Computer Security
Theory

Christian Rossi

Academic Year 2023-2024

Abstract

The course topics are:

- Introduction to information security.
- A short introduction to cryptography.
- Authentication.
- Authorization and access control.
- Software vulnerabilities.
- Secure networking architectures.
- Malicious software.

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:

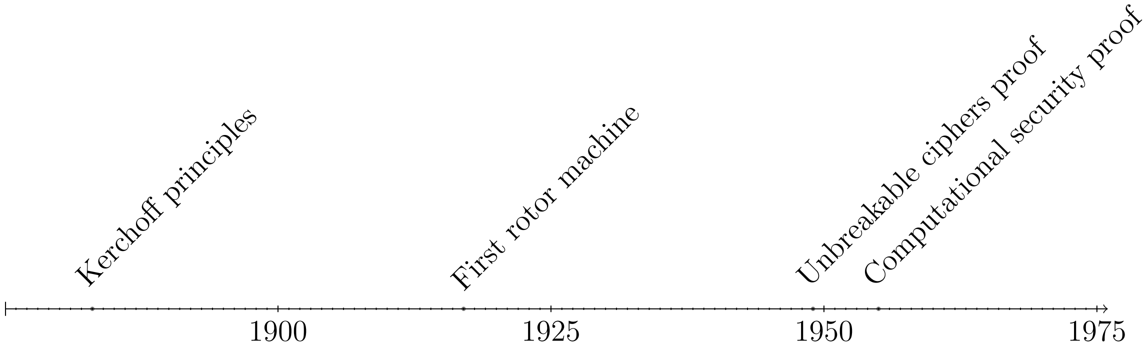
1. The cipher should be practically, if not mathematically, unbreakable.
2. It should be possible to disclose the cipher to the public, including enemies.

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 (λ).



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 Pseudorandom number generators

To expand the key for use in a Vernam cipher with a finite-length key, we require a pseudorandom number generator (PRNG). We assume that the attacker's computational capability is limited to poly(λ) computations.

Definition (*Cryptographically safe pseudorandom number generators*)**.** A cryptographically secure pseudorandom number generator is a deterministic function:

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

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, cryptographic pseudo-random number generators (CSPRNGs) are considered as candidates because there is no conclusive evidence supporting the existence of a definitive pseudo-random number generator (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 Pseudorandom Permutations (PRPs), which are derived from PseudoRandom Functions (PRFs).

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

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

A uniformly randomly sampled $f \overset{\$}{\leftarrow} F$ 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 $F = \{f : \{0,1\}^2 \rightarrow \{0,1\}^1\}$ 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 Pseudorandom function

Definition (*Pseudorandom function*)**.** A pseudorandom function (PRF) is denoted as:

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

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:

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

within polynomial time in λ. In other words, given $a \in \{f : \{0,1\}^{in} \rightarrow \{0,1\}^{out}\}$, 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 $F = \{f : \{0,1\}^{in} \rightarrow \{0,1\}^{out}\}$.

2.3.2 Pseudorandom permutation

Definition (*Pseudorandom permutation*)**.** A pseudorandom permutation is a bijective pseudorandom function defined as:

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

It is characterized solely by its seed value and cannot be distinguished from a random function within poly(λ). 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 pseudorandom permutation because its existence would imply $\mathcal{P} \neq \mathcal{NP}$. Construction of such a pseudorandom permutation 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 Pseudorandom Permutations (PRPs) 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, achieved by:

- Deriving the input corresponding to an output without knowledge of the key.
- Determining the key identifying the PRP or narrowing down plausible options.

- Detecting non-uniformities in their outputs.

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. Modern processor architectures such as ARMv8 and AMD64 include dedicated instructions to accelerate the computation of AES.

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.

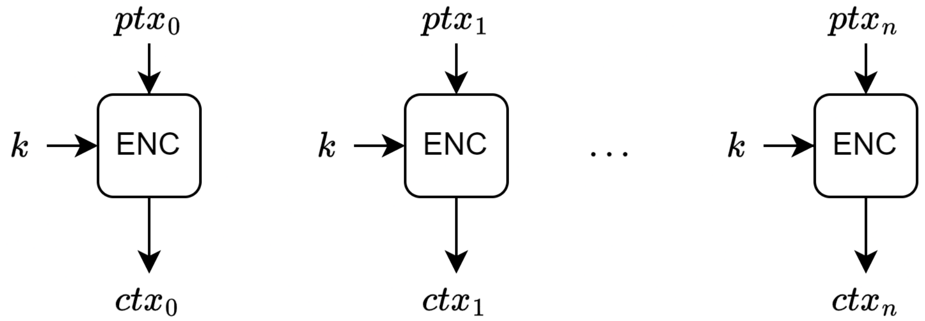


Figure 2.1: ECB encryption mode

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.

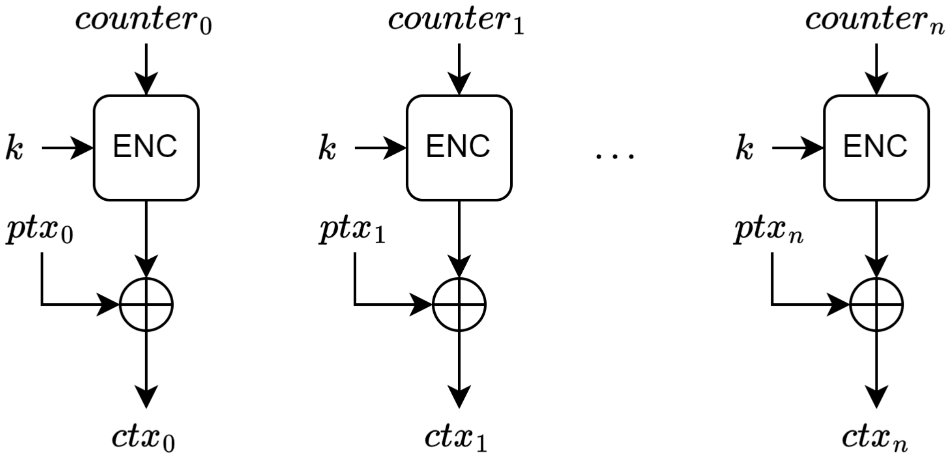


Figure 2.2: CTR encryption mode

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. *Rekeying*: 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 PseudoRandom Permutations (PRPs). 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.

Symmetric ratcheting 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.

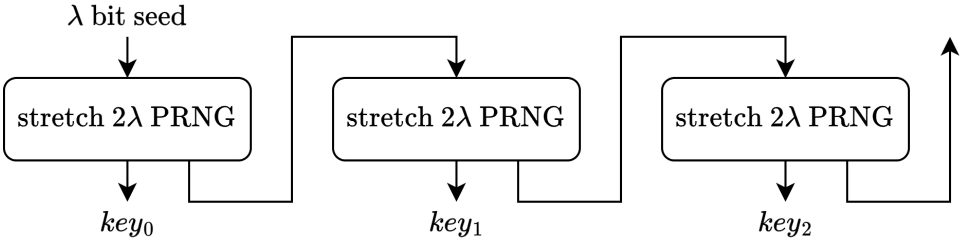


Figure 2.3: Ratcheting

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.

Chosen plaintext attacks 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 Counter (CTR) mode of operation.

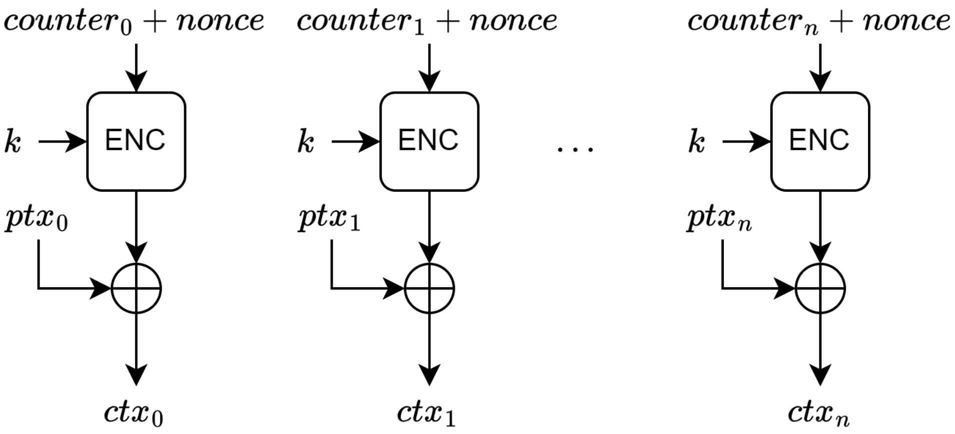


Figure 2.4: Secure ctr

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 (MACs) are required for proper data authentication.

2.5.1 Message authentication codes

A message authentication code 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 message authentication code (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.

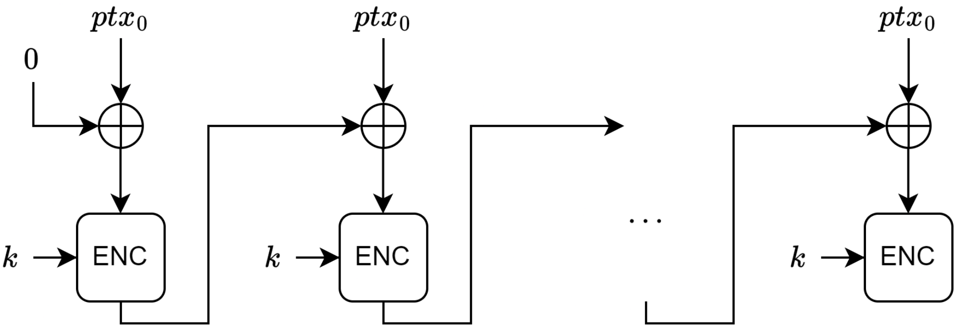


Figure 2.5: CBC encryption mode

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.

MAC usages HTTP cookies serve as a form of note to self for HTTP servers, providing a means to store information locally within a user's browser. However, it is crucial that this information remains unaltered between server reads. To address this concern, the server employs a process where it computes a tag for the cookie using a cryptographic function, denoted as `compute_tag(cookie, k)`. This tag is then stored alongside the corresponding cookie as a pair (cookie, tag), ensuring the integrity and authenticity of the cookie's contents.

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 $H : \{0,1\}^* \rightarrow \{0,1\}^I$, 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 bitstring 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 by bodies such as NIST and ISO.

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 Message Authentication Codes (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. HMAC (Hash-based Message Authentication Code) is a widely adopted method, standardized in RFC 2104 and NIST FIPS 198. It utilizes a generic hash function as a plug-in, denoted as HMAC-hash name. Examples include HMAC-SHA1, HMAC-SHA2, and HMAC-SHA3.
- *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.

Note Until now, the most effective attack method has been enumerating the secret parameter. This approach suffices for modern block ciphers, with the best attack requiring approximately $\mathcal{O}(2^\lambda)$ operations. However, asymmetric cryptosystems depend on complex computational problems, where brute-forcing the secret parameter is not the optimal strategy. For instance, factoring a number of λ bits demands computational effort approximately $\mathcal{O}\left(e^{k(\lambda)^{\frac{1}{3}} \cdot \log(\lambda)^{\frac{2}{3}}}\right)$. It's crucial to note that comparing the sizes of security parameters rather than their actual complexities can lead to misleading conclusions.

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:

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

Alternatively, when utilizing elliptic curve points with dedicated addition, breaking the computational Diffie-Hellman assumption takes $O\left(2^{\frac{\lambda}{2}}\right)$ time.

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.

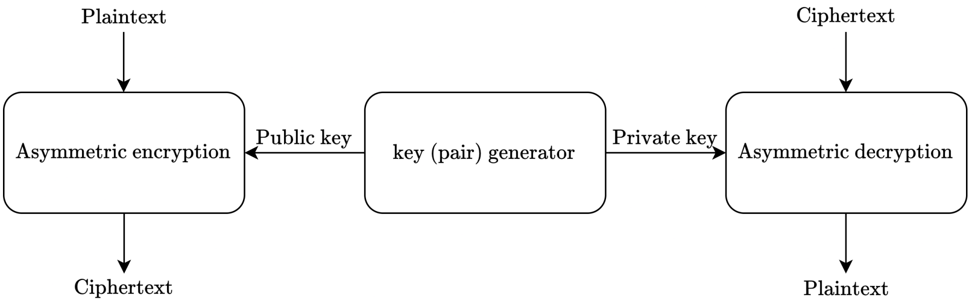


Figure 2.6: Key encryption

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 pseudorandom permutation (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 bitstring 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 bitstring 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.

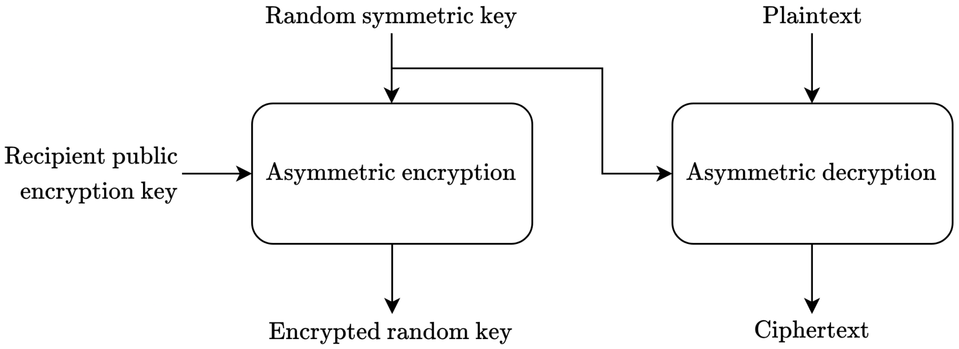


Figure 2.7: Modern encryption

2.6.3 Digital signatures

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.

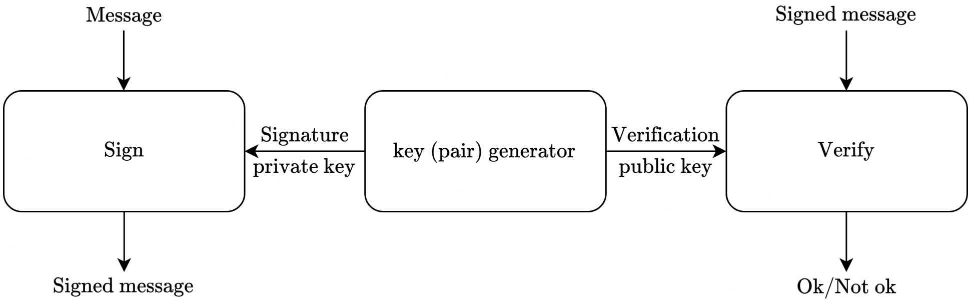


Figure 2.8: Digital signature

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 bitstring, referred to as a challenge. Successful verification of the challenge signature by the client serves as compelling evidence of identity to the server.

2.7 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/identity pairing. To facilitate this process, there is a requirement for a standardized format for distributing these signed pairs securely across systems.

2.7.1 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. One of the most commonly adopted formats for digital certificates is detailed in the ITU X.509 standard.

2.7.2 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.

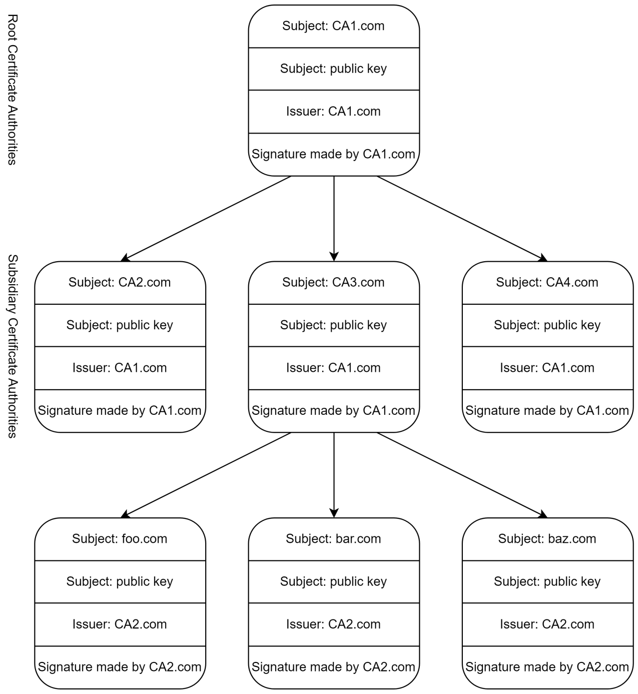


Figure 2.9: Certificate authorities hierarchy

2.8 State of the art

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

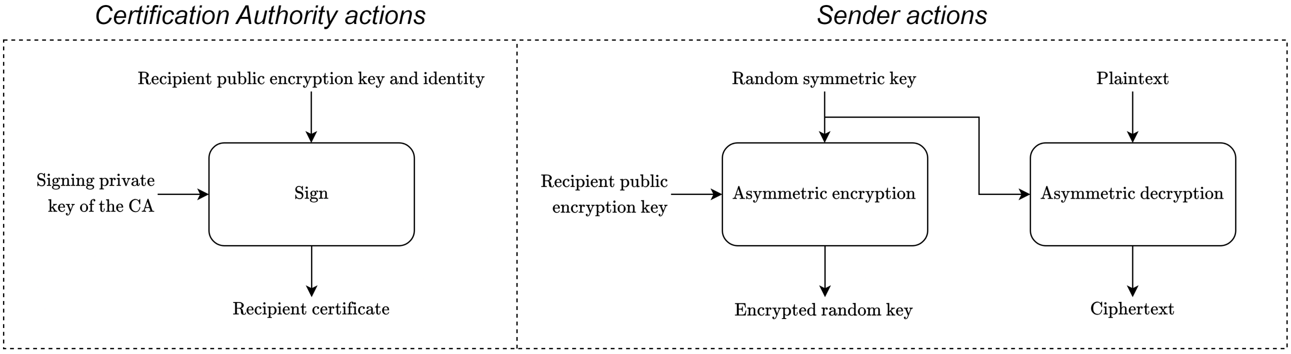


Figure 2.10: Secure communication protocols

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. Instead, there's a growing focus on exploring alternatives, currently undergoing standardization as of April 2022.

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.

2.9 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.

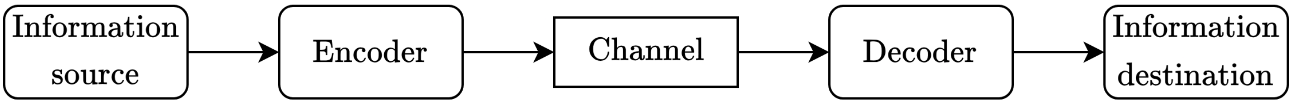


Figure 2.11: Shannon's communication structure

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.

2.9.1 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:

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

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 2.9.1. *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 bitstrings 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 bitstring of length $H(\mathcal{X})$, disregarding momentarily the effort involved in decoding the guess.

2.9.2 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:

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

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 bitstring 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.

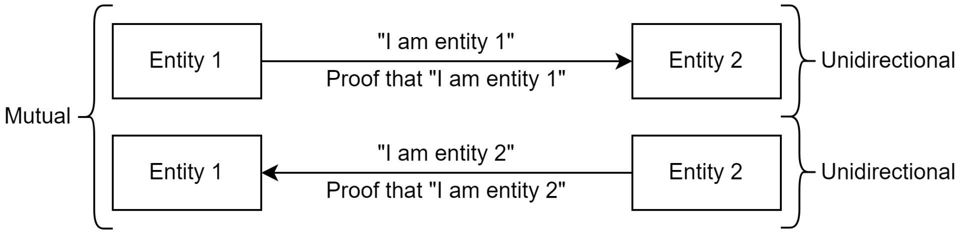


Figure 3.1: Authentication direction

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, PIN, or secret handshake).

2. *Possession-based factors*: items that the entity possesses (e.g., door key, smart card, or token).
3. *Inherent factors*: characteristics that are unique to the entity (e.g., face, voice, or fingerprints).

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

In human-centric scenarios, inherent factors (3) are more commonly utilized than possession-based factors (2), and possession-based factors are more commonly used than knowledge-based factors (1).

In machine-centric scenarios, knowledge-based factors (1) are more frequently employed than possession-based factors (2), and possession-based factors are more commonly used than inherent factors (3).

3.2 Knowledge-based factor

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

Advantages The advantages of passwords are: low cost, ease of deployment, and low technical barrier.

Disadvantages The primary drawbacks include the susceptibility of these secrets to:

- Theft or snooping.
- Guessing by unauthorized individuals.
- Being cracked through various methods.

Countermeasures Potential countermeasures include:

- Regularly changing or expiring passwords to prevent prolonged exposure.
- Utilizing lengthy passwords with a diverse range of characters to enhance complexity.
- Ensuring that passwords are not directly associated with the user to deter predictable patterns.

To determine the most effective countermeasure, it's essential to anticipate the most probable attack in the given scenario. Once identified, prioritize countermeasures that are feasible for users to follow and are likely to mitigate the identified threat effectively.

Countermeasures entail costs due to inherent 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.

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

3.2.1 User education and password complexity

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

When it comes to password complexity, it's 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 Secure password exchange

At its core, authentication involves sharing a secret. To mitigate the risk of secrets being stolen, several strategies can be employed:

- Implement mutual authentication whenever feasible.
- Utilize a challenge-response scheme, which involves exchanging random data to prevent replay attacks.

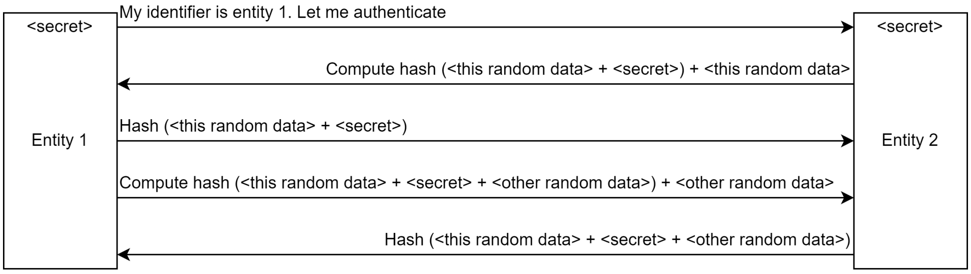

<secret>
Entity 1
<secret>
Entity 2
My identifier is entity 1. Let me authenticate
Compute hash (<this random data> + <secret>) + <this random data>
Hash (<this random data> + <secret>)
Compute hash (<this random data> + <secret> + <other random data>) + <other random data>
Hash (<this random data> +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

Figure 3.2: Countermeasure costs

3.2.3 Secure password storage

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

- Employ cryptographic protection: Ensure that passwords are never stored in clear text. Instead, consider 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 being stored in intermediate storage locations, as this can pose security risks. Regularly review and manage cached data to minimize potential exposure.

3.3 Possession-based factor

The most commonly utilized possession-based factors encompass tokens, smart cards, and smartphones.

Advantages Possession-based offer several advantages, including the human factor (making it less likely to hand out a key), relatively low cost, and a good level of security.

Disadvantages However, passwords come with notable drawbacks, such as being hard to deploy and susceptible to being lost or stolen.

Countermeasures To address these vulnerabilities, potential countermeasures may involve implementing a second factor alongside passwords or exploring alternative authentication methods.

3.3.1 Technologies

The possible solution to implement this solution are:

- One-time password generators operate on the principle of a secret key synchronized with a counter on the host system. The process involves the client computing a Message Authentication Code (MAC) using the counter and key, which is then verified by the host system. The system ensures that the counter matches the expected value, with the counter typically changing every 30 to 60 seconds. Examples of applications utilizing one-time password generators include online banking platforms and administrative consoles like Amazon AWS.

- Smart cards, including those with embedded readers in USB keys, consist of a CPU and non-volatile RAM housing a private key. In the authentication process, the smart card verifies its identity to the host system through a challenge-response protocol. It utilizes the private key to sign the challenge, ensuring the private key remains secure within the device and does not leave it. Smart cards are designed to be tamper-resistant to some degree.

- Static OTP lists consist of sequences known to both the client and the host. The host selects challenges, typically random numbers or specific criteria. The client responds to these challenges, ideally transmitting the response over an encrypted channel for added security. To safeguard the list, the host employs techniques like hashing to avoid storing it in plain text.

- Time-based one-time password software replicates the functionality of password generators. However, a key distinction lies in their implementation:

  - Password generators are typically closed, embedded systems.

  - Password-generation apps operate on general-purpose software and hardware platforms.

3.4 Inherent factor

The most commonly utilized inherent factors encompass Biometric authentication.

Advantages Inherent factors offer several advantages, including a high level of security and the absence of extra hardware requirements.

Disadvantages However, inherent factors come with notable drawbacks, such as being challenging to deploy, probabilistic matching, invasive measurement techniques, susceptibility to cloning, changes in biometric characteristics over time, privacy concerns, and issues for users with disabilities.

Countermeasures To mitigate these vulnerabilities, potential countermeasures may include regularly re-measuring biometric data, securing the biometric authentication process, and providing alternative authentication methods for users who may face difficulties with biometric authentication.

3.4.1 Technologies

Examples of biometric technology include:

- Fingerprint recognition.
- Facial geometry analysis.
- Hand geometry (palm print) recognition.
- Retina scanning.
- Iris scanning.
- Voice analysis.
- DNA analysis.
- Typing dynamics analysis.

Fingerprint Fingerprint authentication involves several steps. First, during enrollment, a reference sample of the user's fingerprint is acquired by a fingerprint reader. From this sample, features are derived, focusing on minutiae such as end points of ridges, bifurcation points, core, delta, loops, and whorls. To enhance accuracy, features from multiple fingers and different positions may be recorded.

These extracted feature vectors are then securely stored in a database.

During authentication, when the user attempts to log in, a new reading of the fingerprint 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, where the system either incorrectly matches the user's fingerprint or fails to recognize it, respectively.

3.5 Single sign on

Single Sign-On (SSO) addresses the complexity of managing and remembering multiple passwords, a common challenge for users. The 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.

The solution offered by SSO is to establish one identity, typically supported by one or two authentication factors, and designate one trusted host. Users authenticate or sign on to this trusted host. Then, other hosts can verify a user's authentication status by querying the trusted host. This streamlined approach simplifies the authentication process for users while maintaining security standards across multiple platforms.

Disadvantages The drawbacks of Single Sign-On (SSO) include having a single point of trust: the trusted server. If this server is compromised, all affiliated sites are compromised as well. Additionally, the password reset scheme needs to be flawlessly secure, with email serving as the trusted component.

Developers often find it challenging to implement SSO correctly. The process involves a complex flow, and while libraries are available to assist, they may contain bugs that introduce vulnerabilities.

3.5.1 Password managers

Dealing with and recalling numerous passwords poses a complex challenge. Users often resort to using the same passwords across various sites, leading to the duplication of password policies.

A potential solution involves adopting a single identity approach, incorporating one or two authentication factors, and relying on a trusted host. Once a trusted host is selected, users authenticate themselves on it using a master password.

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*: it should maintain a secure environment for users and data.
    - *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 confidentiality, integrity, or availability (CIA) of a system is termed an exploit.

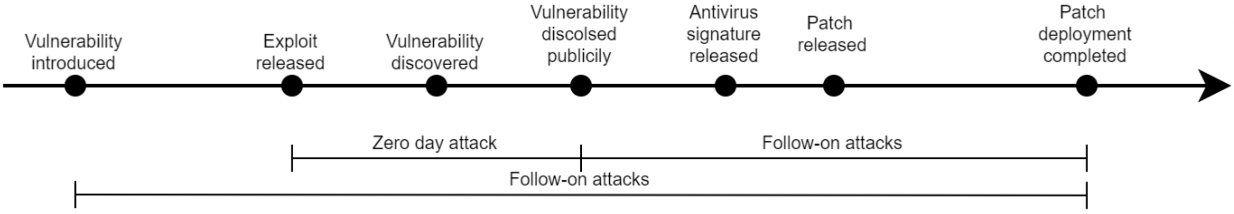


Figure 4.1: Vulnerability lifecycle

4.1.1 Principles of secure design

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, like transitioning from SUID to RUID, 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, as they pose intricate security challenges.

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 like `mktemp` and `opt` 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 such as `/dev/[u]random` to ensure the integrity of cryptographic operations and secure communications.

Summary 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. Additionally, exercising caution with the SUID permission bit is crucial due to its potential security implications.

4.2 Buffer overflow

Binary formats contain vital information regarding the file's organization on disk, memory loading procedures, file type (executable or library), machine class, and sections such as data and code.

ELF binaries ELF (Executable and Linkable Format) binaries adhere to the following structure:

- *ELF header*: this component delineates the overarching structure of the binary. It specifies the file type and demarcates the boundaries for section and program headers.

- *Program headers*: these headers elucidate how the file will be loaded into memory. They segment the data into distinct segments and establish mappings between sections and segments.
- *Section headers*: these headers provide a representation of the binary as it exists on disk. They define 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, i.e., uninitialized data.
  - `.data`: reserved for initialized data.

Definition (*Segment*)**.** Segments represent the runtime view of an ELF (Executable and Linkable Format) 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.

Definition (*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 Process creation in Linux

Program creation When a program is executed, 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. This virtual address space provides isolation and abstraction, allowing each program to have its own memory layout.
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. These segments typically include sections such as code, data, and other resources required for execution.
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 stack is used for managing function calls and local variables, while the heap is utilized for dynamically allocated memory. Additionally, the kernel determines the entry point of the program, where execution should begin, and directs control flow accordingly. Subsequently, the kernel initiates execution by jumping to the designated entry point of the program.

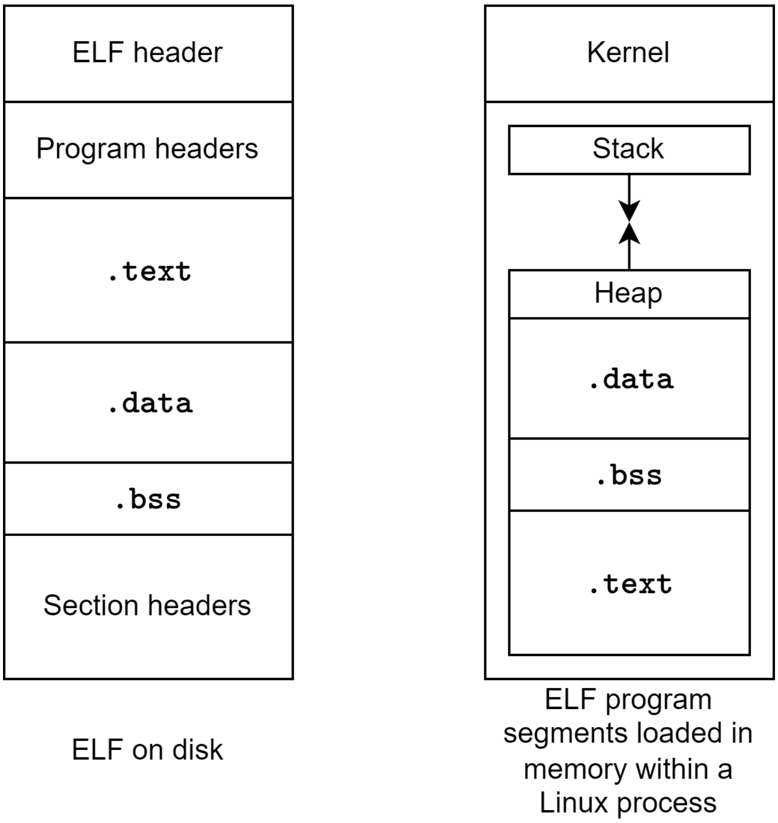


Figure 4.2: Process layout in Linux

Program running When a program is correctly created, the virtual address space contains the following elements:

- *Argc, Env pointer, and stack*: This region encompasses statically allocated local variables, including 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`: initialized data, such as global variables, is stored here.
- `.bss`: this section contains uninitialized data, which is zeroed out when the program begins execution.
- `.text`: the executable code, comprising machine instructions, resides in this segment.
- *Shared libraries.*

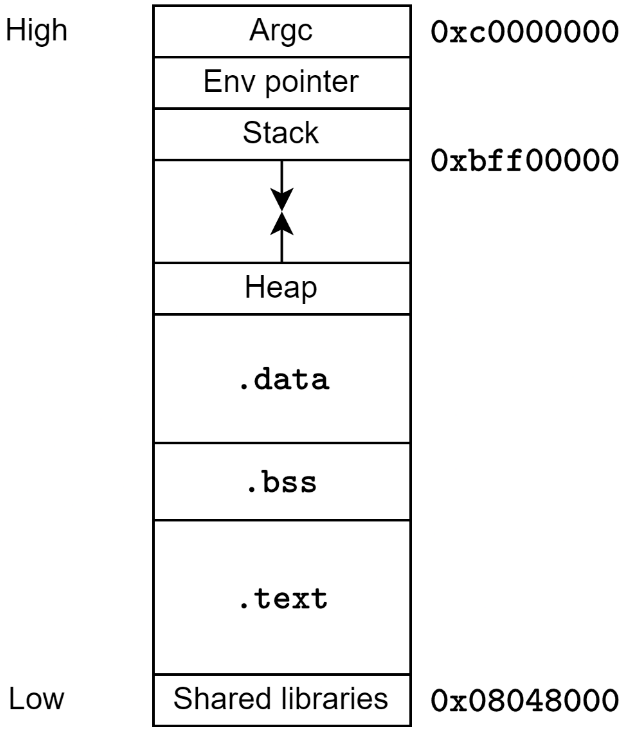


Figure 4.3: Code structure

Program termination When the CPU is about to call the `foo()` function and `foo()` 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 `foo()` function. When `foo()` 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.

This process ensures that the program maintains proper control flow and continues executing subsequent instructions after the completion of the `foo()` function.

Summary When a function is called, its activation record, containing local variables, parameters, and the return address, is allocated on the stack. Control is transferred to the called function, and execution begins from its entry point.

When a function ends, it returns control to the original function caller by jumping back to the return address stored in its activation record. To ensure proper control flow, the CPU saves the return address of the caller's frame on the stack before jumping to the callee. Once the callee's execution is complete, the CPU retrieves the return address from the stack, restoring the caller's frame and continuing execution from where it left off.

Modern compilers typically employ a 16-byte (2^4) stack-boundary alignment by default for performance reasons, especially on certain CPUs.

When compiling with gcc without specifying `-mpreferred-stack-boundary=2` (2^2, or 4 bytes), the resulting code will allocate memory in increments of 16 bytes, even for smaller data types. This means that, for variables or data types smaller than 16 bytes, additional padding will be added to maintain the 16-byte alignment. This alignment strategy optimizes memory access and improves performance, particularly for CPU architectures that benefit from aligned memory access patterns.

4.2.2 Stack smashing

Stack smashing, a term first mentioned in a 1972 report (ESD-TR-7315), gained widespread attention and was popularized by the 1994 article "Smashing the Stack for Fun and Profit" by Aleph1, which is considered a must-read in the field of computer security.

The concept of stack smashing often occurs in C programming due to unsafe practices. For example, consider a function `foo()` that allocates a buffer, such as `char buf[8]`, without performing proper size checking. If this buffer is subsequently filled with data that exceeds its allocated size, it can lead to a buffer overflow vulnerability.

Several standard C library functions are prone to causing stack smashing if not used carefully. These include: `strcpy`, `strcat`, `fgets`, `gets`, `sprintf`, `scanf`.

Failure to properly handle these functions can result in stack smashing vulnerabilities, which attackers may exploit to execute arbitrary code, crash the program, or gain unauthorized access to sensitive information. Therefore, it's essential for programmers to be vigilant and use safer alternatives or apply proper input validation and buffer size checks to mitigate the risk of stack smashing vulnerabilities in their code.

Instead of going out of the stack space we need to jump to a valid memory location that contains, or can be filled with, valid executable machine code. Possible solutions are:

- Environment variable.
- Built-in, existing functions.
- Memory that we can control: the buffer itself or some other variable.

4.2.3 Buffer address guessing

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 located somewhere near the stack pointer (ESP). Debugging tools like GDB (GNU Debugger) can help in determining the approximate location of ESP at runtime.
- *Variability*: unfortunately, the exact address of the buffer may change with each execution or across different machines. This variability makes it difficult to reliably predict the buffer's address.
- *CPU precision*: CPUs are not intelligent enough to handle imprecise memory accesses gracefully. Even a small offset error can lead to failure in fetching and executing instructions, potentially causing the program to crash or exhibit unexpected behavior.

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 location of the buffer.

In practical scenarios, obtaining the ESP (stack pointer) value can be achieved by 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 (first case) might differ by a few words from the ESP value obtained by reading directly within the process (second case).

Despite these methods, a precision issue persists. Even with accurate ESP values, there's still a challenge in precisely determining the location of the buffer. This lack of precision can impede the reliable execution of arbitrary code in a stack overflow scenario. Further solutions to address this precision problem will be discussed in the subsequent slides.

NOP sled To locate the desired address within the buffer, we can utilize a NOP sled. A NOP sled is essentially a sequence of No Operation instructions, represented by the hexadecimal value 0x90 on x86 architecture. These instructions do nothing when executed, serving as a landing strip for the program's execution flow. The idea is that wherever the program falls within the NOP sled range, it will encounter valid instructions, and eventually reach the end of the sled where the actual executable code resides.

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.

In essence, the NOP sled provides a flexible and reliable mechanism for redirecting the program's execution flow to the intended code, overcoming the precision issues associated with predicting the exact memory address of the buffer.

4.2.4 Process to run

Historically, the primary goal of an attacker has been to spawn a privileged shell, either on a local or remote machine.

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

In essence, shell code can perform a wide range of actions, not limited to spawning a shell. However, spawning a shell has been a common objective due to the elevated privileges it provides.

The desired process for an attacker typically involves invoking the `execve` system call. This system call executes a program referred to by its pathname. `execve` is part of a family of system calls required to execute privileged instructions.

Calling convention In Linux systems, invoking a system call involves executing a software interrupt using the `int` instruction, passing the value `0x80`. This triggers the kernel to execute the specified system call, such as `execve`, and perform the requested action:

1. `movl $syscall_number, eax`
2. Syscall arguments GP registers (ebc, ecx, edx):
   (a) `mov arg1, %ebx`

   (b) `mov arg2, %ecx`

   (c) `mov arg3, %edx`

3. `int 0x80`

4. Syscall is executed.

Shell code Creating shell code involves a process that typically starts with high-level code and involves translating it into machine instructions, often using assembly language. However, 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. This often involves cleaning up the code and removing unnecessary instructions.

4. *Extract opcode*: identify the opcode (operation code) for each relevant instruction. The opcode represents the specific operation to be performed by the CPU.

5. *Create the shell code*: finally, assemble the extracted opcodes into a sequence of bytes to form the shell code. This shell code can be injected into a vulnerable program or used in exploits to achieve the desired outcome.

Alternative We demonstrated this capability using an overflowed buffer, but it can also be achieved with other memory regions. The benefit is the ability to execute this remotely (where input equals code). However, there are drawbacks: the buffer may not be sufficiently large, the memory must be designated as executable, and reliable address guessing is necessary.

4.2.5 Defending against buffer overflows

We can apply a layered defense strategy against buffer overflows:

1. *Source code level defenses*: identify and eliminate vulnerabilities within the source code.

2. *Compiler level defenses*: render vulnerabilities non-exploitable through compiler-level techniques.

3. *Operating system level defenses*: implement measures on the operating system level to obstruct attacks, or at the very least increase their difficulty.

Source code level defenses C/C++ programming can be fortified to prevent buffer overflows, with an understanding that such overflows stem from programmer errors. This necessitates educating developers, integrating security measures into the System Development Life Cycle, conducting targeted testing, and employing source code analyzers. Moreover, the adoption of safer libraries can contribute significantly. For instance, utilizing functions from the Standard Library like `strncpy` and `strncat`, which allow specifying the length parameter explicitly, or opting for BSD versions such as `strlcpy` and `strlcat`. Furthermore, languages featuring dynamic memory management, such as Java, inherently offer greater resilience against these vulnerabilities. Integrating such languages into the development process can serve as an additional layer of defense.

Compiler level defenses Warnings issued during compile time serve as an initial line of defense, alerting developers to potential vulnerabilities before execution. Additionally, randomized reordering of stack variables, while considered a temporary measure, can introduce variability, making it harder for attackers to exploit memory vulnerabilities. Moreover, embedding stack protection mechanisms during compilation enhances security. The canary mechanism, for instance, involves inserting a sentinel value between local variables and control values. During the function's epilogue, the integrity of this canary is verified. If tampering is detected upon function return, the program is terminated. There are various types of canaries implemented for enhancing security:

- Terminator canaries are constructed using terminator characters, often /0, which cannot be copied by string-copy functions. This characteristic prevents them from being overwritten during potential attacks.

- Random canaries are generated as random sequences of bytes when the program is executed. This randomness adds an additional layer of defense against buffer overflow attacks. They are commonly implemented using options like `-fstack-protector` in GCC.

- Random XOR canaries, similar to random canaries, are also randomly generated but are additionally XORed with a portion of the structure that requires protection. This XOR operation adds complexity to the canary, making it more difficult for attackers to manipulate even in scenarios where buffer overflow doesn't occur.

Operating system level defenses OS Level Defenses encompass several protective measures:

- Non-executable stack configuration ensures that data is distinguished from executable code, mitigating the risk of stack smashing attacks on local variables. However, it's worth noting that certain programs, such as older versions of the JVM, may require executing code on the stack, posing a challenge to this defense mechanism.

- To enforce non-executable stack policies, hardware mechanisms like the NX bit are leveraged.

- 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. Common variants include return to libc (ret2libc) attacks using C library functions or more sophisticated methods like return-oriented programming (ROP).

- Address Space Layout Randomization (ASLR) further fortifies defenses by repositioning memory elements, including the stack, at random locations during each execution. This randomness makes it challenging for attackers to predict return addresses accurately. ASLR is active by default in Linux kernels newer than 2.6.12, with a typical randomization range of 8MB, configurable through /proc/sys/kernel/randomize_va_space.

4.3 Format string bugs

Definition (*Format string*)**.** A format string provides a solution for crafting output strings that incorporate variables formatted as desired by the programmer.

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

Placeholders serve to specify the formatting type for data inclusion in a string:

- `%d` or `%i`: decimal.
- `%u`: unsigned decimal.
- `%o`: unsigned octal.
- `%X` or `%x`: unsigned hexadecimal.
- `%c`: character.
- `%s`: string (char*), printing characters until the null terminator `/0` is encountered.

Examples of format print functions include: **`printf`**, **`fprintf`**, **`vfprintf`**, **`sprintf`**, **`vsprintf`**, **`snprintf`**, and **`vsnprintf`**.

snprintf() Consider the following call:

```
snprintf(buf, 250, "%x %x %x");
```

When `snprintf()` parses the format string, it anticipates three additional parameters from the caller to substitute for the three `%x` placeholders. As per the calling convention, these parameters should be pushed onto the stack by the caller. Consequently, `snprintf()` assumes they are on the stack prior to the preceding arguments. This situation enables the possibility of unintentionally reading up to fifteen bytes from the stack.

When examining the stack in reverse, we typically encounter a portion of our format string. This is logical since the format string is commonly stored on the stack. Consequently, we can retrieve what we've placed on the stack. To inspect the stack, we employ the `%N$x` syntax, indicating the N-th parameter, coupled with straightforward shell scripting. This method can also be utilized to search for valuable data in memory, representing a vulnerability known as information leakage.

4.3.1 Stack writing

The placeholder `%n` allows us to write the number of characters (bytes) printed so far into the memory address pointed to by the argument. Stack writing can be implemented as follows:

1. Place the address (addr) of the memory cell (target) to be modified onto the stack.
2. Utilize `%x` to locate it on the stack (`%N$x`).
3. Replace `%x` with `%n` to write a number into the cell pointed to by addr, i.e., target.

To ascertain the number of bytes printed thus far, the placeholder `%c` can be employed. This placeholder specifies the precision of the printed number and can also be applied to strings. If you prepend a `0` before the number, it inserts zero padding before the first character.

To write an arbitrary number (0x6028) to the target address (0xBFFFF6CC), we can proceed as follows:

1. Include the target address as part of the format string, pushing it onto the stack.
2. Use `%x` to locate the target address on the stack (`%N$x`). Let's denote the displacement as pos.
3. Employ `%c` and `%n` to write 0x6028 into the cell pointed to by the target address. Remember to consider the parameter of `%c` along with the length of the printed characters.

32 bit addresses To facilitate efficient writing and avoid processing excessive data, we'll split each DWORD (32 bits, up to 4GB) into two WORDs (16 bits, up to 64KB) and execute two writing rounds.

Keep in mind that once we commence counting upwards with `%c`, we cannot count downwards. Our progression must remain in an upward direction, necessitating some mathematical calculations. We first insert the word with the lower absolute value, followed by the word with the higher absolute value.

We must execute this writing procedure twice within the same format string. We need the following:

- Identify the target addresses for the two writes, which are 2 bytes apart.
- Determine the displacements of the two targets.
- Perform mathematical computations to derive the arbitrary numbers to write. These numbers, when summed, should yield the 32-bit address.

Here's the general procedure for performing the writing operation:

1. Place the two target addresses of the memory cells to be modified onto the stack as part of the format string.
2. Utilize `%x` to locate `<target_1>` on the stack (`%N$x`). Let's denote the displacement as pos. For instance, `<target_2>` will be located at pos+1 (i.e., it's positioned one DWORD up).
3. Employ `%c` and `%n` to:

   - Write the lower absolute value into the cell pointed to by `<target_1>`.
   - Write the higher decimal value into the cell pointed to by `<target_2>`.

Target addresses The TARGET address can vary depending on the specific context of the exploit. Here are some common locations where the TARGET address might be found:

- *Saved return address* (saved EIP): this is typical in scenarios resembling a basic stack overflow. You must locate the address on the stack, often by overflowing a buffer and overwriting the return address of a function.
- *Global offset table (GOT)*: used for dynamic relocations of functions in shared libraries. Overwriting entries in the GOT can redirect the execution flow to attacker-controlled code.
- *C library hooks*: functions such as `malloc`, `free`, `printf`, may have hooks that allow customization or interception of their behavior. Overwriting these hooks can alter program behavior.
- *Exception handlers*: overwriting exception handlers can allow control of program execution when exceptions occur.
- *Other structures and function pointers*: depending on the application, there may be other structures or function pointers stored in memory that can be manipulated to redirect program flow.

4.3.2 Countermeasure

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

- Memory error countermeasures, as discussed earlier, play a significant role in preventing exploitation by introducing safeguards like stack canaries, ASLR, and DEP.
- Modern compilers aid in vulnerability mitigation by issuing warnings upon detecting risky calls to printf-like functions during code compilation.
- Patched versions of `libc` (C standard library) are developed to mitigate format string vulnerabilities. For example, some versions count the expected arguments and verify their alignment with the number of placeholders in the format string.

4.3.3 Generalization

The core issue with format string vulnerabilities extends beyond printing functions. Essentially, any function with specific characteristics is at risk:

- Known as variadic functions, they accept a variable number of parameters.
- Parameters are dynamically resolved at runtime by retrieving them from the stack.
- These functions incorporate a mechanism, like placeholders, enabling direct or indirect read/write access to arbitrary memory locations.
- Users have control over these functions, influencing their behavior.

C-like format string interpreters (e.g., printf, sprintf) operate based on a user-specified string. This string can express various functionalities:

- Counters, tracking the number of characters printed.
- Conditional writes to arbitrary memory locations.
- Read operations and arithmetic computations.

These capabilities are sufficient to implement conditional jumps and loops, making the behavior of printf and similar functions Turing complete.

CHAPTER 5

Web application security

5.1 Web application security

Web applications have become the predominant method for delivering software in various contexts. They are commonly utilized within corporate intranets and are prevalent in SAAS and cloud environments. These applications are often designed to be accessible to the public, resembling public web services in their functionality. They are typically built on top of the stateless HTTP protocol, which necessitates the emulation of state management to maintain user sessions and data between requests. Additionally, HTTP's built-in authentication mechanisms are relatively weak, prompting the implementation of additional authentication measures to bolster security.

Client The golden rule in web application security asserts that the client should never be considered trustworthy. Therefore, it is imperative to thoroughly filter and scrutinize any data received from the client. This approach helps mitigate the risks associated with potential malicious inputs or attacks originating from users interacting with the application.

Data filtering Filtering data presents a formidable challenge, but implementing a series of validation techniques can bolster security:

- *Allowlisting*: Authorizing only anticipated data to traverse.
- *Blocklisting*: Additional filtration to exclude recognized malicious content in addition to allowlisting.
- *Escaping*: Transforming special characters into less harmful counterparts.

A foundational principle: allowlisting typically offers greater security compared to blocklisting.

5.2 Cross site scripting

Imagine a basic blog application where anyone can post comments. Users enter their comments into a text field, which is then displayed to other visitors.

Without any filtering mechanism, an attacker could input malicious code like:

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

If this input isn't filtered, the malicious script would execute on subsequent visitors' screens. This type of attack is called Cross-Site Scripting (XSS).

XSS is a vulnerability where client-side code can be injected into a page. There are three types:

1. *Stored XSS*: the attacker's input is stored on the target server in a database. When a victim retrieves the stored malicious code from the web application, it's not made safe to render in the browser.

2. *Reflected XSS*: client input (i.e., request payload) is returned to the client by the web application in a response (e.g., error message, search result). The response includes some or all of the input provided in the request without being stored and made safe to render in the browser.

3. *DOM-based XSS*: user input never leaves the victim's browser. The malicious payload is directly executed by client-side script (e.g., script that modifies/updates the DOM in-memory representation of a page including forms).

With XSS, it's possible to:

- Steal cookies or hijack sessions.
- Manipulate sessions and execute fraudulent transactions.
- Eavesdrop on private information.
- Initiate drive-by downloads.
- Effectively bypass the same-origin policy.

For added security, consider browsing with `noscript` enabled.

Same-Origin Policy The Same-Origin Policy is a standard implemented by all web clients. SOP dictates that all client-side code (such as JavaScript) loaded from one origin, A, should only access data from the same origin, A. However, the modern web often blurs these boundaries with mechanisms like Cross-Origin Resource Sharing (CORS) and client-side extensions.

5.2.1 Content security policy

Content Security Policy is a W3C specification designed to instruct the browser about what content should be trusted and what should not. Think of it as an extension of the same-origin policy, offering more flexibility and expressiveness in defining policies. Technically, CSP consists of a set of directives transmitted from the server to the client via HTTP response headers.

Directives Numerous directives are available in Content Security Policy (CSP), including:

- `script-src`: specifies origins from which client code can be loaded.
- `form-action`: lists valid endpoints for form submission.
- `frame-ancestors`: lists sources permitted to embed the current page as frames and applets.
- `img-src`: defines origins from which images can be loaded.
- `style-src`: similar to script-src, but for stylesheets.

It's important to note that while these directives are specified, their implementation ultimately depends on the browser.

CSP is gradually gaining popularity, but faces challenges due to its potential to disrupt functionality with strict policies and susceptibility to bypassing with relaxed policies.

Practical barriers and challenges include:

- Determining who writes the policies.
- The predominantly manual process of policy creation.
- Limited automation possibilities, leaving some aspects manual.
- Maintenance of policies over time, particularly challenging as modern pages load content from numerous resources, and both pages and resources can evolve dynamically.

5.3 SQL injection

To end a statement in SQL we simply put a semicolon followed by a double dash line that it is used to comment a part of the line. In that way we can modify the query as we desire.

Another way to modify a query is to use the following line where a parameter is requested:

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

In that way the query will be modified resulting in always true.

Unions By using unions we can retrieve data from different tables:

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

Will show contents of a different table. Will work only if the number and the data types of the columns are the same.

5.3.1 Insert injection

On insert we can use the double dash line to comment a section of the statement and change the inserted values. We can also modify the insertion query by adding a sub-query that retrieve some wanted data like passwords and then retrieve the value of the data with another query:

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

5.3.2 Blind injections

Some SQL queries do not display returned values. Rather, they do, or do not do, stuff based on the return value. As a result, we cannot use them to directly display data, but we can play with their behavior to infer data.

5.3.3 Solutions

We can avoid SQL injections in the following ways:

- Input sanitization (validation and filtering).
- Using prepared statements (parametric query) instead of building query strings (if languages allows). In this case we have variable placeholders, that is not a string concatenation. For instance, in PHP we have:

```
$stmt = $db->prepare(SELECT *
                     FROM users
                     WHERE username = ? AND password = ?" )
$stmt -> execute(array($username,$psw));
```

- Not using table names as field names (to avoid information leakage).
- Limitations on query privileges. Different users can execute different types of queries on different tables/DBs, so separate privileges from DB admin point of view.

5.4 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.

5.4.1 Session creation with cookies

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.

5.4.2 Cross-site request forgery

Cross-site request forgery (CSRF) forces a user to execute unwanted actions on a web application in which they are currently authenticated using ambient credentials, such as cookies. In this context, cookies play a crucial role in session management as they are automatically included with every request originating from the browser. This automatic inclusion allows attackers to craft malicious requests that are routed to the vulnerable web application through the victim's browser. As a result, websites cannot distinguish whether the requests from authenticated users are the result of explicit user interaction or not.

Mitigation To mitigate CSRF attacks, implementing a random challenge token is effective. This token is unique and associated with the user's session. It is regenerated at each request and included in forms that involve sensitive operations. The token is sent to the server and compared against the stored token; server-side operations are permitted only if the tokens match. Importantly, these tokens should not be stored in cookies.

Additionally, using the SameSite attribute for cookies helps prevent session cookies from being sent with cross-site requests. By setting the SameSite attribute, websites can specify cookie behavior. When SameSite is set to ”strict,” cookies are not sent for any cross-site usage. When set to ”lax,” cookies are sent for cross-domain navigation but not for cross-site POST forms, images, frames, and similar requests.

5.5 Other issues

5.5.1 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.

5.5.2 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 consist of various physical media, topologies, and protocols. To manage this complexity effectively, a layered approach is essential.

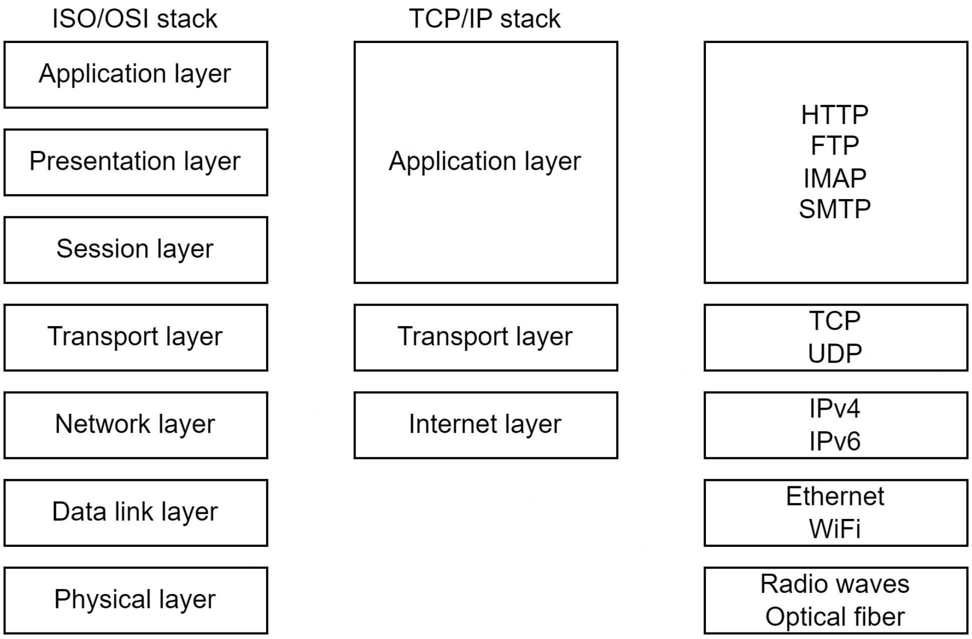


Figure 6.1: Layering and protocols

Addressing In network communications, hosts are uniquely identified by their addresses, akin to phone numbers or postal addresses. Each network layer employs its own addressing scheme:

- *Data link layer*: uses MAC addresses (for Ethernet), a globally unique address embedded in the Network Interface Card (NIC), and the ARP protocol translates an IP address to a MAC address

- *Internet layer*: uses IP addresses, identifies a network host globally, and can include private addresses as defined by RFC 1918 for IPv4.
- *Transport layer*: uses ports and identifies a specific service on a host.

Transport protocols Transport protocols facilitate communication between hosts:

- *UDP* (User Datagram Protocol): connection-less, and lightweight wrapper around an IP packet, primarily adding a port number.
- *TCP* (Transmission Control Protocol): connection-oriented, and manages state through concepts such as closed, open, and established connections. Connections are initiated using a three-way handshake.

6.1.1 Typical network attacks

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 Denial Of Service

Denial of Service (DoS) attacks aim to disrupt the availability of services. Examples include Killer Packets, SYN floods, Smurf attacks, amplification attacks, and Distributed DoS (DDoS).

6.2.1 Killer packets

Ping of Death A pathological ICMP echo request that exploits memory errors in the protocol implementation, causing the target system to crash or behave unpredictably.

Teardrop This attack exploits vulnerabilities in the TCP reassembly process. It sends fragmented packets with overlapping offsets, causing the target system's kernel to hang or crash during reassembly.

Land attack

Land Attack In older systems, such as Windows 95, sending a packet with the source IP equal to destination IP, and the SYN flag set could cause the TCP/IP stack to loop and lock up.

6.2.2 Syn flood attacks

These attacks exploit the TCP three-way handshake. The attacker sends a large volume of SYN requests with spoofed source addresses, filling up the queue with half-open connections. As a result, SYN requests from legitimate clients are dropped.

Mitigation SYN cookies mitigate this attack by replying with a SYN+ACK while discarding the half-open connection, waiting for a subsequent ACK to establish a connection.

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

6.2.3 Smurf

The attacker sends ICMP packets with a spoofed sender address (victim) to a network's broadcast address, causing a flood of responses to overwhelm the victim.

6.3 Sniffing

Normally, a network interface card (NIC) intercepts and forwards to the operating system only the packets addressed to the host's IP.

In promiscuous mode, the NIC passes any packet it reads off the wire to the operating system, regardless of the packet's destination address.

Originally, Ethernet networks operated on a shared medium, using BNC cables. Although RJ-45 cables changed the physical connector, the medium remained shared due to the use of hubs, which broadcast traffic to every host within the broadcast domain.

In contrast, switches selectively relay traffic only to the specific wire corresponding to the correct NIC, based on the Ethernet address. This selective relaying is primarily for performance enhancement, not for security reasons.

6.3.1 ARP spoofing

Address Resolution Protocol (ARP) is responsible for mapping 32-bit IPv4 addresses to 48-bit hardware, or MAC, addresses. RP works with a simple request-reply mechanism with a request and a reply. However, ARP lacks authentication, making it susceptible to spoofing. An attacker can easily forge ARP replies, leading to a lack of trust in the received information. Each host caches these replies, which can be viewed using the command `arp -a`.

Mitigation To counter ARP spoofing, it's essential to verify responses before trusting them, especially if they conflict with existing address mappings. One approach is to add a sequence or ID number in the request to help validate the responses.

6.3.2 CAM tables

CAM tables are integral to switches as they cache the mapping of MAC addresses to specific ports. Switches are susceptible to ARP spoofing attacks, compromising the integrity of CAM tables. A CAM table can be saturated in the following way:

- Switches rely on CAM tables to efficiently direct traffic based on MAC addresses.
- Tools like Dsniff (macof) can inundate switches with a massive volume of spoofed packets, saturating the CAM table within seconds (known as MAC flooding).

- When the CAM table reaches full capacity, the switch can no longer effectively cache ARP replies. Consequently, it resorts to broadcasting all incoming traffic to every port, resembling the behavior of a hub.

Mitigation Implementing PORT Security, a term coined by CISCO, helps mitigate these vulnerabilities.

6.3.3 Spanning Tree Protocol

The Spanning Tree Protocol (STP) 802.1d is crucial for preventing loops in switched networks by constructing a spanning tree (ST). Switches determine the structure of the ST through the exchange of Bridge Protocol Data Unit (BPDU) packets to elect the root node.

However, since BPDU packets are not authenticated, attackers can manipulate the topology of the spanning tree for malicious purposes such as sniffing or ARP spoofing.

6.3.4 Spoofing

UDP spoofing In UDP (User Datagram Protocol), the IP source address lacks authentication, making it susceptible to modification in UDP or ICMP packets. This process is relatively simple for attackers. However, if the attacker is on a different network, they won't receive responses to their spoofed packets, as these responses will be directed to the spoofed host (known as blind spoofing). However, if the attacker is on the same network, they can sniff the remaining traffic or employ ARP spoofing to further exploit vulnerabilities.

TCP spoofing Unlike UDP, TCP (Transmission Control Protocol) employs sequence numbers for packet reordering and acknowledgment. It utilizes a semi-random Initial Sequence Number (ISN) during the connection setup. If a blind spoofer can predict the ISN, they can complete the TCP three-way handshake blindly without receiving responses. However, it's crucial that the spoofed source does not receive response packets. Otherwise, it might respond with a Reset (RST) packet, potentially disrupting the connection.

6.3.5 TCP session hijacking

TCP session hijacking involves seizing control of an active TCP session. Here's how it typically occurs:

1. The attacker (C) intercepts and monitors packets between two parties (A and B), recording sequence numbers.

2. C disrupts A's connection, such as by launching a SYN Flood attack, causing A to experience a seemingly random disruption in service.

3. C assumes A's identity by spoofing A's address and using a correct Initial Sequence Number (ISN) to engage in dialogue with B. B remains unaware of the deception.

Numerous tools, like hunt and dsniff, automate this attack process. If the attacker is a man in the middle, they can inject content into the flow without disrupting B's session, thereby controlling or resynchronizing all traffic passing through.

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

6.3.6 Domain Names System

The Domain Name System (DNS) is responsible for translating domain names into numerical IP addresses. Rather than employing a hash or a file of all mappings, DNS operates through a distributed database system:

- A hierarchy of servers maintains mappings, with each server retaining a small cache of these mappings.
- DNS operates over UDP on Port 53, with messages lacking authentication.
- When a domain name is requested and not found in the local cache, the system queries a DNS server.

This hierarchy of DNS servers contains resource records to match domain names with IP addresses.

Domain name resolution Consider the example of typing ”sports.polimi.com”:

- he system checks /etc/hosts and the DNS cache.
- If not found locally, it traverses the hierarchy, querying DNS servers from the root to the authoritative server of ”polimi.com”.
- Finally, it sends an HTTP request to the obtained IP address.

When a non-authoritative DNS server receives a resolution request, it may:

- Respond from cache if available.
- Recursively resolve the name on behalf of the client.
- Iteratively provide the authoritative DNS address.

Cache poisoning attack Here’s how a cache poisoning attack unfolds:

1. The attacker initiates a recursive query to the victim DNS server.
2. The victim DNS server contacts the authoritative server.
3. Impersonating the authoritative DNS server, the attacker sniffs or guesses the DNS query ID and spoofs the answer.
4. The victim DNS server, trusting the response, caches the malicious record.

Subsequently, all clients redirected to the poisoned DNS server are led to the malicious website.

6.3.7 Dynamic Host Configuration Protocol

The Dynamic Host Configuration Protocol (DHCP) is a protocol designed to dynamically allocate IP addresses and network parameters to devices within a network. Here's how it functions:

- DHCP automatically assigns a new IP address to a computer when it connects to the network.
- It enables network administrators to centrally manage and distribute configuration parameters to network hosts, including IP addresses, router information, and subnet masks.

Limitations However, DHCP lacks authentication, potentially impacting performance. It operates over UDP, and a DHCP server must run continuously. If the DHCP server becomes unavailable, clients may lose access to the enterprise network.

DHCP Operations DHCP involves several key operations:

1. DHCP DISCOVER: Client requests an IP address.
2. DHCP OFFER: DHCP server offers an IP address to the client.
3. Upon receiving DHCP ACK from the server, the client can begin using the assigned IP address.
4. DHCP RELEASE: Client voluntarily relinquishes the IP address.

DHCP poisoning attack Since DHCP is unauthenticated, attackers can intercept DHCP requests, respond first, and manipulate client configurations. Through spoofed DHCP responses, attackers can set IP addresses, DNS addresses, and default gateways for victim clients, leading to potential network breaches.

6.4 ICMP redirect attack

ICMP (Internet Control Message Protocol) is utilized for sending debugging information and error reports between hosts, routers, and other network devices at the IP level. ICMP messages fall into three categories: requests, responses, and error messages. Key ICMP functions include:

- *Echo request/reply*: used to test connectivity (ping).
- *Redirect*: used to inform hosts about more efficient routes (gateways).

Route change requests Routers, not hosts, are responsible for maintaining and updating routing information. They are expected to discover and manage the best routes for every destination. Initially, hosts start with minimal routing information and gradually learn new routes from routers. A host may boot up knowing the address of only one router, which might not always be the most optimal route.

6.4.1 ICMP redirect

An ICMP Redirect message informs a host that a better route exists for a given destination and provides the gateway for that route. When a router detects a host using a suboptimal route, it:

- Sends an ICMP Redirect message to the host and forwards the original packet.
- The host is expected to update its routing table accordingly.

An attacker can exploit this by forging a spoofed ICMP Redirect packet to re-route traffic to a specific route or a particular host, which may not be a legitimate router. This attack can:

- Hijack traffic, directing it through the attacker's computer.
- Facilitate a denial-of-service (DoS) attack.

Weak authentication An ICMP message includes the IP header and a portion of the payload (usually the first 8 bytes) of the original IP datagram. To forge a convincing ICMP Redirect message, an attacker must intercept a packet in the original connection, necessitating their presence on the same network. This creates a half-duplex man-in-the-middle (MITM) situation. Handling of ICMP Redirects varies by operating system.

Route mangling If an attacker can announce routes to a router, they can manipulate routing in various ways. Common routing protocols with no or weak authentication include:

- IGRP (Interior Gateway Routing Protocol).
- RIP (Routing Information Protocol).
- OSPF (Open Shortest Path First).

More secure protocols like EIGRP (Enhanced Interior Gateway Routing Protocol) and BGP (Border Gateway Protocol) offer authentication options, but these are often underutilized.

6.5 Firewall

Definition (*Firewall*)**.** A firewall is a network access control system that inspects and verifies all packets flowing through it.

The main functions of a firewall typically include:

- IP packet filtering.
- Network Address Translation (NAT).

A firewall must serve as the single enforcement point between a screened network and outside networks.

Insider attacks Firewalls inspect all traffic passing through them but are powerless against insider attacks unless the network is partitioned. They are generally ineffective against unchecked paths within the network.

Firewall rules Firewalls are essentially computers and can have vulnerabilities. Most firewalls are single-purpose machines, often embedded appliances with minimal firmware that offer few or no additional services, resulting in a smaller attack surface. Firewalls act as bouncers at the network boundary, enforcing security rules. Ineffective rules result in inadequate protection. Firewall rules implement higher-level security policies, which should be based on a default-deny approach.

Taxonomy Firewalls are categorized based on their packet inspection capabilities, from lower to higher layers:

- *Network layer firewalls*: packet filters and stateful packet filters.
- *Application layer firewalls*: circuit-level gateways and application proxies.

6.5.1 Packet filters

Packet filters process packets individually, decoding the IP and part of the TCP headers. They are stateless and cannot track TCP connections or fully inspect payloads. Often found on routers as Access Control Lists (ACLs), packet filters predicate actions (block, allow, log, etc.) based on packet conditions.

6.5.2 Stateful packet filters

Dynamic packet filters enhance standard packet filters by tracking the state of TCP connections, ensuring the correct sequence of packets (e.g., SYN followed by SYN-ACK). This tracking improves the safety of deny rules but can impact performance as it is connection-based rather than packet-based. Dynamic filters can also perform deeper content inspection, reconstruct application-layer protocols, and offer features like NAT, packet defragmenting, and reassembly to handle fragmented packets.

Session handling A session is an atomic, transport-layer exchange of application data between two hosts. Main transport protocols include:

- TCP (Transmission Control Protocol): the session is the TCP connection itself.
- UDP (User Datagram Protocol): connectionless, but sessions can be inferred for NAT and control purposes. UDP is widely used (e.g., DNS, VoIP, video streaming) despite its lack of connection orientation, making it difficult to secure.

Application layer Application-layer inspection is crucial for protocols that transmit network information. Stateful firewalls must manage these protocols to ensure security.

6.5.3 Circuit firewalls

Circuit firewalls relay TCP connections, acting as a TCP-level proxy. The client connects to a specific port on the firewall, which then connects to the desired server.

6.5.4 Application proxies

Application proxies operate at the application layer, inspecting, validating, and manipulating protocol data (e.g., rewriting HTTP frames). They can authenticate users, apply specific filtering policies, perform advanced logging, and offer content filtering or scanning (e.g., antivirus, spam filtering). Proxies may require client modifications and specific protocol servers.

6.6 Multi-zone architecture

Perimeter defense typically assumes internal network safety while keeping external threats out. However, scenarios like remote access to resources (e.g., web servers, FTP, mail transfer) and remote user access to corporate networks challenge this assumption. Mixing externally accessible servers with internal clients reduces internal network security.

A practical solution is a multi-zone architecture, splitting the network by privilege levels and using firewalls to regulate access. This involves creating a DMZ (demilitarized zone) for public servers (web, FTP, DNS, SMTP) while keeping critical data away from the DMZ, which is considered as risky as the internet.

6.6.1 Virtual Private Network

VPNs enable remote employees to work as if they were in the office and connect remote sites without dedicated lines, ensuring confidentiality, integrity, and availability (CIA) of data over public networks. VPNs create encrypted overlay connections over public networks, supporting full and split tunneling:

- *Full tunneling*: all traffic goes through the VPN, allowing complete control and application of security policies.
- *Split tunneling*: only traffic to the corporate network uses the VPN; other traffic goes directly to the ISP, improving efficiency but reducing control.

VPN technologies The possible technologies for a VPN are:

- PPTP (Point-to-Point Tunneling Protocol): a proprietary Microsoft protocol, an extension of PPP with added authentication and confidentiality.
- TLS-Based VPNs: using pure TLS, SSH tunnels, or OpenVPN (TLS-based).
- IPsec: provides security extensions for IPv6, backported to IPv4, offering authentication and confidentiality at the IP layer.

6.7 Network security protocols

Issues of communications security Communications security faces several challenges. Firstly, problems of remoteness include the need to establish trust between parties, protect sensitive data, and ensure the atomicity of transactions, meaning that each transaction must be complete and accurate. Internet protocols also present issues, particularly around authentication and confidentiality. Additionally, there is the transparency and critical mass problem, which involves ensuring that security protocols are widely adopted and consistently used.

Secure protocols Two notable protocols that have addressed these challenges are HTTPS (HTTP over SSL/TLS) and SET (Secure Electronic Transaction). HTTPS ensures communication confidentiality and integrity, and allows for mutual authentication. However, it does not guarantee how data is used and often does not strictly authenticate clients in practice. SET, developed by the VISA and MasterCard consortium, focused on securing transactions rather than connections, ensuring data usage and transaction security. Despite these strengths, SET failed to achieve critical mass and widespread adoption.

6.7.1 Transport Layer Security

SSL, originally designed by Netscape to secure web communication, became a de facto standard and evolved into TLS, standardized by the IETF. TLS, which succeeded SSL v3 and is currently at version 1.3, ensures the confidentiality and integrity of communications, server authentication, and optional client authentication. It uses both symmetric and asymmetric cryptography to balance performance and security. Versions up to TLS 1.1 are now considered insecure. The TLS handshake involves several steps to establish a secure connection:

- Client hello: the client sends a list of supported cipher suites and random data to the server.
- Server hello: the server responds with selected cipher suite, additional random data, and its certificate. The client must verify the certificate.
- Key exchange: the client sends a pre-master secret encrypted with the server's public key and a client certificate signed with its private key.
- Secure communication: an encrypted communication channel is established.

TLS is designed to be flexible, supporting different suites of algorithms for key exchange, symmetric encryption, digital signatures, and hash functions. During the handshake, clients and servers compare cipher suites to agree on shared algorithms. This flexibility ensures that TLS can adapt to technical advancements.

TLS is inherently resistant to man in the middle attacks, as it ensures that only the legitimate parties involved in the communication can decrypt and modify the data.

Summary TLS offers several benefits, including the protection of transmission confidentiality and integrity and the authentication of servers and, optionally, clients. However, it does not provide protection before or after transmission, such as on the server, client, or against abusers. Additionally, TLS relies on the Public Key Infrastructure (PKI), which introduces some limitations.

6.7.2 Limitations and mitigations

The effectiveness of TLS depends on the security and trustworthiness of root and intermediate Certificate Authorities (CAs). CAs are responsible for domain and organization validation and must meet specific requirements to be included in browser and OS root programs. Removing a non-compliant CA can disrupt many websites, making it a difficult decision. To address some limitations, several mechanisms have been developed:

- HSTS (HTTP Strict Transport Security) uses an HTTP header to instruct browsers to always connect to a domain using HTTPS, with some browsers maintaining HSTS preload lists to enforce this.
- HPKP (HTTP Public Key Pinning), though now deprecated, allowed browsers to pin specific certificates or CAs to prevent trust issues from mis-issued certificates.
- Certificate Transparency requires CAs to log metadata of issued certificates to an independent log, which browsers can enforce to defend against certificate mis-issuance.

6.7.3 Secure Electronic Transaction

SET was a collaborative effort by VISA and MasterCard to protect transactions rather than connections. The cardholder would send order details to the merchant and payment data to the payment gateway, which would verify the correspondence using a dual signature that linked the two pieces of a message directed to different recipients. Despite its innovative approach, SET failed due to scalability issues, particularly the need for cardholders to obtain digital certificates, which hindered widespread adoption. Today, simpler methods, such as redirects with tokens to bank websites, are commonly used for secure transactions.

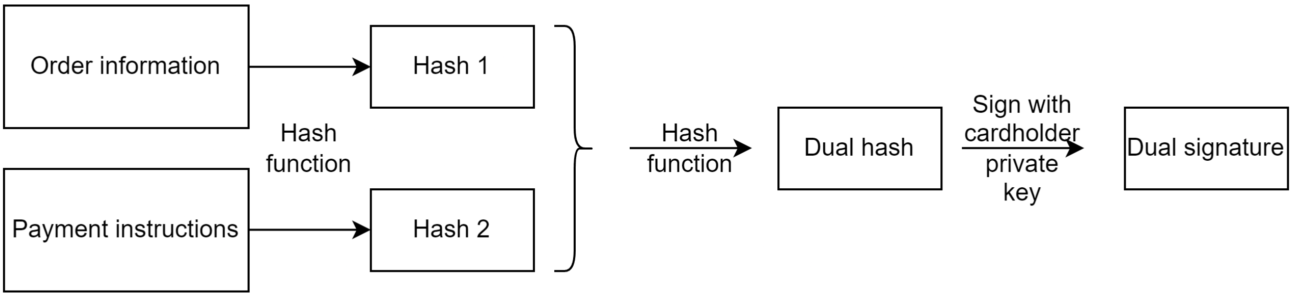


Figure 6.2: Dual signature generation

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 boot loaders. 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/Torpig. Although considered somewhat outdated, interest in boot viruses is resurging, particularly with the advent of diskless workstations and virtual machines, such as SubVirt.

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 (C&C) 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, sshd, and passwd files. They often trojanize utilities to hide their presence, such as ps, netstat, ls, find, du, who, w, finger, and ifconfig in Linux environments. In Windows, userland rootkits typically target Task Manager, Process Explorer, Netstat, and ipconfig.

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 post-mortem analysis. The first complete Linux Kernel Module (LKM) rootkit was presented by plaguez in 1998.

Syscall hijacking Syscall hijacking involves manipulating the syscall table, the Interrupt Descriptor Table (IDT), or the Global Descriptor Table (GDT). For instance, after the syscall table was hidden in kernel 2.6, attackers began scanning the IDT for a FAR JMP instruction pointing to the syscall table address. Other methods include detour patching and direct patching through /dev/mem or /dev/kmem, as demonstrated by Silvio Cesare, showing it was feasible even with monolithic kernels.

Advanced rootkits Advanced rootkits extend beyond software to embed themselves in hardware components and firmware. Brossard introduced a bootkit 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.

7.7 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, sshd, and passwd files. They often trojanize utilities to hide their presence, such as ps, netstat, ls, find, du, who, w, finger, and ifconfig in Linux environments. In Windows, userland rootkits typically target Task Manager, Process Explorer, Netstat, and ipconfig.

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 post-mortem analysis. The first complete Linux Kernel Module (LKM) rootkit was presented by plaguez in 1998.

Syscall hijacking Syscall hijacking involves manipulating the syscall table, the Interrupt Descriptor Table (IDT), or the Global Descriptor Table (GDT). For instance, after the syscall table was hidden in kernel 2.6, attackers began scanning the IDT for a FAR JMP instruction pointing to the syscall table address. Other methods include detour patching and direct patching through /dev/mem or /dev/kmem, as demonstrated by Silvio Cesare, showing it was feasible even with monolithic kernels.

Advanced rootkits Advanced rootkits extend beyond software to embed themselves in hardware components and firmware. Brossard introduced a bootkit 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 Instruction Set Architecture (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

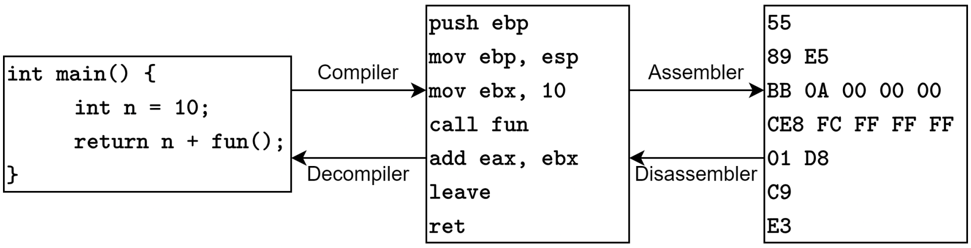
The x86 architecture employs several general-purpose registers, including EAX, EBX, ECX, EDX, ESI, EDI (utilized as source and destination indices for string operations), EBP (serving as the base pointer), and ESP (acting as the stack pointer). Additionally, it incorporates:

- 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:

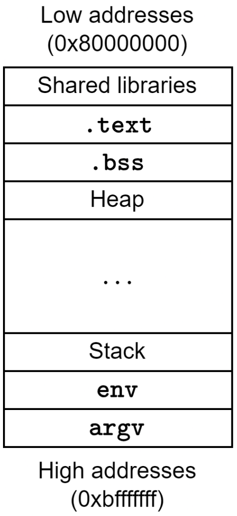


Figure A.2: Simplified program memory layout

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.