 1918 for IPv4.

- *Transport layer*: uses port numbers to identify specific services on a host.

Transport protocols Transport protocols facilitate communication between hosts:

- *User Datagram Protocol* (UDP): a connectionless protocol that provides a lightweight wrapper around an IP packet, primarily adding a port number for addressing.
- *Transmission Control Protocol* (TCP): a connection-oriented protocol that manages state through concepts such as closed, open, and established connections. It uses a three-way handshake to establish connections.

Definition (*Denial Of Service*)**.** A Denial of Service (DoS) attack targets the availability of a service, rendering it inaccessible to legitimate users.

Definition (*Sniffing*)**.** Sniffing refers to the unauthorized reading of network packets, compromising confidentiality.

Definition (*Spoofing*)**.** Spoofing involves forging network packets, thereby compromising the integrity and authenticity of communications.

6.2 Network attacks

In typical network operations, a Network Interface Card (NIC) is configured to intercept and forward only the packets addressed specifically to its host IP address. However, when a NIC is set to promiscuous mode, it captures all packets it reads from the network, regardless of their destination address. This mode can be used for network diagnostics, but it also exposes all traffic to potential snooping.

Historical context Originally, Ethernet networks utilized a shared medium with BNC cables, meaning that all traffic was broadcast to every device on the network. Even though RJ-45 cables replaced BNC, the use of hubs maintained this broadcast behavior. Hubs transmit all incoming traffic to every device within the broadcast domain, making all network traffic visible to all connected hosts.

Modern networks often use switches instead of hubs. Switches enhance performance by forwarding packets only to the specific port associated with the destination MAC address, rather than broadcasting to all ports. This selective forwarding improves efficiency but is not designed for security purposes.

Spoofing UDP (User Datagram Protocol) does not authenticate the source IP address, making it vulnerable to spoofing. Attackers can easily alter the source address in UDP or ICMP packets, leading to various malicious outcomes. If an attacker is on a different network than the target, they won't receive responses to their spoofed packets, as these responses are sent to the forged source address. This is known as blind spoofing. Conversely, if the attacker is on the same network as the target, they can capture responses or use techniques like ARP spoofing to exploit further network vulnerabilities.

TCP (Transmission Control Protocol) differs from UDP by using sequence numbers for packet ordering and acknowledgment. During connection establishment, TCP uses a semi-random Initial Sequence Number (ISN). An attacker who can accurately predict the ISN might complete the TCP three-way handshake without receiving responses. However, this requires that the attacker's spoofed source address does not receive any responses. If response packets are received, the target may send a Reset (RST) packet, which can disrupt the connection and potentially alert the target to the attack.

6.2.1 Killer packets attacks

We may use killer packets to perform a DoS on a user. We may use:

- *Smurf attack*: this attack involves sending ICMP echo requests (pings) with a spoofed source address (the victim) to a network's broadcast address. This results in a flood of responses from all devices on the network, overwhelming the victim and potentially causing system crashes or erratic behavior.
- *Teardrop attack*: this exploit takes advantage of vulnerabilities in the TCP reassembly process. By sending fragmented packets with overlapping offsets, the attacker can cause the target system's kernel to hang or crash during packet reassembly.
- *Land attack*: this attack targets older systems, such as Windows 95. It involves sending a packet where the source IP address is the same as the destination IP address, with the SYN flag set. This can cause the TCP/IP stack to enter a loop, leading to a system lock-up.

6.2.2 Syn flood attacks

SYN Flood attacks exploit the TCP three-way handshake process. An attacker sends a high volume of SYN requests with spoofed source addresses, filling the server's queue with half-open connections. This congestion prevents legitimate SYN requests from being processed, effectively denying service to legitimate users.

Countermeasures To mitigate SYN Flood attacks, SYN cookies can be employed. This technique involves sending a SYN+ACK response while discarding the half-open connection. The server waits for a subsequent ACK from the client to complete the connection establishment, reducing the impact of the attack.

6.2.3 Botnets

A botnet is a network of compromised computers, known as bots, that are controlled remotely by an attacker through a command-and-control infrastructure. Botnets are used for various malicious activities, including spamming, phishing, information theft, and conducting large-scale DDoS attacks.

6.2.4 ARP spoofing

The Address Resolution Protocol (ARP) is used to map 32-bit IPv4 addresses to 48-bit MAC addresses. ARP operates through a simple request-reply mechanism: a device sends an ARP request to find the MAC address associated with a given IP address, and the device holding that IP address responds with an ARP reply containing its MAC address.

However, ARP lacks authentication, making it vulnerable to spoofing attacks. In ARP spoofing, an attacker sends forged ARP replies to a network, associating their own MAC address with the IP address of another device, such as the gateway. This can lead to a range of issues including MITM attacks, traffic interception, and DoS.

Countermeasures To counter ARP spoofing, it's essential to verify responses before trusting them, especially if they conflict with existing address mappings. Validate ARP replies before accepting them, particularly if they conflict with known or expected address mappings. This helps ensure that ARP responses are legitimate.

6.2.5 MAC flooding

Content Addressable Memory (CAM) tables are essential components of network switches, used to map MAC addresses to specific ports. This mapping allows switches to efficiently direct traffic only to the appropriate port based on the destination MAC address. However, CAM tables are vulnerable to attacks such as ARP spoofing, which can compromise their integrity.

One common attack on CAM tables is MAC flooding, where attackers use tools to overwhelm the CAM table with a large volume of spoofed MAC addresses. When the CAM table becomes saturated, the switch loses its ability to store MAC address-to-port mappings effectively. As a result, the switch defaults to broadcasting all incoming traffic to every port, similar to the behavior of a hub, leading to reduced network performance and potential security risks.

Countermeasures Implementing port security, a concept often associated with Cisco, can help mitigate MAC flooding attacks. Port security features allow administrators to define and limit the number of MAC addresses that can be learned on a switch port, thus preventing CAM table saturation.

6.2.6 TCP hijacking

TCP session hijacking involves taking over an active TCP connection between two parties. Here's a typical process for executing this attack:

1. *Intercept and monitor*: the attacker (C) intercepts and observes packets exchanged between two parties (A and B), capturing critical information such as sequence numbers.

2. *Disrupt the connection*: the attacker disrupts A's connection, potentially through methods like a SYN Flood attack. This causes A to experience a random or seemingly arbitrary service interruption.

3. *Impersonate and engage*: the attacker, then, impersonates A by spoofing A's IP address and using a correct Initial Sequence Number (ISN) to initiate communication with B. B is unaware that they are now interacting with C instead of A.

Various tools and scripts can automate this attack, making it more efficient. When an attacker is positioned as a MITM, they can inject content into the communication flow without disrupting B's session, allowing them to control or resynchronize all traffic passing through.

Definition (*Man In The Middle*)**.** A Man In The Middle (MITM) attack encompasses various techniques where an attacker impersonates either the server or the client to intercept or manipulate communication between them.

6.2.7 DNS poisoning

The Domain Name System (DNS) is essential for converting domain names into IP addresses. Instead of using a single comprehensive file or hash for mappings, DNS relies on a distributed database system:

- *Hierarchical servers*: the DNS system is structured with a hierarchy of servers, each maintaining a cache of domain-to-IP mappings.
- *UDP communication*: DNS queries and responses use UDP on Port 53 and do not include authentication.
- *Query process*: when a domain name is requested and not found in the local cache, the system queries a series of DNS servers.

Each DNS server in this hierarchy holds resource records that link domain names with IP addresses.

Cache poisoning attack A cache poisoning attack manipulates DNS responses to redirect users to malicious sites. Here's how it typically unfolds:

- *Initiate query*: the attacker sends a recursive query to a target DNS server.
- *Contact authoritative server*: the target DNS server queries the authoritative DNS server for the domain.
- *Spoofed response*: the attacker intercepts or guesses the DNS query ID and sends a forged response, pretending to be the authoritative server.
- *Cache malicious record*: the target DNS server accepts and caches the fraudulent record, believing it to be legitimate.

As a result, any client querying the poisoned DNS server will be redirected to the attacker's malicious website.

6.2.8 DHCP poisoning

The Dynamic Host Configuration Protocol (DHCP) is used to automatically assign IP addresses and network configuration parameters to devices on a network. Here's how DHCP operates:

- *Automatic IP assignment*: when a device connects to the network, DHCP assigns it a new IP address automatically.
- *Centralized management*: network administrators can centrally manage and distribute configuration settings such as IP addresses, router information, and subnet masks.

Despite its convenience, DHCP has some limitations:

- *Lack of authentication*: DHCP does not authenticate requests or responses, making it vulnerable to attacks.
- *UDP dependency*: DHCP operates over UDP, which does not guarantee delivery or order of messages.
- *Server dependency*: DHCP servers must be continuously operational. If a DHCP server becomes unavailable, devices may lose access to network resources.

DHCP poisoning In a DHCP poisoning attack, the lack of authentication is exploited to manipulate network configurations. Here's how the attack typically works:

1. *Intercept requests*: the attacker intercepts DHCP requests from clients on the network.
2. *Spoof responses*: the attacker sends spoofed DHCP responses before the legitimate server can reply.
3. *Manipulate configurations*: through these forged responses, the attacker can assign malicious IP addresses, DNS servers, and default gateways to the victim devices.

As a result, victim clients may receive incorrect network settings, potentially leading to network breaches, traffic interception, or DoS.

6.2.9 ICMP redirect

The Internet Control Message Protocol (ICMP) is used for sending error messages and diagnostic information between network devices, such as hosts and routers. ICMP messages are categorized into three main types: requests, responses, and error messages.

Routing information is primarily managed and updated by routers, not hosts. Routers determine and maintain the best paths for reaching different destinations. Initially, hosts may have limited routing knowledge and rely on routers to learn and update routes.

ICMP redirect An ICMP Redirect message notifies a host that a more efficient route to a specific destination is available and provides the address of the better gateway. The typical process is as follows:

- *Detection*: a router detects that a host is using a suboptimal route for a particular destination.
- *Notification*: the router sends an ICMP Redirect message to the host and forwards the original packet.
- *Update*: the host updates its routing table to use the suggested gateway.

However, attackers can exploit ICMP Redirect messages by sending spoofed redirects. This can hijack traffic and facilitate DoS attacks.

6.3 Firewall

Definition (*Firewall*)**.** A firewall is a network security system designed to monitor and control incoming and outgoing network traffic based on predetermined security rules.

The primary functions of a firewall include:

- *IP packet filtering*: examining and controlling packet flow based on IP address and port number.
- *Network Address Translation* (NAT): modifying IP address information in packet headers to improve network security and efficiency.

A firewall acts as a gatekeeper, enforcing security policies between a protected internal network and external networks.

While firewalls are effective at managing traffic between different networks, they are generally ineffective against insider threats unless the network is properly segmented. They cannot prevent unauthorized activities originating from within the network itself.

Firewalls are essentially specialized computers and may have vulnerabilities of their own. Most firewalls are purpose-built appliances with minimal firmware and limited additional services, reducing their attack surface. They enforce security policies by applying rules that should follow a default-deny approach, where all traffic is denied unless explicitly allowed.

Firewalls can be categorized based on their packet inspection capabilities:

- *Network layer firewalls*: include packet filters and stateful packet filters that operate at the network layer.
- *Application layer firewalls*: include circuit-level gateways and application proxies that operate at the application layer.

6.3.1 Packet filters

Packet filters process individual packets by examining IP and part of the TCP headers. They are stateless, meaning they do not track the state of TCP connections or fully inspect packet payloads. Packet filters, often implemented as Access Control Lists (ACLs) on routers, base their decisions on packet-specific conditions such as IP addresses and port numbers.

6.3.2 Stateful packet filters

Stateful packet filters build upon basic packet filtering by tracking the state of active TCP connections. They ensure packets follow the correct sequence and provide more robust security. While they offer deeper inspection and additional features such as NAT, packet defragmentation, and reassembly, they can also impact performance due to their connection-oriented nature.

6.3.3 Circuit firewalls

Circuit firewalls function as TCP-level proxies. They relay TCP connections by allowing clients to connect to a specific port on the firewall, which then establishes a connection to the desired server on behalf of the client.

6.3.4 Application proxies

pplication proxies operate at the application layer, inspecting, validating, and modifying protocol data. They provide additional features such as user authentication, specific filtering policies, advanced logging, and content filtering. Application proxies may require modifications to clients and servers to function properly.

6.3.5 Multi-zone architecture

Traditional perimeter defense strategies often assume that once external threats are blocked, the internal network remains secure. However, this assumption is challenged by scenarios such as remote access to resources and remote user access to corporate networks. These scenarios can compromise internal network security by mixing externally accessible servers with internal clients.

A robust approach to addressing these security concerns is the implementation of a multi-zone architecture. This approach divides the network into distinct zones based on privilege levels and uses firewalls to control access between them. A key component of this architecture is the creation of a Demilitarized Zone (DMZ) where public-facing servers are placed. The DMZ is considered a high-risk area similar to the internet, and critical data and systems are kept separate from it.

Virtual Private Network Virtual Private Networks (VPN) allow remote employees to securely access corporate resources and connect remote sites without requiring dedicated leased lines. They ensure the confidentiality, integrity, and availability (CIA) of data over public networks by creating encrypted connections. VPNs support two main types of tunneling:

- *Full tunneling*: all traffic is routed through the VPN, which allows comprehensive control over security policies and traffic monitoring.
- *Split tunneling*: only traffic intended for the corporate network is routed through the VPN, while other traffic goes directly to the internet. This method can improve efficiency but reduces control over non-corporate traffic.

6.4 Network security protocols

Communications security must address several critical challenges:

- *Trust and integrity*: establishing trust between parties, protecting sensitive data, and ensuring the atomicity of transactions—ensuring each transaction is complete and accurate.
- *Protocol limitations*: internet protocols often face issues with authentication and confidentiality.
- *Adoption and consistency*: ensuring that security protocols are widely adopted and consistently implemented across different systems and organizations.

Two prominent protocols that tackle these challenges are HTTPS (HTTP over SSL/TLS) and SET (Secure Electronic Transaction).

- *HTTPS*: Ensures the confidentiality and integrity of communications while allowing for mutual authentication. Despite its strengths, HTTPS does not guarantee how data is used nor does it always enforce strict client authentication in practice.
- *SET*: Developed by the VISA and MasterCard consortium, SET focuses on securing transactions rather than just the connections. It ensures data usage and transaction security through mechanisms like dual signatures. However, SET struggled with adoption due to scalability issues and the requirement for cardholders to obtain digital certificates. Today, simpler and more scalable methods, such as redirects with tokens to bank websites, are commonly used.

6.4.1 Transport Layer Security

TLS, the successor to SSL (Secure Sockets Layer), was developed by the IETF. It is designed to ensure:

- *Confidentiality and integrity*: protects data during transmission.
- *Authentication*: provides server and optional client authentication.

TLS uses both symmetric and asymmetric cryptography to balance performance and security. The TLS handshake involves the following steps:

1. *Client hello*: the client sends a list of supported cipher suites and random data to the server.
2. *Server hello*: the server responds with the chosen cipher suite, additional random data, and its certificate. The client must verify this certificate.
3. *Key exchange*: the client sends a pre-master secret encrypted with the server's public key, along with a client certificate if required.
4. *Secure communication*: an encrypted channel is established for communication.

TLS supports a range of algorithms for key exchange, encryption, digital signatures, and hashing. It is designed to be flexible and can adapt to advancements in cryptographic techniques. TLS is inherently resistant to MITM attacks, ensuring that only legitimate parties can decrypt and modify data.

TLS offers robust protection for the confidentiality and integrity of transmitted data and authenticates servers and clients. However, it does not secure data before or after transmission, such as on the server or client side, and it relies on PKI, which can have its own limitations. The effectiveness of TLS depends on the security and trustworthiness of CAs. CAs are responsible for validating domains and organizations and must meet stringent requirements to be included in browser and OS root programs. Removing a non-compliant CA can disrupt many websites, making such decisions complex.

6.4.2 Secure Electronic Transaction

SET, a collaborative effort by VISA and MasterCard, was designed to secure transactions by using dual signatures to link order details sent to merchants with payment data sent to payment gateways. Despite its innovative approach, SET faced scalability challenges and the complexity of requiring digital certificates for cardholders. This hindered its widespread adoption. Modern methods, such as token-based redirects to bank websites, have largely replaced SET for secure transactions.

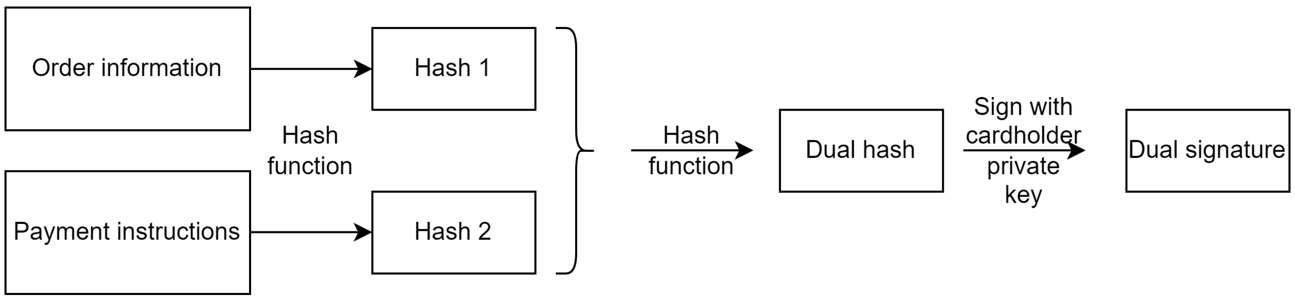


Figure 6.2: Dual signature generation

CHAPTER 7

Malicious software

7.1 Introduction

Malware, a portmanteau of malicious software, refers to code that is intentionally designed to violate security policies. Malware can be categorized into several types, each with distinct characteristics:

- *Viruses*: these are pieces of code that self-propagate by infecting other files, typically executables, but also documents with macros and bootloaders. They are not standalone programs and require a host file to spread.
- *Worms*: these are standalone programs that self-propagate, often remotely. They spread by exploiting vulnerabilities in hosts or through social engineering tactics, such as email worms.
- *Trojan horses*: these are programs that appear benign but conceal malicious functionality. They often enable remote control by an attacker, allowing unauthorized access to the infected system.

7.1.1 History

In detail, the main steps are:

- 1971: Creeper is the first self-replicating program on PDP-10.
- 1981: first outbreak of Elk Cloner on Apple II floppy disks.
- 1983: the first documented experimental virus, as part of Fred Cohen's pioneering work. The term "virus" was coined by Len Adleman.
- 1987: Christmas worm (mass mailer) hits IBM Mainframes, causing 500,000 replications per hour and paralyzing many networks.
- 1988: Internet worm (November 2, 1988) created by Robert Morris Jr., leading to the birth of CERT.
- 1995: Concept virus, the first macro virus, appears.

- 1998: Back Orifice trojan, demonstrating the lack of security in Microsoft systems, is released for the IRC masses.
- 1999: Melissa virus, a large-scale email macro-virus, spreads widely.
- 1999: first DDoS attacks via trojaned machines (zombies) occur.
- 1999: Kernel Rootkits become public with tools like Knark, which modify the system call table.
- 2000: ILOVEYOU worm spreads widely through email, employing social engineering techniques.
- 2001: Code Red worm, a large-scale exploit-based worm, emerges.
- 2003: SQL Slammer worm propagates extremely quickly through UDP.
- 2004: malware creating botnet infrastructures begins to appear, with examples like Storm Worm, Torpig, Koobface, Conficker, and Stuxnet.
- 2010: scareware, ransomware, and state-sponsored malware become more prevalent.

7.1.2 Theory of computer viruses

In 1983, Fred Cohen theorized the existence of computer viruses and produced the first examples. From a theoretical computer science perspective, these viruses represent an intriguing concept of self-modifying and self-propagating code. However, the security challenges posed by viruses quickly became apparent. One significant challenge is the impossibility of creating a perfect virus detector.

Let P be a perfect detection program. We can construct a virus V that includes P as a subroutine:

- If $P(V) =$ True, V halts and does not spread, thus V is not a virus.
- If $P(V) =$ False, V spreads, thus V is a virus.

7.1.3 Infection techniques

Boot viruses target the Master Boot Record (MBR) of the hard disk, which is the first sector on the disk, or the boot sector of partitions. Early examples include the Brain virus, with more recent instances like Mebroot or Torpig. Although considered somewhat outdated, interest in boot viruses is resurging, particularly with the advent of disk less workstations and Virtual Machines.

File infectors, on the other hand, come in several forms. Simple overwrite viruses damage the original program, while parasitic viruses append code and modify the program's entry point. Multi-cavity viruses inject code into unused regions of the program code.

7.1.4 Attackers' motivations

Modern attackers are primarily focused on monetizing their malware. Direct monetization methods include the abuse of credit cards and connecting to premium numbers. Indirect monetization involves information gathering, abuse of computing resources, and renting or selling botnet infrastructures. This has led to the development of a growing underground (black) economy. Within this cybercrime ecosystem, organized groups engage in various activities such as exploit development and procurement, site infection, victim monitoring, and selling exploit kits. They also provide support to their clients.

7.1.5 Antivirus and anti-malware

The basic strategy for antivirus and anti-malware protection is signature-based detection. This method utilizes a database of byte-level or instruction-level signatures that match known malware, often employing wildcards and regular expressions. Heuristics are used to check for signs of infection, such as code execution starting in the last section, incorrect header size in the PE header, suspicious code section names, and patched import address tables. Behavioral detection aims to identify the signs or behaviors of known malware and detect common behaviors associated with malware.

7.2 Macro viruses

Traditionally, data files were considered safe from viruses. However, the advent of macro functionalities introduced a new threat by allowing code to be embedded within these files. For example, spreadsheet macros can perform various actions such as modifying a spreadsheet, altering other spreadsheets, accessing the address book, and even sending emails. One notable and successful instance of this type of virus is the Melissa virus, which proved to be particularly difficult to remove.

7.3 Worms

In November 1988, a program written by Robert Morris Jr., a Ph.D. student at Cornell, brought down the ARPANET. This program, later known as the Morris Worm, was capable of connecting to other computers and exploiting several vulnerabilities, such as buffer overflow in the service and password cracking, to replicate itself onto a second computer. Once the copy was established, it would begin running and repeat the process, causing an infinite loop of self-replication across the network. This unexpected behavior resulted in widespread disruption.

7.3.1 Mass mailers

The introduction of email software that allowed attached files, including executables, led to the emergence of mass mailer worms. These worms could spread by emailing themselves to others, often by exploiting the address book to appear more trustworthy. Modern variations of mass mailers use social networks to spread, such as suspicious-looking messages on Twitter or Facebook from friends.

7.3.2 Mass scanners

Modern worms often use mass scanning techniques to spread rapidly. The basic pattern involves infecting a computer and seeking out new targets, with the potential to spread within minutes and infect hundreds of thousands of hosts. Scanning methods include selecting random addresses, favoring local networks, permutation scanning (dividing up the IP address space), hit list scanning, and combining techniques, as seen with the Warhol worm.

7.3.3 Worm activity and the internet

Despite initial fears that the Internet would be plagued by increasingly sophisticated worms, major worm outbreaks have been rare since 2004. Although vulnerabilities existed and there were times when the community braced for significant impacts, no worm has specifically targeted the Internet infrastructure. However, attackers have typically avoided targeting the infrastructure directly, likely due to their need for the infrastructure to remain operational for their own purposes.

7.4 Bots

The rise of bots began with the abuse of IRC bots, notably during the IRCwars, which included one of the first documented DDoS attacks. In 1999, the trinoo DDoS attack tool emerged, initially running on Solaris and later ported to Windows. Setting up botnets with trinoo was mostly a manual process. In August 1999, a DDoS attack using at least 227 bots targeted a server at the University of Minnesota. The 2000s saw high-profile DDoS attacks against major websites like Amazon, CNN, and eBay, drawing significant media attention.

Definition (*Botnets*)**.** A botnet is a network that consists of several malicious bots controlled by a commander, commonly known as a bot-master or bot-herder.

Botnets pose various potential threats. For the infected host, they can harvest identity, financial, and private data, including email address books and any other type of data present on the victim's machine. For the rest of the Internet, botnets can be used for spamming, DDoS attacks, propagation via network or email worms, and supporting illegal internet activities such as hosting phishing sites and drive-by-download sites.

To defend against malware, several mitigation strategies are employed. Patching is crucial, as most worms exploit known vulnerabilities, though patches are ineffective against zero-day worms. Signature-based detection must be developed automatically because worms spread too quickly for human response. Intrusion or anomaly detection systems can notice fast-spreading and suspicious activity, potentially driving the automated generation of signatures.

7.5 Stealth techniques

7.5.1 Virus

Virus scanners typically detect viruses by searching around the entry point of programs. To evade detection, viruses employ several stealth techniques. One such technique is Entry Point Obfuscation, where viruses, such as multi-cavity viruses, hijack control later in the execution process after the program is launched. This can involve overwriting import table addresses (like libraries) or function call instructions.

Polymorphism is another technique where a virus changes its layout with each infection. The same payload is encrypted using a different key for each infection, making signature analysis practically impossible, although antivirus software might still detect the encryption routine. Metamorphism goes further by creating different versions of the code that appear different but perform the same functions, complicating detection even more.

7.5.2 Malware

Malware often includes a dormant period during which it exhibits no malicious behavior. Its payload can be event-triggered, frequently through a command and control channel. Anti-virtualization techniques are used to evade analysis, as malware often detects if it is running in a virtual environment, which suggests it might be under scrutiny by security labs, antivirus sandboxes, or security specialists.

Modern malware detects execution environments to complicate analysis. It can identify virtual machines, hardware-supported virtual machines, and emulators through timing attacks and environment detection. Encryption and packing techniques are also employed, where malicious content is encrypted, and a small encryption/decryption routine changes the key with each execution. Typical functions of these routines include compressing/decompressing, encrypting/decrypting, incorporating metamorphic components, and implementing anti-debugging and anti-VM techniques.

These advanced techniques make it difficult for antivirus software to detect and analyze complex malware, which often utilizes rootkit techniques to further obscure its presence.

Analysis The process of analyzing suspicious executables typically follows a workflow: first, a suspicious executable is reported, then it is automatically analyzed, followed by manual analysis, and finally, an antivirus signature is developed.

Dynamic analysis involves observing the runtime behavior of the executable. It has the advantage of dealing with obfuscation techniques like metamorphism, encryption, and packing, but it might not cover all the code, particularly dormant code.

Static analysis involves parsing the executable code, which provides comprehensive code coverage and detects dormant code, but it can be hindered by obfuscation techniques such as metamorphism, encryption, and packing.

7.6 Rootkit

Historically, rootkits emerged as a way for attackers to maintain root access on a compromised machine. These tools allow attackers to make files, processes, users, and directories disappear, effectively rendering themselves invisible. Rootkits can operate in either userland or kernel-space.

In userland, rootkits can backdoor login mechanisms and password files. They often trojanize utilities to hide their presence.

Rootkit types Userland rootkits are easier to build but are often incomplete and more easily detected through cross-layer examination and the use of non-trojaned tools. Kernel space rootkits, on the other hand, are more challenging to construct but can completely hide artifacts. These rootkits can only be detected via postmortem analysis.

Syscall hijacking Syscall hijacking involves manipulating the syscall table, the Interrupt Descriptor Table (IDT), or the Global Descriptor Table (GDT).

Advanced rootkits Advanced rootkits extend beyond software to embed themselves in hardware components and firmware. Brossard introduced a rootkit that operates independently of the BIOS. Rootkits can also target the firmware of Network Interface Cards (NICs) or video cards. In virtualization systems, rootkits can act as hypervisors, making detection extremely difficult.

APPENDIX A

The x86 architecture

A.1 Introduction

The Instruction Set Architecture (ISA) serves as the abstract blueprint for a computer architecture, outlining its logical structure. It encompasses essential programming elements like instructions, registers, interrupts, and memory architecture. Importantly, the ISA may deviate from the physical microarchitecture of the computer system in practice.

A.1.1 History

The x86 ISA originated in 1978 as a 16-bit ISA with the Intel 8086 processor. Over time, it transitioned into a 32-bit ISA with the Intel 80386 in 1985. Finally, in 2003, it advanced to a 64-bit ISA with the AMD Opteron processor.

Characterized by its Complex Instruction Set Computing (CISC) design, the x86 ISA retains numerous legacy features from its earlier iterations.

A.1.2 Von Neumann architecture

Von Neumann architecture, named after mathematician and physicist John von Neumann, is a conceptual framework for designing and implementing digital computers. It consists of four main components:

1. *Central Processing Unit* (CPU): this is the brain of the computer, responsible for executing instructions. It contains an arithmetic logic unit (ALU) for performing arithmetic and logical operations, and a control unit that fetches instructions from memory, decodes them, and controls the flow of data within the CPU.

2. *Memory*: Von Neumann computers have a single memory space that stores both data and instructions. This memory is divided into cells, each containing a unique address. Programs and data are stored in memory, and the CPU accesses them as needed during program execution.

3. *Input/Output* (I/O) devices: these devices allow the computer to interact with the external world. Examples include keyboards, monitors, disk drives, and network interfaces.

Data is transferred between the CPU and I/O devices through input and output operations.

4. *Bus*: the bus is a communication system that allows data to be transferred between the CPU, memory, and I/O devices. It consists of multiple wires or pathways along which data travels in the form of electrical signals.

In Von Neumann architecture, programs and data are stored in the same memory space, and instructions are fetched from memory and executed sequentially by the CPU. This architecture is widely used in modern computers and forms the basis for most general-purpose computing devices. However, it has some limitations, such as the Von Neumann bottleneck, where the CPU is often waiting for data to be fetched from memory, leading to inefficiencies in performance.

The memory is structured into cells, with each cell capable of holding a numerical value ranging from -128 to 127.

A.2 Features

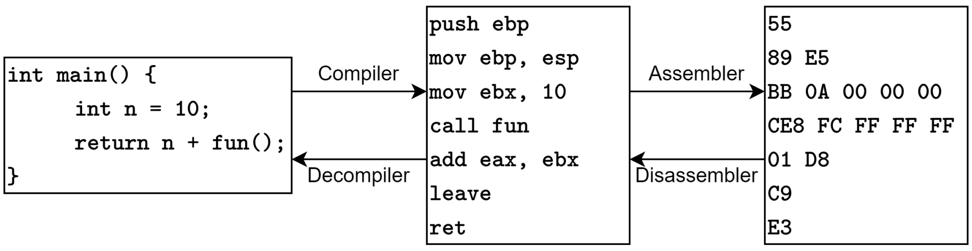
The x86 architecture employs several general-purpose registers, including EAX, EBX, ECX, EDX, ESI, EDI (utilized as source and destination indices for string operations), EBP (serving as the base pointer), and ESP (acting as the stack pointer). Additionally, it incorporates:

- The instruction pointer (EIP) in x86 architecture remains inaccessible directly, but undergoes modification through instructions like `jmp`, `call`, and `ret`. Its value can be retrieved from the stack, known as the saved IP. This register is 32 bits in size and serves as a holder for boolean flags that convey program status, including overflow, sign, zero, auxiliary carry (BCD), parity, and carry. These flags indicate the outcome of arithmetic instructions and play a crucial role in controlling program flow. In terms of program control, the direction flag manages string instructions, dictating whether they auto-increment or auto-decrement. Additionally, EIP controls system operations pertinent to the operating system.
- Program status and control are managed by the EFLAGS register.
- Segment registers are also utilized in the architecture.

The core data types include:

- *Byte*: 8 bits
- *Word*: 2 bytes
- *Dword* (Doubleword): 4 bytes (32 bits)
- *Qword* (Quadword): 8 bytes (64 bits)

Assembly language is unique to each Instruction Set Architecture (ISA) and directly corresponds to binary machine code. The process of converting assembly language instructions into machine code is illustrated in the diagram below:



```
int main() {
      int n = 10;
      return n + fun();
}
```

```
push ebp
mov ebp, esp
mov ebx, 10
call fun
add eax, ebx
leave
ret
```

```
55
89 E5
BB 0A 00 00 00
CE8 FC FF FF FF
01 D8
C9
E3
```

Figure A.1: From source code to machine code

A.3 Syntax

In the x86 architecture, two primary syntaxes are commonly used:

- Intel syntax: This is the default syntax in most Windows programs.
- AT&T syntax: This syntax is default in most UNIX tools.

The Intel syntax is the simpler of the two. Additionally, in x86, instructions have variable length.

A.3.1 Basic instructions

Data transfer Data transfer is accomplished using the following command:

```
mov destination, source
```

Where:

- `source`: immediate value, register, or memory location.
- `destination`: register or memory location.

This command facilitates basic load and store operations, allowing for register-to-register, register-to-memory, immediate-to-register, and immediate-to-memory transfers. It's important to note that memory-to-memory transfers are invalid in every instruction.

Addition and subtraction Addition and subtraction operations are executed using the following commands:

```
// destination = destination + source
add destination, source
// destination = destination - source
sub destination, source
```

Where:

- `source`: immediate value, register, or memory location
- `destination`: register or memory location

It's important to note that the size of the `destination` operand must be at least as large as the `source` operand.

Multiplication Multiplication is performed using the following commands:

```
// destination = implied_op * source
mul source
// signed multiplication
imul source
```

Here, `source` represents a register or memory location. Depending on the size of `source`, the implied operands are as follows:

- First operand: AL, AX, or EAX.
- Destination: AX, DX:AX, EDX:EAX (twice the size of `source`).

Division Division is carried out using the following command:

```
div source
// signed division
idiv source
```

Here, `source` represents a register or memory location. These commands compute both the quotient and remainder. The implied operand for the division operation is EDX:EAX.

Logical operators To perform logical operations such as negation or bitwise operations, the following commands are used: `neg`, `and`, `or`, `xor`, and `not`.

Compare and test To compare two operands or perform bitwise AND operation between them, the following commands are used:

```
// computes op1 - op2
cmp op1, op2
// computes op1 AND op2
test op1, op2
```

These operators set the flags ZF (Zero Flag), CF (Carry Flag), and OF (Overflow Flag) based on the result of the operation but discard the actual result.

Conditional jump Conditional jumps are executed using the following command:

```
j<cc> address or offset
```

This command jumps to the specified address or offset only if a certain condition `<cc>` is met. The condition is checked based on one or more status flags of EFLAGS and can include conditions such as O (overflow), NO (not overflow), S (sign), NS (not sign), E (equal), Z (zero), and NE (not equal).

Other possible jump instructions include:

```
// jump if zero
jz
// jump if greater than
jg
// jump if less than
jlt
```

These instructions allow for conditional branching based on specific conditions evaluated by the processor's status flags.

Unconditional jump Unconditional jumps are executed using the following command:

```
jmp address or offset
```

This command unconditionally transfers control to the specified address or offset by setting the Instruction Pointer (EIP) to the designated location.

The offset can also be relative, causing the EIP to be incremented or decremented by the specified offset value.

Load effective address The load effective address instruction is performed with the following syntax:

```
lea destination, source
```

In this command:

- `source` represents a memory location.
- `destination` denotes a register.

Functionally similar to a `mov` instruction, `lea` doesn't access memory to retrieve a value. Instead, it calculates the effective address of the `source` operand and stores it in the `destination` register, effectively storing a pointer rather than a value.

No operations The `nop` instruction simply advances to the next instruction without performing any operation. Its hexadecimal opcode, `0x90`, is widely recognized. This command holds significant utility in exploitation scenarios.

Interrupts and syscall Interrupts return an integer ranging from 0 to 255. System calls are invoked using the instructions `syscall` in Linux and `sysenter` in Windows.

A.3.2 Conventions

In x86 architectures, a convention known as endianness is employed. This convention dictates the sequential ordering of bytes within a data word in memory.

Big endian Big endian systems store the most significant byte of a word in the lowest memory address.

Little endian Little endian systems store the least significant byte of a word in the lowest memory address.

Note IA-32 architecture follows the little endian convention.

A.4 Program layout and functions

The mapping of an executable to memory in Linux involves several sections:

- `.plt`: this section contains stubs responsible for linking external functions.
- `.text`: this section contains the executable instructions of the program.

- `.rodata`: this section holds read-only data contributing to the program's memory image.
- `.data`: this section holds initialized data contributing to the program's memory image.
- `.bss`: this section holds uninitialized data contributing to the program's memory image. The system initializes this data with zeros when the program starts running.
- `.debug`: this section holds symbolic debugging information.
- `.init`: this section holds executable instructions contributing to the process initialization code. It executes before calling the main program entry point (typically named main for C programs).
- `.got`: this section holds the global offset table.

The program memory layout is depicted in the following simplified diagram:

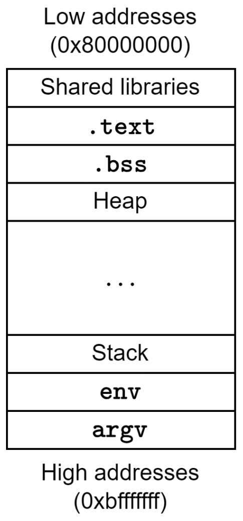


Figure A.2: Simplified program memory layout

A.4.1 Stack

The stack operates on a Last In, First Out (LIFO) principle and is crucial for managing functions, local variables, and return addresses in programs. Its management is facilitated through the use of the ESP register (stack pointer). It's important to note that the stack grows towards lower memory addresses, which means it extends downward in the address space.

Push To insert a new element into the stack, the following command is used:

```
push immediate or register
```

This command places the immediate or register value at the top of the stack and decrements the ESP by the operand size.

Push To remove an element from the stack, the following command is used:

```
pop destination
```

This command loads a word from the top of the stack into the destination and then increases the ESP by the operand's size.

A.4.2 Functions handling

When encountering a `call` instruction, the address of the next instruction is pushed onto the stack, and then the address of the first instruction of the called function is loaded into the EIP register.

Upon encountering a `ret` instruction, the return address previously saved by the corresponding `call` is retrieved from the top of the stack.

At the start of a function, space must be allocated on the stack for local variables. This region of the stack is known as the stack frame. The EBP register serves as a pointer to the base of the function's stack frame. At the function's entry point, the following steps are typically taken:

1. Save the current value of EBP onto the stack.
2. Set EBP to point to the beginning of the function's stack frame.

Upon encountering a `leave` instruction, the caller's base pointer (EBP) is restored from the stack.

Conventions Conventions dictate the method of passing parameters (via stack, registers, or both), the responsibility for cleaning up parameters, the manner of returning values, and the designation of caller-saved or callee-saved registers.

The high-level language, compiler, operating system, and target architecture collaboratively establish and adhere to a specific calling convention, which is an integral part of the Application Binary Interface (ABI).

In x86 C compilers, the declaration conventions are governed by the `cdecl` modifier. Although the `cdecl` modifier can be explicitly used to enforce these conventions, the standard rules dictate that:

- Arguments are passed through the stack in a right-to-left order.
- Parameter cleanup is the responsibility of the caller, who removes the parameters from the stack after the called function concludes.
- The return value is stored in the EAX register.
- Caller-saved registers encompass EAX, ECX, and EDX, while other registers are considered callee-saved.

In x86 C compilers, the calling conventions follow the `fastcall` modifier. While explicitly using the `_fastcall` modifier enforces these conventions, the standard guidelines dictate that:

- Parameters are passed in registers: rdi, rsi, rdx, rcx, r8, and r9, with subsequent parameters passed on the stack in reverse order (caller cleanup).
- Callee-saved registers include rbx, rsp, rbp, r12, r13, r14, and r15.
- Caller-saved registers (scratch) encompass rax, rdi, rsi, rdx, rcx, r8, r9, r10, and r11.
- The return value is stored in rax. If the return value is 128-bit, it's stored across rax and rdx registers.