Computer Security

Niccoló Didoni

February 2022

Contents

Part I

Introduction

Chapter 1

Basic concepts

Computer security is a bit different from other subjects, in fact it is based on adversarial behaviours and it doesn't have a book on which we can find how to do things. More precisely, when securing a system we have to take into account that we are working against an attacker that tries to compromise the system. The problem is that we don't have a manual that says how to secure a system, but we only know what we shouldn't do.

1.1 Security paradigm

1.1.1 CIA paradigm

Computer security is based on the CIA paradigm. A system is secure if it satisfies the properties (i.e. Confidentiality, Integrity and Availability) specified by the CIA paradigm. In particular such properties are

- **Confidentiality**. Confidentiality means making sure that the information that needs to be available to a person, is only available to that person and to no one else. In other words, data should be accessed only by those authorised. This properties specifies if data can or can't be read.
- **Integrity**. Integrity means making sure that information is accessed only by those authorised (as defined by confidentiality) and only in the authorised way. Namely data can be modified only by those who are entitled to do so and in authorised ways (e.g. an entity might increment some value but not decrement it). This property controls how data is modified and ensures that data is either modified correctly or not modified at all.
- **Availability**. Availability means making sure that an authorised person has to be able to access data in the authorised way (as specified by the confidentiality and integrity properties) in a specified time constraint. This property is fundamental because without this it would be easy to ensure the first two, in fact if we isolate a system we grant both confidentiality (if no one can access it, no unauthorised entity can access it) and integrity (if no one can access it, no one can modify some data in an authorised way). Since we want systems to be used, we also have to ensure that they are available (i.e. when a user wants to access a service, it has to be available). This property is very important because technology aims at building systems that are every time more available, thus we have to ensure the first two property even

for system whose availability evolves swiftly. Availability can also be defined as the speed at which information goes around between the components of a system.

CIA properties have to be balanced for security to work.

1.1.2 Vulnerabilities and exploits

A system might have vulnerabilities that can be exploited by an attacker to compromise a system. Let us clarify what vulnerability and exploit mean.

Vulnerability

A vulnerability is something that allows an attacker to violate the CIA paradigm. Notice that a vulnerability isn't necessarily an error or a design mistake, but it can be an unavoidable characteristic of the system that can be exploited to violate the CIA paradigm.

Exploit

An exploit is a way to use a vulnerability for violating the CIA paradigm. A vulnerability can be exploited in many ways, thus many exploits can leverage the same vulnerability.

Lock example

Let us consider a lock to understand the difference between vulnerability and exploit. A lock is made of metal, thus when the pins move in the pin-holes they generate friction and if the pins are pushed against the walls of the pin-holes with enough force, then the pins can even stop moving because the springs can't generate enough force. In this example friction is a vulnerability of the lock because it can be exploited to open the lock without the key. In particular one could use a tension wrench (i.e. an l-shaped piece of metal) to apply tension on the cylinder (i.e. rotate it) and then one could use the rake (i.e. another piece of metal) to move the pins in their opening position. This method exploits the fact that pins are kept in place by friction (thanks to the tension wrench) even when the rake isn't pushing the pin.

Properties

Here's a list of properties of vulnerabilities and exploits.

- Exploits and vulnerabilities can be disjoint. This means that one might know that a vulnerability exists but without knowing an exploit for that vulnerability, or vice versa. Usually we want to know a vulnerability to better understand its exploit. In our example one can learn the exploit (i.e. the process of picking a lock) without knowing the underlying reason why it works.
- In some cases, vulnerabilities can't be solved because are an intrinsic property (or characteristic) of the system. Luckily, we can always try to make exploits harder and reduce the risk of an attack. For instance, in the lock example, friction between metals is an intrinsic property of the lock, not a design mistake.

- Usually a system has many vulnerabilities, thus it can be attacked on different fronts. For this reason, it's useless to fix a small vulnerability (i.e. one hard to exploit) if there exist other that are much easier to exploit. Metaphorically, we say that a chain (i.e. a system) is as strong as its weaker chain (i.e. as the easiest vulnerability to exploit). Relating to the lock example, we can say that is useless to put a super secure lock on a thin wooded door because it's easier to break the door than picking the lock.

- Sometimes if the resources secured aren't worth much, it might not be worth fixing a vulnerability, or better said, replacing the system with a more secure one. Basically we have to estimate and relate the value of what we are securing and of the security mechanism.

1.1.3 Assets and threats

Understanding what we are protecting and what can attack a system is a fundamental task in computer security. Before introducing the concepts of assets and threats we should understand the difference between protection and security.

- **Protection**. Protection represents how an entity can defend itself from an attack, independently from the fact that an attack may or may not happen.

- **Security**. Security considers the fact that an entity is in a certain context that can generate some attacks.

To clarify this difference, consider a military truck in a war zone and a normal car in a city. The former offers more protection than the latter but the car is more secure because it is in a safe environment and (if we do not consider crashes), nothing bad is going to happen. This shows that to evaluate the security of a system we have to define what a threat is and how to describe the threats for a system.

Asset

Definition 1 (Asset)**.** *An asset is something the system is protecting.*

Assets have different forms in computer science, in fact an asset can be

- The hardware or software of a system.

- Data. Data is a fundamental asset and it's very important to defend it.

- Reputation. An attack might make public some information and ruin the reputation of a person or of a company. In this sense, protecting reputation is even more valuable than data itself.

- A concept. For instance when protecting a system that controls an airspace we have to consider that an attack may violate the security of the airspace.

Value of assets Every asset has an intrinsic value. When talking about values it's important to differentiate between three different values

- The value of the asset for the entity that has it.

- The value of the asset for the attacker.

- The value of the damage done if the asset is attacked.

These values can be very different, for instance an attacker might have no particular interest in compromising an asset but the damage generated by the attack could be extremely high (usually much higher than the value of the asset for the entity that has it).

Ethical aspects When talking about security of computer systems, we should also consider the ethical aspects. In particular, different attacks might have very bad consequences on the people involved in the attack (not only on those who has the compromised asset).

Threat

After defining what an asset is, we have all the ingredients to define a threat.

Definition 2 (Threat)**.** *A threat is a circumstance that potentially causes a CIA violation.*

In other words a threat defines what can happen to an asset. Some examples of threats are Denial Of Service attacks and identity theft. When talking about threats we have to differentiate between

- **Threat landscape**. The threat landscape of a system is the context in which the system operates.
- **Threat model**. The threat model defines the types of attacks a system can receive.

The threat model is fundamental when securing a system because it allows to define what menaces we have to defend against.

Building a threat model Since the threat model is so important, it's useful to see how it can be built. One can

1. Take the assets of the system.
2. Consider the CIA properties.
3. For each asset analyse what can happen if a property is compromised, i.e. what is the impact of a CIA violation.

When talking about threats it's important to define who or what is the source of that threat, in particular

Definition 3 (Threat agent)**.** *A threat agent is whatever can make a threat (i.e. an attack) occur.*

Threat agents can be

- **Intentional**. Intentional agents act intentionally to harm a system.
- **Unintentional**. Unintentional agents cause harm without intentionally wanting to.

Another important thing to point out is that we are always going to refer to **attackers** when talking about threat agents, in fact attackers are the most general type of threat agents. In particular,

- An **hacker** is a person that has an advanced knowledge of computer security and computer science and always learns.

- A **black hat** is a malicious hacker.

As we can see from this definitions not all attackers are hackers nor black hats, for instance a social engineer can gain access to a system guessing the password of a person using social hints and still have no experience in computer science. Considering attackers, we build a more complete threat model because we consider all possible entities that can cause a threat.

1.1.4 Risk

When talking about vulnerabilities, we stated that some are intrinsic to a system (e.g. friction between metal parts of a lock). This means that all systems have some vulnerabilities, thus all systems are vulnerable. For this reason the goal of security shouldn't be to build an invulnerable system because it's impossible.

When securing a system we have to consider risk,

Definition 4 (Risk)**.** *Risk is the statistical evaluation of how exposed to damage something is.*

Risk is a composition of different factors, in particular

- The **asset** to protect.
- The **vulnerabilities** of the system to which the asset belongs.
- The **threats** that menace the system.

In formulas we can say that $Risk$ is the composition (expressed with $\times$) of $Asset$, $Vulnerabilities$ and $Threats$.

$$Risk = Asset \times Vulnerabilities \times Threats$$

Basically, **vulnerabilities allow threats to get to an asset**. Risk is fundamental because security is based on risk. Also notice that if the value of an asset is well defined we can compute the risk as the product between its value and the probability p of something happening to it

$$Risk = value \cdot p$$

Some risk factors are controllable, others are not, in particular

- **Assets and vulnerabilities are controllable**. An asset is controllable because we can contain the damage an attacker can do, for instance doing backups. Vulnerabilities are also controllable because we can try to close or fix them using, for instance, authentication mechanisms or anti-viruses. As we have already seen, vulnerabilities can't be reduced to zero.
- **Threats are not controllable**. Threats come from attackers on which we have no control. Sometimes threats can be affected through choices, in particular the entity that holds the assets can provoke or reduce threats. For instance a company that fires a lot of people without reason can provoke a cyberattack while an eco-friendly company can reduce the possible threats. That being said, a security expert can't control the company directly, thus we have to assume that threats are not controllable.

Since we want to reduce risks, we should analyse which risk factors can be reduced to 0:

- Vulnerabilities can't be nulled, in fact some are intrinsic to a system.
- Threats could be null but usually there always is a threat, thus it's important to define threats and start analysing a system from its threat model. Threats are in fact the most important factor when securing a system.

Cost

Since we can only control assets (i.e., reduce the damage that can be done to an asset) and vulnerabilities (i.e., reduce them) we should find a balance between damage and vulnerabilities reduction and their cost. Cost is very important since every improvement has a cost that can be

- **Direct**. Direct costs are those that come directly from operating on the system. Some examples are buying an anti-virus, installing a software, evaluate the security of a system.
- **Indirect**. Indirect costs consider the impact on the users. In some sense, these costs are even more relevant than direct costs because they impact on the final user of the system. Some examples of indirect costs are
  - *Usability.* When we introduce a security measure, it can reduce the usability of the system. Consider for instance the fact that we have to type passwords to unlock the smartphone; a user needs more time to unlock the phone but the security level is (in general) higher. Sometimes users feel this indirect cost more than a monetary (direct) cost.
  - *Performance.* Introducing a security layer worsens the system's performance. This is inevitable because adding a software level always reduces performance and we can only control by how much.
  - *Privacy.* In some cases security systems need to collect personal information to work properly. This is a terrible trade-off because a user has to give away its personal information to protect them from others. Basically, we are protecting confidentiality by losing confidentiality.
  - *Productivity.* Security mechanisms can reduce productivity, for instance if a clerk has to insert a password before every critical operation, he/she is less productive.

Since cost is very important, we have to maximise security (i.e. adding protection) while minimising cost. Simply adding protection can make things worst because we increase some cost (e.g. indirect cost) and even generate new threats as a consequence.

Rules' inertia

Sometimes we enforce rules that have no actual effect in protecting against a threat. Such rules are not removed because of two reasons

- They are used as advertisement, to make people think that the system is secure.
- For those taking the decision, it's not convenient to remove the rule. In fact, say one wants to remove the rule, if something goes wrong everyone is going to say it's his/her fault for removing the rule and causing the damage.

This behaviour is called **inertia**, in particular we say that rules have inertia because, once enforced, they tend not to be removed.

1.1.5 Trust

The last fundamental concept we have to analyse before diving into more specific topics is trust.

When we are analysing a system (e.g., the threat model) we can keep analysing it at deeper levels. At every level we can assume it's secure (or we have secured it), but we have to check if the lower level is also secure. Consider for instance a computer program. We can decide that the source code is secure, but then we have to analyse the compiler because it might have vulnerabilities. After analysing the compiler we should also ask ourselves if the operating system is secure. This recursion can ideally go on forever, in fact we can check if the processor could introduce some vulnerabilities and so on. We also should take into account the fact that we might not know about some features of a system because they weren't documented by the company that sold it. Long story short, this search has no end, thus we should define a level where we stop because we think (notice that this is just an hypothesis) that no threat can occur below that level. The elements that we assume are secure (i.e. no threat menaces them) are called **trusted elements**.

Notice that trusted elements are element we trust, but that are not necessarily worth of trusting, in fact we assume that there is no threat, but they might be not secure. If an element is worth of trust we say it's **trustworthy**. If an element is trusted and trustworthy, everything works fine. Otherwise if an element is only trusted, something might go wrong (e.g., we might not have considered a threat that should make the element not trusted). If a trusted element fails, it should break.

Security nihilism

Let's go back to the infinite search we analysed before. One might think that there is no point in securing things at one level if the level below might introduce new vulnerabilities (and there always is a level below). This idea is called security nihilism and is rather wrong, in fact **the goal of security is to make things secure enough with respect to the threat model**. This means that we should try to mitigate the threats that can exploit the vulnerabilities of the system.

Part II

Cryptography

Chapter 2

Introduction

Cryptography is a fundamental building block for building secure software and communication. Let's start our journey into cryptography by defining it.

> **Definition 5** (Cryptography)**.** *Cryptography is a set of techniques to allow people to secure communication and data storage in presence of an attacker.*

This definition makes it very clear that the main goal of cryptography is to **hide content**. A very important concept to understand is that security is not cryptography and cryptography doesn't solve security. **Cryptography is a way to transform a problem in another problem which is easier to solve**.

Properties provided by cryptography Cryptography mainly provides four properties

- **Confidentiality**. Confidentiality ensures that only those allowed can read some information.
- **Integrity** and **freshness**. Integrity ensures that only those allowed can write and modify some information. In other words unauthorised entities should be able to tamper (i.e. interfere in order to cause damage or make unauthorized alterations).
- **Authenticity**. Authenticity ensures that data and their origin are guaranteed. In other words, it ensures that some information is sent from the real sender and that it hasn't been modified by another entity.
- **Non repudiation** (commitment). Non repudiation ensures that the data creator can't repudiate created data. For instance, a person that signed a document can't say that he/she didn't sign it. This property is the dual of authenticity.

Cryptography also has more advanced features that can be used, for instance, for proofs of knowledge (i.e. proving a secret without revealing the secret itself).

2.1 Kerckhoffs's principles for a cipher

Kerckhoffs defined 6 principles that every good cipher (i.e. object that uses cryptography to encrypt and decrypt data) should adhere to. Before introducing Kerckhoffs's principles, we have to define what a cipher is.

Definition 6 (Cipher). *A cipher is an object that uses cryptography to encrypt and decrypt data, i.e., to hide and un-hide content.*

Now we are ready to introduce Kerckhoffs's principles for a cipher.

Theorem 1 (Kerckhoffs's). *Every good cipher should adhere to the following principles*

- *Every good cipher must be practically, if not mathematically, unbreakable. Practically, it's very hard to guarantee that a cipher is hard to break, thus we can try to demonstrate that a cipher is hard to break until a certain level (e.g., given some resource, or a computing power).*
- *It must be possible to make a cipher public. This means that even if the algorithm for encrypting and decrypting data is made public, an attacker can't decrypt data without the key (i.e. the secret on which the cipher is based). This property is fundamental and is the one around which many modern cipher are built.*
- *The key must be communicable without written notes and changeable whenever the correspondents want.*
- *Every good cipher must be applicable to telegraphic communication.*
- *Every good cipher must be portable, and should be operable by a single person.*
- *Every good cipher should not require mental load for the user. This means that encrypting and decrypting shouldn't rely on the user, but on a machine instead.*

2.2 Randomness

Randomness is very important in cryptography, in fact we need it to generate random strings that can be used as secrets. Unfortunately, computers are deterministic machines thus it's hard to generate random numbers. This means that, when we talk about randomness, we want the process that generates the values to be random, not the output itself.

2.2.1 Definitions

Before describing the various ways a text can be encrypted, we should define the building blocks of a cipher.

Definition 7 (Plaintext). *A plain text* `ptx` *is something we want to encrypt.*

In the digital world a plaintext is a string of bits $\{0,1\}^l$ of length l. The set of all possible plaintexts `ptx` is P.

$$P = \left\{p : p \in \{0,1\}^l, l \in \mathbb{N}\right\}$$

Definition 8 (Ciphertext). *A ciphertext* `ctx` *is the set of output messages of a cryptosystem (i.e., a cipher).*

In the digital world a ciphertext is a string of bits $\{0,1\}^{l'}$ of length l'. The set of all possible cipher texts `ctx` is C.

$$C = \left\{c : c \in \{0,1\}^{l'}, l' \in \mathbb{N}\right\}$$

Notice that, the length of a plain text can be different from the one of the corresponding cipher text (i.e. $l \neq l'$).

Definition 9 (Key space)**.** *The key space K is the set of all possible keys that can be used in the cipher.*

In the digital world, a key is a string of bits $\{0, 1\}^\lambda$ of length λ. Now that all the main building blocks are ready we can start and define the meaning of encryption and decryption.

Definition 10 (Encryption function)**.** *An encryption function $\mathbb{E}$ is a function*

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

that takes as input a plain text $p \in P$ and a key $k \in K$ and returns the ciphertext corresponding to the plaintext p encrypted with key k.

Definition 11 (Decryption function)**.** *A decryption function $\mathbb{D}$ is a function*

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

that takes as input a cipher text $c \in C$ and a key $k' \in K$ and returns the plain text p that, encrypted with k returns c.

Note that, the keys k and k' for encryption and decryption can be two different keys or the same one.

If we consider a plain text $p \in P$ and a couple of keys k, k' for encryption and decryption, then it must be true that

$$\mathbb{D}(\mathbb{E}(p, k), k') = p$$

A visual representation of the encryption and decryption blocks are shown in Figure 2.1. To wrap things up, a cipher is defined as a tuple

$$< P, K, C, \mathbb{E}, \mathbb{D} >$$

where

- P is the set of possible plaintexts to cipher.
- K is the set of possible keys that can be used for encrypting and decrypting.
- C is the set of possible outputs of the encryption function.
- $\mathbb{E}$ is the encryption function.
- $\mathbb{D}$ is the decryption function.

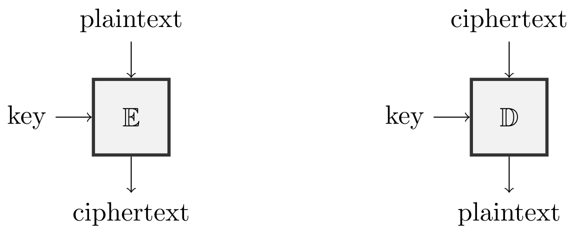


Figure 2.1: An encryption (left) and a decryption (right) block.

Chapter 3

Confidentiality

Let's start our analysis of cryptosystems by describing how they ensure confidentiality. Confidentiality ensures that only who has the right key k can read (i.e. decrypt) a cipher text c previously encrypted with k, or, in case of asymmetric encryption schemes (more on this later), with another key k'. **Confidentiality can be enforced using a cipher**.

3.1 Types of attacks

An attacker can execute different type of attacks on a cyptosystem, depending on the information he/she knows. In particular, the main attacks are

- **Ciphertext only attack**. An attacker only knows a cipher text.
- **Known plaintext attack**. An attacker knows a ciphertext and the associated plaintext.
- **Chosen plaintext attack**. An attacker can choose a plaintext and obtain the associated ciphertext. This type of attack might seem very unlikely but it's actually quite common. Consider for instance the response of a server to a badly formatted request; an attacker can send a request and, since the protocol is known, he/she knows the expected response.
- **Active attack**. An attacker can tamper (i.e. stimulate and modify) the data and observe the reaction of a decryption capable entity. In some extreme cases the attacker can also see the decrypted data.

The first three attacks are passive attacks because the attacker can only listen and observe the system. On the other hand, the last attack is active because the attacker actively interacts with the system.

3.1.1 Power of attacks

When trying to define which among of the attacks is more powerful, we should define what powerful means. In particular

- An active attack is more powerful in the sense that it requires less effort because an attacker can do more things (i.e. observation and interaction) than in passive attacks.
- A passive attack is more powerful in the sense that if an attacker can break a cryptosystem with a passive attack, it can do the same with an active attack.

3.2 Symmetric encryption

Symmetric ciphers allow entities to encrypt and decrypt a message using the same shared key (which has to be a secret shared only between sender and receiver). Symmetric encryption schemes are computationally cheap, hence they are well suited for session communication (i.e., to exchange multiple messages and documents in a short period of time).

3.2.1 Perfect cipher

Our first step in defining a cipher is trying to build a perfect cipher (i.e. a perfectly secure cipher). A perfect cipher ensures that when two entities exchange an encypted message, a third party can't understand any new information. More formally, the probability that an entity A sends a message M to another entity B doesn't change after sending a message M'. In formulas

$$P(p_{send} = p) = P(p_{send} = p | c_{send} = c)$$

Shannon's cipher

Shannon demonstrated that any symmetric cipher $< P, K, C, \mathbb{E}, \mathbb{D} >$ with $|P| = |K| = |C|$ is perfect if and only if

- Every key is used with probability $\frac{1}{|k|}$.
- A unique key maps a given plaintext into a given ciphertext.

$$\forall (p, c) \in P \times C, \exists ! k \in K : \mathbb{E}(p, k) = c$$

Basically, for couple of ciphertext c and plaintext p it exists only a key k that maps p in c.

Notice that saying that P, C and K have the same cardinality is different from saying that for each couple of elements (p, c) there exists only one key k that maps p in c because in the first case we might map every element of P in every element of C using the same key k.

A cipher with the properties defined by Shannon is perfect because if the number of used keys where smaller than the number of ciphertexts (and plaintexts) sent, an attacker could get some new information every time a message is sent.

Practical feasibility

To use this cipher, we should generate a key as long as all the messages that we are going to send. This is because keys have to transferred and generating a key for each message would require to send the keys on a public channel. Transferring a single big key might be done out-of-band but the same isn't doable for many keys.

Bruteforcing perfect ciphers

Perfectly secure ciphers are impossible to bruteforce. To demonstrate this let us consider a key of 5 bits, hence a cipher with 32 possible keys. If we try each of the 32 keys to decipher a ciphertext c, we obtain all sensible messages, thus we don't know which one was sent and we don't know any new information on the cipher.

3.2.2 Pseudo Random Permutations

As we have seen, perfect ciphers can't be used in real-life application, thus we should use another approach to create ciphers. In particular we have to use keys that are shorter than the message (this means that surely we aren't going to talk about perfect ciphers) and to ensure security we try to reduce the problem of hiding a message to an hard mathematical problem (e.g., factorising a large number). Since the hard mathematical problem is infeasible to compute practically, we have secured the message. The problem is that we have to demonstrate that the cipher perfectly maps in the mathematical problem, i.e., there aren't shortcuts (i.e., vulnerabilities) that can solve the problem in a reasonable time.

Building a hard to break cipher

After understanding that we have to map a cipher in an hard mathematical problem, we should outline a procedure for building an encryption algorithm. The procedure works as follows

1. Define the behaviour of an ideal attacker.
2. Assume a given computational problem is hard.
3. Prove that any non ideal attacker (i.e. that doesn't have complete knowledge about the cipher and plain text and the encryption algorithm) solves the problem.

If we can show that there is a way for the attacker to go around the hard problem, then we show that the cryptosystem has a vulnerability.

Generating the key

Another important problem to solve when building a cipher is how to generate the key. In particular we want to generate a key at random. To do so we need a Cryptographically Secure PsudoRandom Number Generator (CSPRNG). A CSPRNG is a deterministic function

$$r : \{0,1\}^{\lambda} \to \{0,1\}^{\lambda+l}$$

whose output can't be distinguished from a truly random sequence of numbers. This means that an attacker can't distinguish if a series of numbers has been generated by a CSPRNG or by a truly random event (i.e. a person that throws a dice multiple times).

We have no proof that such function exists, but we have no proof that such function doesn't exists either. This means that, in practice we can only say if a function CSPRNG is not secure.

Pseudo Random Permutations

If we had a perfect CSPRNG we could xor the key obtained from the number generator with the text to encrypt and obtain the ciphertext but CSPRNG are in practice very hard to to build (even though it's possible to do so), thus, to build them, we have to use a simpler building block called Pseudo Random Permutation (PRP). In practice these blocks are called **block ciphers**. Each of these blocks takes as input a plaintext (or ciphertext and a key, both of fixed length, and generates the ciphertext (or the plaintext) of fixed length. An example of PRP is shown in Figure 2.1.

Bruteforcing PRPs

A PRP is broken if an ideal PRP can be told apart from the broken PRP with less than 2^λ operations (where lambda is the length of the key). In other words an attacker has to use less operations than the number of keys. An attacker can distinguish an ideal PRP from a PRP when

- He/she can derive the input from the output.
- He/she can derive the key that identifies the PRP.
- He/she shows that k is not a key of the PRP.
- He/she can identify a non-uniformity in the inputs or outputs of the PRP.

Key length

As we have just seen, the security of a PRP is based on the length of the key, thus we should find a value for λ that makes bruteforcing unfeasible. In modern cipher we consider

- 64 bits keys **not secure** because it's feasible to bruteforce the key in human times (i.e. months or years).
- 80 bits keys **legacy**.
- 128 bits keys **10-15 years security**.
- 256 bits keys **long-term security**.

This evaluation applies only for ciphers that have no known vulnerability, or better said for which there is no shortcut.

Real world block ciphers

Some of the most known block ciphers (i.e., PRPs) are

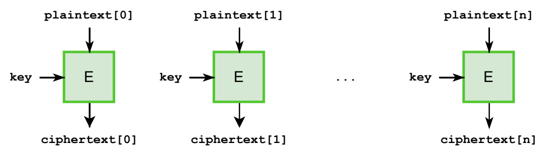
- **DES**. DES is now obsolete because it uses 56 bits keys but was widely used some decades ago. DES's security can be improved using triple-DES. This encryption algorithm uses two keys to alternatively encrypt a plaintext. In other words using triple DES triples the length of the key (from 56 bits to 168) making it more secure than its single-key version.
- **AES**. AES is the current standard for encryption. It uses a 128, 192 or 256 bit keys to encrypt a 128 bit block of plaintext. Being so used, the most common processors' architectures have instructions to accelerate encryption with AES.

3.2.3 Encrypting long text

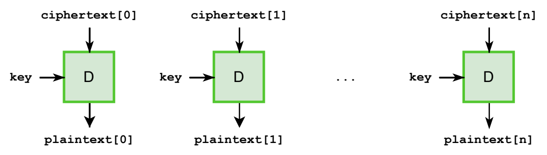
Usually the text we have to encrypt is much longer than the length of the key used in a block cipher, thus we have to find ways to encrypt the whole text with the same key. The basic idea is to use a single block cipher, i.e., a cipher that uses a key to encrypt a block of fixed length N and split the data to encrypt in many chunks of N bits that are encrypted using the same key and the same block cipher. In simple words, we take a group of identical block ciphers, each of which encrypts (or decrypts) a chunk of the data to encrypt (or decrypt).

Electronic code book (ECB)

An Electronic Code Book (ECB) allows to cipher long text splitting them in chunks of the same length of the key and using the same key to encrypt all the chunks. A visual representation of a ECB is shown in Figure 3.1.



(a) An Electronic Code Book cipher used for encryption.



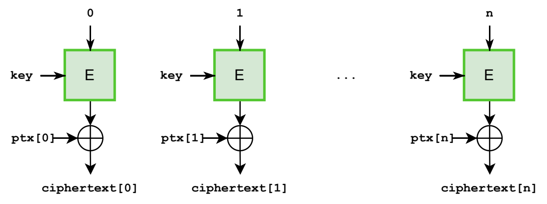
(b) An Electronic Code Book cipher used for decryption.

Figure 3.1: An Electronic Code Book cipher.

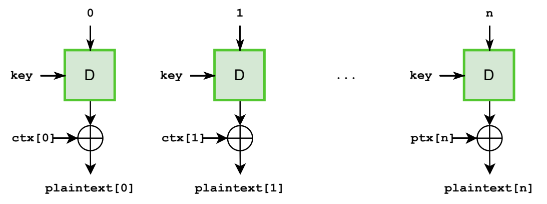
This method works but it has a big issue, in fact two identical chunks are mapped in the same ciphertext, thus an attacker could understand something about the plaintext. For instance, the world *the* is very common in English, thus one might think that if many blocks have the same ciphertext, then the corresponding plaintext could be the word *the*.

Counter mode (CTR)

To solve ECB's problem we can use a Counter mode block cipher. Every block cipher (i.e., every PRP) takes as input a counter (initialised to 0) and the key. The output of each block is then xored with the block to encrypt. The decryption phase works similarly, the only difference being that we xor the output of the PRP with the ciphertext to obtain the plaintext. A visual representation of a CTR cipher is shown in Figure 3.2.



(a) A counter mode cipher for encryption.



(b) A counter mode cipher for decryption.

Figure 3.2: A counter mode cipher.

The main issue with the CTR cipher is that it isn't robust against chosen plaintext attacks. This means that, if an attacker has a set of plaintext, he/she can understand what is the plaintext of an encrypted message because the encryption mechanism only relies on a counter (which can be guessed) and the plaintext. In other words, the attacker can encrypt the set of messages he knows (he/she only has to understand what is the starting counter, but that's not a big issue) and check if some messages he/she is spoofing correspond to the one he/she encrypted. The problem here is that, a given message and a counter, is always mapped to the same ciphertext.

Notice however that CTR solves the problem of repeated words in the message because each word is xored with a different value (the output of the PRP), hence its ciphertext is different.

Symmetric ratchet

To solve CTR weakness against chosen plaintext attacks we can add some randomisation. In particular, we can use nonces and change the key used to encrypt each block so that two identical messages are encrypted in a different way (i.e., $m_1 = m_2$ but $\mathbb{E}(m_1) \neq \mathbb{E}(m_2)$). The mechanism used to compute a different key for each message is called ratchet. A ratchet allows to generate only new keys making it impossible (or computationally very hard), given a key, to obtain a previous key.

Practically, a ratchet is made of a series of blocks, each of which is a pseudo-random number generator that takes as input a key of length λ and generates a new key of length 2λ (we say that the input key has been stretched). The first half of the output key is used as encryption key while the second half is fed to the next block of the ratchet to generate the next key. The ratchet is initialised with a random seed (i.e., a random number) of length λ. A visual representation of a ratchet is shown in Figure 3.3.

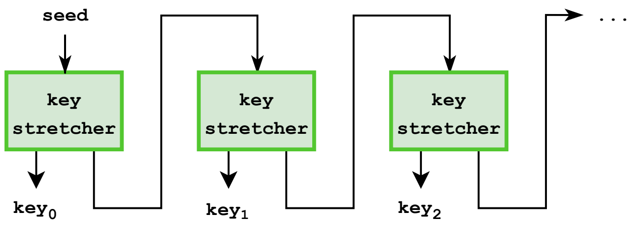
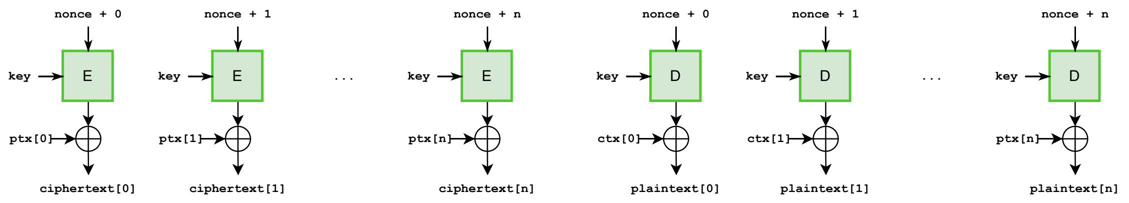


Figure 3.3: A ratchet.

Notice that, a ratchet isn't a cipher itself, but it's used to generate keys that can be used in a cipher.

CPA-secure CTR

Another way to mitigate chosen plaintext attacks (CPA) is to add a nonce to a normal CTR cipher. Basically, instead of using only a counter as input of each PRP, we use the sum of the counter and a nonce (i.e., a randomly chosen number). A visual representation of a CPA-secure CTR cipher is shown in Figure 3.4.



(a) A CPA-secure counter mode cipher for encryption.

(b) A CPA-secure counter mode cipher for decryption.

Figure 3.4: A CPA-secure counter mode cipher.

Adding a nonce allows us to obtain a different ciphertext when we encrypt the same message multiple times. This result is obtained thanks to the nonce (that changes every time a new message is encrypted). Note that the nonce is public.

3.3 Asymmetric encryption

Asymmetric encryption changes the perspective with respect to symmetric encryption. In particular, in this case instead of having a single shared key that has to be secretly shared and saved by two parties, we have a couple of keys, one for each party:

- A public key k^+ that is publicly shared with everyone.
- A private key k^- that is kept as a secret.

Private and public keys are build so that,

- **Something encrypted with Alice's private key k_A^- can be decrypted only with her public key k_A^+.** More properly, it's computationally hard to decrypt a message encrypted with k_A^- without k_A^+.
- **Something encrypted with Alice's public key k_A^+ can be decrypted only with her private key k_A^-.** More properly, it's computationally hard to decrypt a message encrypted with k_A^+ without k_A^-.
- **Computing Alice's private key k_A^- given her public key k_A^+ is computationally hard**.

This means that, when Bob wants to send a message to Alice, he can encrypt it with her public key (which is public and known by Bob) and he's sure that only Alice can decrypt and read that message. Notice that, since we use a publicly known key, asymmetric encryption is sometimes called **public-key encryption**.

3.3.1 Diffie-Hellman key agreement

The Diffie-Hellman key agreement protocol is a protocol that allows two parties to agree on a key using a public channel. Being able to use a public channel to build a shared key is fundamental for symmetric encryption, in fact two parties that want to communicate must share a secret without meeting in person and physically sharing it in secret (e.g., a client and a server). Note that, the Diffie-Hellman protocol is used to agree on a key not to exchange a it because the key is not shared on the public channel.

The Diffie-Hellman protocol is based on the discrete logarithm problem which is a computationally hard problem.

The protocol

Say Alice and Bob want to agree on a secret k_{AB}. The Diffie-Hellman protocol works as follows

1. Alice and Bob choose a large secure prime p (i.e., so that the discrete logarithm problem is hard to solve). p can be publicly disclosed and shared.
2. Alice and Bob choose a number g with some properties (it has to be a primitive radix of $\mathbb{Z}_p^*$). g can be publicly disclosed and shared.
3. Alice secretly chooses an integer a between 1 and $p-1$ (ideally much bigger than 1). a is kept secretly by Alice.
4. Bob secretly chooses an integer b between 1 and $p-1$ (ideally much bigger than 1). b is kept secretly by Bob.

5. Alice computes $g^a \mod p$ and sends it to Bob. Note that this message is exchanged publicly.
6. Bob computes $g^b \mod p$ and sends it to Alice. Note that this message is exchanged publicly.
7. Alice, upon receiving Bob's message, computes $\left(g^b\right)^a \equiv g^{ab} \mod p = k_{AB}$.
8. Bob, upon receiving Alice's message, computes $\left(g^a\right)^b \equiv g^{ab} \mod p = k_{AB}$.

Finally, Alice and Bob have the same shared secret k_{AB} exchanging only messages on a public channel. Let us now understand why this protocol works. First of all let us list all the information that Eve the eavesdropper (i.e., someone that listens all the messages exchanged on the public channel) knows:

- The numbers p and g.
- The message $g^a \mod p$ sent by Alice to Bob.
- The message $g^b \mod p$ sent by Bob to Alice.

If Eve wants to obtain g^{ab} she needs to compute a and b from $g^a \mod p$ and $g^b \mod p$. This problem is called discrete logarithm problem and it's computationally hard to solve. A visual representation of the Diffie-Hellman protocol is shown in Figure 3.5.

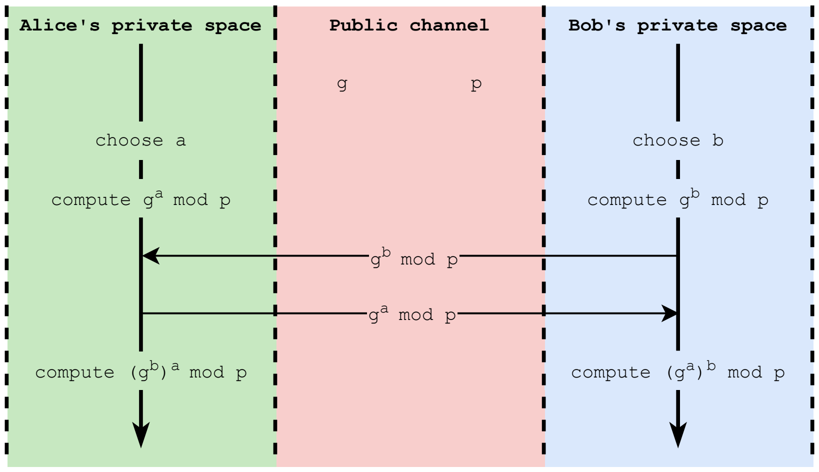


Figure 3.5: A graphical representation of the Diffie-Hellman key agreement protocol.

Man in the middle attacks

The Diffie-Hellman protocol allows to agree on a secret, however it's vulnerable to man in the middle (MITM) attacks. Say Chad is an attacker on the public channel. Chad can establish a communication with Alice, pretending to be Bob, and agree on a shared key with her. Chad can do the same with Bob, pretending to be Alice and agreeing on a shared key. Alice and Bob think they are talking one to each other, and they actually are, but their communication is filtered by Chad in the middle. This means that Chad can read every message exchanged by Alice and Bob because Alice is not talking with Bob directly, but to Bob through Chad. Moreover, Alice's shared key is shared with Chad. In other words

1. Alice wants to send a message m to Bob.
2. Alice sends the message to Chad (thinking he's Bob) using the shared key k_{AC}.
3. Chad can decrypt and read Alice's message because he has the shared key k_{AC}.
4. Chad uses the key k_{BC} shared with Bob to encrypt Alice's message.
5. Chad sends the encrypted message to Bob.
6. Bob can decrypt the message because he shared a key k_{BC} with Bob thinking that he's talking with Alice.

Long story short, this protocol is vulnerable to MITM attacks because it doesn't implement any authentication mechanism.

3.3.2 Public key encryption algorithms

The most used public key encryption algorithms are

- **Rivest-Shamir Adleman** (RSA).
- **ElGamal**.

Rivest–Shamir–Adleman

The RSA algorithm encrypts 2048 or 4096-bit long messages with 2048 or 4096-bit long keys.

ElGamal

The ElGamal algorithm uses kilobit-long keys to encrypt messages. The ciphertext is twice as long as the plaintext.

3.3.3 Secret sharing without agreement

Public key cryptography allows to share a secret (e.g., a symmetric key) without agreeing on it (like in the Diffie-Hellman protocol). Say Alice wants to send a secret k_{AB} to Bob:

1. Alice generates a couple of keys k_A^+ and k_A^- and publishes the public key k_A^+.
2. Bob generates the secret k_{AB} and encrypts it with Alice's public key k_A^+, so that only she can read it.
3. Alice receives $\mathbb{E}(k_{AB}, k_A^+)$, decrypts it using her private key k_A^- and obtains the secret k_{AB}.

The same procedure can be repeated with swapped roles to exchange a new secret k'_{AB}. The secrets k_{AB} and k'_{AB} can be combined to obtain a result similar to the one provided by the Diffie-Hellman protocol (but without agreement).

3.3.4 Efficiency

In theory, asymmetric encryption could be used for secure communication without symmetric encryption. However, in practice, asymmetric crypto-systems are much slower (10 to 1000 times slower) than symmetric ones, hence it's much more convenient to use public key cryptography to exchange the keys used for symmetric encryption.

Chapter 4

Integrity

Another important property that we would like to have is integrity. Integrity means ensuring that no unauthorised part can change the content of a message. Notice that, integrity is different from confidentiality, in fact if an attacker is able to change a ciphertext, we can't detect the change when decrypting the ciphertext (in normal conditions).

4.1 Message Authentication Codes

A Message Authentication Code (MAC) is a small piece of information, called tag, that allows to check if a message has been modified. Note that, MACs only provide integrity but not authentication. A system that uses MACs needs

- A function

  ```
  compute_tag(string, key)
  ```

  that, given a string a key returns the tag corresponding to the input string.

- A function

  ```
  verify_tag(string, tag, key)
  ```

  that, given a string `string`, a tag `tag` and a `key` verifies if the tag `tag` corresponds to the string `string`.

Note that the entity that creates the tag and the one that verify it must have the same key.

4.1.1 Idea attack model

Before analysing how MACs work, let us define what is the ideal condition for an attacker and what result we would like to obtain. Ideally, even if an attacker knows as many message-tag pairs as he/she wants, he/she can't forge a valid tag for a message for which he/she doesn't know the tag.

4.1.2 CBC-MAC

Let us now describe practically how does a MAC work. An example of MAC is CBC-MAC that uses a block cipher. In particular, the message from which we have to generate the tag is split in blocks. Each block is xored with the result of the previous block cipher and put as input of a block cipher. The output of the last block cipher is the tag. A block cipher for generating a CBC-MAC is shown in Figure 4.1.

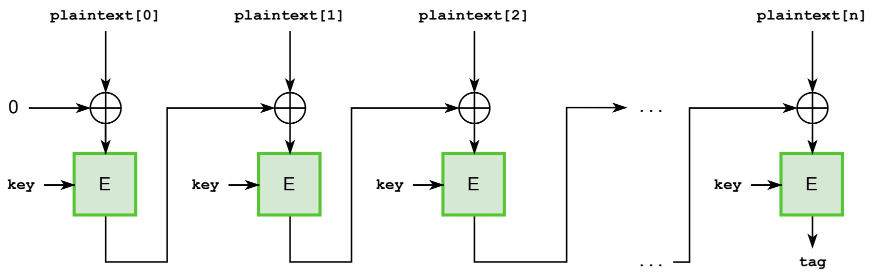


Figure 4.1: A block cipher MAC.

4.1.3 MAC for cookies

MACs can be used to check that a cookie hasn't been modified by the client. Cookies are generated by the server and sent to the client upon receiving the first request. The cookie is then sent to the server with every subsequent request. To verify that the client hasn't changed the cookie, the server can generate the tag together with the cookie and store the tag so that, when it receive a cookie, it can check if it hasn't been modified using the `verify_tag` function.

4.2 Hash functions

One way to enforce integrity is by means of hash functions.

Definition 12 (Hash function)**.** *An hash function h is a function*

$$h : \{0,1\}^* \to \{0,1\}^l$$

that takes in input a string of undefined length and returns a string of fixed length l and for which the following problems are computationally hard

1. *Given $d = h(s)$, find s. s is called the **first preimage** of h.*
2. *Given s and $d = h(s)$ find $s' \neq s$ for which $h(s') = d$. s' is called the **second preimage** of h.*
3. *Find r and s so that $h(r) = h(s)$. When two bit-strings have the same hash output, we say we have a **collision**.*

The output $d = h(s)$ of an hash function is called **digest**. It's important to underline that, since an hash function maps strings of arbitrary length to strings of a fixed length, the domain is much bigger than the codomain, hence two different strings can be mapped to the same digest.

Note that, the second property, called weak collision resistance, is harder to break than the third one, called strong collision resistance. Notice that, if a cipher satisfies the third property, it's hard to break it even in the easiest context (i.e., when the attacker can choose two arbitrary messages), hence the system is strongly resistant (because we can't break it in the most favorable conditions); that's why the third property is called strong collision resistance. Ideally, a concrete hash function should require

- $\mathcal{O}(2^d)$ hash computations to find the first preimage.
- $\mathcal{O}(2^d)$ hash computations to find the second preimage.
- $\mathcal{O}(2^{\frac{d}{2}})$ hash computations to find a collision.

4.2.1 Uses of an hash function

Hash functions can be used in various ways, some examples are:

- Store the hash of a value instead of the value itself (e.g., for passwords).
- HMACs.
- Use only the hash of a disk image obtained in official documents.

HMACs

HMACs are a version of MACs based on hash functions. The main idea is to send a message and its hash. The hash should be computed using also a shared secret between sender and receiver so that an attacker can't forge a new HMAC after modifying the message. Say for instance that Alice wants to send a message m to Bob and that they share a secret k. The interaction between the two works as follows

1. Alice sends to Bob m and $d = h(m, k)$.
2. Bob receives the message m' that could be m or a modified version of m and $d = h(m, k)$.
3. Bob computed $d' = h(m', k)$.
4. Bob checks if $d = d'$.

Note that an attacker can't snoop the conversation because

- If he/she modifies the message but not the hash, Bob can understand it because he finds out that $d \neq d'$.
- He/she can modify the message but not the hash because he/she doesn't have the shared secret k.

4.2.2 Concrete hash functions

Currently, the most used hash functions are

- SHA-2 with 256, 384 or 512 bit-long digests.
- SHA-3 with 256, 384 or 512 bit-long digests.

The functions above are currently unbroken (i.e., they are both strongly and weakly collision resistant). Before SHA-2 and 3 the standard were SHA-1 and MD-5, however these functions should be considered not secure because it's possible (and in some cases easy) to find collisions.

Chapter 5

Authentication

Authentication is another fundamental part in securing systems. In particular, it's needed because

- We would like to check the authenticity of some data. This problem can be tackled by **digital signatures**.
- When we are using someone's public key, we would like to check if that key actually belongs to that entity. Basically, if an entity pretending to be Alice tells us to use key k_A to talk to her, we would like to know if k_A actually belongs to Alice, hence Alice is actually who she says she is. This problem can be tackled with **digital certificates**.

5.1 Digital signatures

Digital signatures provide strong evidence that some data has been sent by a specific entity. The main characteristics of digital signatures are

- They don't need a pre-shared secret.
- They can't be repudiated by the creator (i.e., the creator can't say it wasn't him/her who created the data).
- They are asymmetric cryptography algorithms.

5.1.1 Principle of working

Digital signatures use public key encryption. This means that the entity that encrypts the message (Alice) has to generate a private and a public key. When Alice wants to send a message m to Bob, she

1. Encrypts m with her private key k^-. The encrypted message $m^- = \mathbb{E}(m, k^-)$ is the signature of m.
2. Sends m and m^- to Bob.

Bob, upon receiving Alice message (and the signature),

1. Computes $m^+ = \mathbb{D}(m^-, k^+) = \mathbb{D}(\mathbb{E}(m, k^-))$ using Alice public key k^+.

2. Checks if the message received m' is the same as m^+.

Digital signatures work because

- we have verified that $m = m^+$ decrypting m^- with Alice's public key.
- m^- has been encrypted with Alice's private key k^-.

Since the only way to meaningfully decrypt a message encrypted with Alice private key is to use her public key, then, using Alice public key, we must have decrypted a message encrypted with Alice private key, hence it must have been Alice who sent the message. In practice, when Alice wants to send a signed message m to Bob

1. Alice encrypts the message m using the private key k_A^-.

   $$S = (m, k_A^-(m))$$

   S is the signature of the message m generated by Alice. If Alice sent S to Bob, everyone could read the message so she has to encrypt S so that only Bob can read it.

2. Alice encrypts the couple m, k_A^- with Bob's public key k_B^+.

   $$S_e = k_B^+(m, k_A^-(m))$$

   S_e is the encrypted signature that can only be read by Bob because only Bob can obtain $(m, k_A^-(m))$ using his private key k_B^-.

3. Bob deciphers S_e using his private key k_B^- and obtains the signature of Alice.

   $$S = K_B^-(K_B^+(S_e)) = (m, K_A^-(m))$$

4. Bob deciphers $k_A^-(m)$ using Alice's public key k_A^+ and checks if the result is equal to m.

   $$K_A^+(K_A^-(m)) = m$$

This process works because

- Only Bob can read $m, K_A^-(m)$ because it's encrypted with Bob's public key.
- Bob can decipher $k_A^-(m)$ because everyone knows Alice's public key.
- If $k_A^+(k_A^-(m)) = m$ then only Alice can have encrypted m.

5.1.2 Document signing

Digital signatures can be applied also to entire documents and not only to small messages. In this case (for large documents) signing the whole document would be too computationally expensive, hence what we usually do is signing the hash (Definition 12) of the document. In this case, if Alice wants to send a document d to Bob

1. Alice generates the H hash of the document d.

2. Alice encrypts the digest H using her private key k_A^-.

   $$S = (d, K_A^-(H))$$

   S is the signature of the document d generated by Alice and contains the document to send and the encrypted digest. In this case we don't want to encrypt S because it would be to expensive to encrypt the document d.

3. Bob deciphers $k_A^-(H)$ using Alice's public key k_A^+ and computes the hash H' of d.

   $$H = K_A^+(K_A^-(H))$$

4. Bob checks if the hash H' computed locally equals to the value received

   $$K_A^+(K_A^-(H)) = H = H' = h(d)$$

A problem with document signing

An important thing to underline is that digital signatures are static, this means that the content of the whole document is signed. Unluckily, some document formats support macros, hence what's displayed can change dynamically, without changing the content itself. Consider for instance a document in which it's possible to display the current date. The content of the document is the same (e.g., something like `Selection.InsertDateTime DateTimeFormat:="MM dd, yy"`) however the displayed date is the different every time. This is a big problem because we can sign a document but another person could see a completely different document, yet correctly signed.

5.1.3 User authentication

Digital signatures can also be used for authenticating users. In particular, when a server wants to authenticate an user:

1. The server has the user's public verification key (deposited when the account has been created).
2. The server asks the user to sign (i.e., to encrypt with his/her private key) a challenge (e.g., a random string of bits).
3. The client signs the challenge using his/her private key.
4. The server verifies if the sign is correct. More precisely, the server decrypts the signature with the user's public key and if the resulting value is equal to the challenge sent to the user, then the user is authenticated, because only he/she could have encrypted the message with the private key equivalent to the public key deposited.

This method works really well and doesn't require any passwords, however it's very hard to use on mass-scale application. The main issue is that the user has to create a valid key couple and a valid certificate (more on this later), which could be not as easy as using a password.

5.1.4 Currently used signature schemes

Nowadays, the most used and widespread signature schemes are

- RSA-based signatures that use the RSA algorithm for asymmetric encryption. The main drawback of this scheme is that it's quite slow.
- DSA (Digital Signature Algorithm).

5.2 Digital certificates

Digital signatures are cool, however they still have an issue, which can be mitigated with digital certificates. Say we want to verify that a certain document has been sent by Alice. We can use her public key to decrypt the signed document and, it the result is the same as the actual document sent, we are sure that Alice signed it. However, we still are not sure to be talking to Alice. We know we are using the public key of someone that says to be Alice but we are not sure that she's actually Alice. Say for instance we want to impersonate Alice, we can

1. Create a couple of keys k_a^+, k_a^-.
2. Publish our public key k_a^+ saying that we are Alice.

When requires us to sign a document, we can correctly sign it and the other person can correctly verify our signature, but we aren't Alice. This is where digital certificates come into play.

Definition 13 (Digital certificate)**.** *A digital certificate is a document that provides the correspondence between a public key and an entity.*

Basically a certificate says that a certain public key k^+ belongs to a specific person. When using certificates we can't trick in believing we are Alice because her certificate says that her public key is k_A^+ and not k_a^+. This means that, when Alice wants to send a signed document to Bob:

1. Alice encrypts the message m using the private key k_A^-.
$$S = (m, k_A^-(m))$$
2. Alice encrypts the couple m, k_A^- with Bob's public key k_B^+.
$$S_e = k_B^+(m, k_A^-(m))$$
3. Bob deciphers S_e using his private key k_B^- and obtains the signature of Alice.
$$S = K_B^-(K_B^+(S_e)) = (m, K_A^-(m))$$
4. Bob checks if the public key k_A^+ he has belongs to Alice using her certificate.
5. Bob deciphers $k_A^-(m)$ using Alice's public key k_A^+ and checks if the result is equal to m.
$$K_A^+(K_A^-(m)) = m$$

Certificate Authorities

Digital certificates are only a way to move a problem elsewhere, in fact one might ask who ensures that a certificate belongs to a certain entity. This problem is partially solved by Certificate Authorities (CAs), i.e., companies that provide certificates. CAs also sign certificates so that, when we ask them a certificate we are sure that we are actually talking to a Certificate Authority. In simple words, a Certificate Authorities tells us that a certain certificate is legitimate and valid using digital signatures (if a certificate is correctly signed by a CA and we trust the CA, then it's valid).

Note that, with certificates we have moved the problem of trusting Alice to the problem of trusting who provides authentication for, i.e., who trusts Alice. Moreover, certificates are not the definitive solution, in fact, since a certificate is signed by a Certificate Authority, we would like to check that the Certificate Authority is legitimate, i.e., we would like to verify the CA's certificate. This means that we need another CA that provides the certificate for the first CA. This problem continues indefinitely and the only way to stop the recursion is to have a small set of CAs, trusted by everyone, that sign their own certificate (i.e., they provide trust for themselves). This set of CAs is usually preset on a computer when buying it.

Certificate Authorities are usually organised in a hierarchical structure where each Certificate Authority provides trust (i.e., the certificates) for its children. The small set of CAs that have no parent are at the top of the hierarchy.

Chapter 6

Secure communication scheme

6.1 Communication scheme for secure communication

In this part we have seen different technologies (symmetric encryption, public-key encryption, certificates) that partially solve some problems related to secure communication. Each technology has its drawbacks, however they can be combined to obtain a solid (yet not bulletproof) system for secure communication. In particular, when Alice wants to establish a secure communication channel with Bob:

1. Alice uses certificates and Certificate Authorities to check that she has Bob's legitimate public key k_B^+ (and Bob does the same with Alice's public key k_A^+).
2. Alice and Bob use public-key encryption to exchange a shared key k_{AB} that will be used to encrypt messages from now on.
3. Alice and Bob use the shared key k_{AB} to encrypt their messages.

Let us highlight some important aspects of this scheme

- Alice is reasonably sure of talking to Bob (and vice versa) because they used certificates and signatures to verify each other public key.
- The shared key is a secret between Alice and Bob because they exchanged it using public key cryptography.
- Alice and Bob use symmetric key encryption for normal communication because it's way cheaper than asymmetric key encryption. Moreover, using a key wears it down (i.e., allows attackers to guess it more easily) hence it's better to use the public key as little as possible because changing it would require to create a new certificate (hence it's costly).

Part III

Information theory

Chapter 7

Information theory

7.1 Communication theory

In computer security it's important to quantify the information of a message or, in general, of data. Consider for instance brute-forcing, we want to know how hard it's to try all possible words if a password has some characteristics.

Before analysing the main concepts of information theory, let us define a communication system. A communication system is made of

- A **source** that produces some information.
- An **encoder** that encodes the information produced by the source.
- A **channel** that allows the information to go from source to sender. The channel carries information in the form of a sequence of symbols that belong to an alphabet (e.g. 1s and 0s).
- A **decoder** that decodes the information received from the channel.
- A **destination** that receives the information sent by the source.

A graphical representation of a communication system is shown in Figure 7.1. We can notice that this representation is rather similar to the one used in cryptosystems.

An important concept to point out is that source and destination communicate only through the channel, thus the destination knows what the information sent is only when it receives it. This means that we can model the sender with a random variable $\mathcal{X}$ and we can say that when the destination receives the information, it observes a value (i.e. an outcome) of the random variable $\mathcal{X}$.

$$\begin{tikzcd}\boxed{\text{source}} \arrow[r] & \boxed{\text{encoder}} \arrow[r] & \boxed{\text{channel}} \arrow[r] & \boxed{\text{decoder}} \arrow[r] & \boxed{\text{destination}}\end{tikzcd}$$

Figure 7.1: A communication system.

7.2 Entropy

Now that we have understood how a communication system is modelled, we can start studying the quantity of information carried by a message. This quantity is called **entropy** $H(\mathcal{X})$. In other words entropy $H(\mathcal{X})$ measures the quantity of information of a source $\mathcal{X}$. Formally,

Definition 14 (Entropy)**.** *The entropy of a random variable $\mathcal{X}$ is the average level of information, surprise, or uncertainty inherent to the variable's possible outcomes*

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

where

- *n is the number of possible messages that can be sent by the source.*
- *p_i is the probability that the source sends message i.*
- *b is the base in which the word is encoded. If we consider the digital world, we use $b = 2$.*

Entropy is expressed in bits, in fact it represents the smallest number of bits needed to encode all the messages that can be sent. If we use less bits than H we lose some information, hence entropy defines a lower bound for compression algorithms.

Entropy in bruteforcing Entropy is very useful to understand if a bruteforce attack is feasible. More precisely, entropy represents the minimum number of bits needed for encoding messages, thus we have to bruteforce at least 2^H messages.

Example Let us consider an example to better understand how entropy works. Say we want to compute the entropy of a source $\mathcal{X}_1$that generates random 6 letters words (even without meaning). In this situation the source can generate 6^{26} words (if we consider a 26 letters alphabet) and each word is generated with the same uniform probability $p_i = \frac{1}{6^{26}}$. The entropy is therefore

$$H(\mathcal{X}_1) = \sum_{i=0}^{6^{26}-1} -\frac{1}{6^{26}} \cdot \log_2\left(\frac{1}{6^{26}}\right) \simeq 28.2 \text{ bits}$$

This means that a random 6 letters word should be encoded in at least 29 bits and a bruteforce attack should check at worst 2^{29} words.

Let us now consider a second source $\mathcal{X}_2$ that randomly generates 6 letter words from the English dictionary. In this case there are far less words (say 6300) and the probability distribution is the same (uniform), thus the entropy of $\mathcal{X}_2$ is

$$H(\mathcal{X}_2) = \sum_{i=0}^{6299} -\frac{1}{6300} \cdot \log_2\left(\frac{1}{6300}\right) \simeq 12.6 \text{ bits}$$

This means that the information brought by a 6 letter word from a dictionary is much less than the one brought by a general 6 letter word.

7.2.1 Min-entropy

In some cases, entropy is way too general because it computes an average among the possible realisations, thus if there is a really unlikely realisation with very large value and many likely realisations with small values, the entropy has a value in between the two extremes. To solve this we can introduce the min-entropy H_∞.

Definition 15 (Min entropy)**.** *The min entropy of a random variable $\mathcal{X}$ is the entropy of the most probable realisation.*

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

Basically the min-entropy considers only the most probable outcome of the random variable $\mathcal{X}$. In other words, guessing the most common outcome of $\mathcal{X}$ is at least as hard as guessing a $\mathcal{H}_\infty(\mathcal{X})$ bit long bitstring.

Part IV

Security mechanisms

Chapter 8

Authentication

Authentication is a security mechanism that allows to ensure that an entity is who it says to be. Notice that usually we use the words identification and authentication interchangeably but in computer security they express two different concepts, in particular

- **Identification** allows an entity to **declare** who it is. In other words identification is a **declaration**.
- **Authentication** allows to **verify** if the entity is actually who it says to be. In other words authentication is a **proof** that the declaration is correct.

To better understand this concept, let us consider an example. When we log in a computer account, we have to provide

- A username, which is the way through which we identify ourselves to the computer.
- A password, which is the way we prove the computer that we actually are that user.

Authentication sides Authentication can be

- **Mutual** if both entities authenticate themselves.
- **Unidirectional** if only one entity authenticates itself.

Entities Authentication can be

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

Remembering these categories and recognises what type of authentication we have to develop in a system is fundamental, in fact each type of interaction suits better with different techniques.

8.1 Authentication factors

An entity can authenticate itself using (one or more) factors. In particular an entity can prove its identity using

- **Something it knows** (e.g a password).
- **Something it has** (e.g. a key or a card).
- **Something it is** (e.g. the voice, the face or the fingerprint).

These factors can be combined to obtain multi-factor authentication. Notice we have to use only different factors (e.g. using two things we know is not multi-factor). Usually multi-factor authentication involves a password (for historical reasons passwords where used first) and some other factor. For instance two-factors authentication 2FA involves a password and a something we have (i.e. the second factor).

Authentication factors for different entities We, as humans, authenticate

- **Usually** with something we **are** (e.g. we recognise a people by their faces).
- **Rarely** with something we **have** (e.g. a badge for a policeman). Also notice that many times we authenticate a an office or a job, not a specific person. Another thing to point out is that a passport, for instance, isn't something we have but something we are because it represents our person.
- **Almost never** with something we **know**.

In human-computer authentication it's the complete opposite, in fact we authenticate to a computer

- **Usually** using something we **know** (e.g. a password).
- **Sometimes** using something we **have** (e.g keys for banks accounts).
- **Rarely** using something we **are**.

These difference in hierarchies is a key point in authentication. Humans are used to recognise people by who their are because it is intrinsic of human beings, it's something we do naturally and without training and we are pretty good at it. On the other hand we, as humans, aren't able to keep secrets, thus asking a person to authenticate with something he/she knows is very hard (and we know it will probably fail). The reason why we keep using passwords is because it's easy for programmers and designers to develop password based authentication systems. Basically we build systems that are

- Easy to develop.
- Hard to use for humans.

This means that it's common for authentication systems to ask people to authenticate with a password, but it's not normal for users to do so, thus we, can't ask users to authenticate correctly with something they know.

It's also worth noticing that this prevalence in password-based authentication comes from an historical heritage. Back in the 60s, when the first authentication system was developed, computers weren't powerful enough for face recognition (i.e. something we are), thus we had to use password-based systems. Now that computers are more powerful, biometric authentication is becoming more common but the password trend is still prevalent.

8.1.1 Something you know

The first authentication factor we are going to analyse is something we know. In particular we are going to list and rank vulnerabilities and threats (notice that the asset to protect is the password, i.e. the thing we know) and then we will try to improve security.

Advantages Before analysing the vulnerabilities of of a password-based system, we can list some advantages.

- Passwords have a **low cost**, in fact they are easy to implement and have to be implemented only on software. On the other hand, the cost (indirect) is high for users because of the fact that we are not used to keep a secret.
- It's **easy to deploy** passwords.
- Passwords have a **low technical barrier**, in fact they can be implemented on every device, from the more to the less powerful.

Notice that these advantages are seen from the developer side.

Vulnerabilities

Passwords are vulnerable to

- **Guessing**. Guessing uses some personal information about the user to try and guess a password. This vulnerability is based on the fact that users aren't done for remembering secrets (i.e. random lengthy words), thus they usually choose passwords based on personal information (e.g. the birthday, the name, an hobby). Attacks based on this vulnerability are very common these days because social networks are full of personal information that can be used to guess a user's password.
- **Dictionary attacks**, or in general **cracking**. Dictionary attacks use a list of words (e.g. `rockyou.txt`) and common passwords to guess a user's password. The dictionary can contain passwords obtained from data leaks. This is particularly interesting because usually a user reuses the same password on multiple accounts (always remember that humans aren't done for remembering secrets). Dictionary attacks doesn't only try all words in the dictionary, but also a variation of them (e.g. `password`, `p4ssw0rd`, `PASSWORD`). Usually cracking tools allow to specify rules for such variations.
- **Snooping**. Snooping leverages the fact that usually people write their passwords down in plain-sight (e.g. in post-its sticked to the monitor). Snooping also includes **social engineering** that aims at forcing or tricking users into giving away passwords (e.g. phishing).
- **Bruteforcing**.

The first three vulnerabilities are called informed guesses because use some information to reduce the search scope (opposite to bruteforcing that tries all combinations).

Threats

Now that we have identified passwords' vulnerabilities, we can focus on the threats, i.e. on ranking the vulnerabilities.

1. The most common vulnerability is **guessing** in fact social networks are full of personal information that can be used to guess a password. Furthermore, guessing a password usually involves a limited number of guesses, thus it can get unnoticed in online services.
2. The second most common vulnerability is **Snooping**.
3. The last vulnerability is **cracking** (and in extreme cases bruteforcing). This is particularly true for online services, in fact cracking requires, in general, a large number of guesses, thus a web application could prevent an attacker to try multiple passwords.

Countermeasures

Now that we have in mind from what and how often we have to defend, we can try and find some countermeasures.

- **Increasing the complexity** of the password. This means requiring the user to insert non-alphanumerical characters (e.g., %, _, punctuation and so on). This type of countermeasure mitigates cracking and can be enforced by the computer when the user chooses the password (e.g., a password is valid only if it contains a number of non-alphanumerical characters).
- **Increasing the length** of the password. This countermeasure mitigates cracking and can be enforced by the computer when the user creates the password.
- Ask the user to **frequently change passwords**. This countermeasure mitigates cracking and can be enforced by the system.
- **Do not write a password down**. This countermeasure mitigates snooping but it's almost impossible to enforce because we can't control what the user does.
- **Do not reuse passwords**. This countermeasure mitigates guessing (if we consider as guessing trying the user's passwords appeared in a data leak). Using unique passwords is very powerful because it acts against the most dangerous threat (i.e. the most frequent one) but it can't be enforced because a system can't check if the user already used the password on another system.
- **Do not use personal information**. This countermeasure mitigates guessing and is very powerful (for the same reasons of not reusing passwords) but it's not too easy to enforce (a system might recognise that a password contains a date or a name).

Educating users

As we have seen, the only countermeasures that can be enforced are the ones that mitigates the least dangerous threat. This means that passwords have a great indirect cost (different from the initial low cost) because many users will choose a weak password that can compromise the whole system.

The only way to solve this problem is to educate users on choosing strong passwords and using authentication systems correctly. Some ways to educate users are

- **Teaching**.

- Using **password meters** that guide the user into choosing the right password (basically we are teaching the user how to create passwords while creating them). Depending on how we guide users we can obtain better results, in fact we always have to take into account that the system is interacting with a user, thus the user-interface is very important.

A final remark When an authentication system is broken, the problem usually isn't in the digital and technical part but in the user part. This doesn't mean the the fault is on the user but that the system has been designed poorly to interact with the users and it doesn't consider the nature of the users. **Always remember that users aren't done for keeping secrets, thus designing a system on the assumption that a user will choose a different, randomly-chosen, lengthy and unique password is a bad design choice that will lead to many security issues**.

Secure password exchange

An important part in authenticating with a secret (e.g. with a password) is how such secret is shared between two entities. In particular, to exchange passwords more safely, we can use mutual authentication and challenge-response protocols.

Secure password storage

Another important aspect of managing passwords is how to store them. In particular passwords are stored by the website that requires authentication and should never be stored in clear (i.e. plain text). One might think to encrypt them but it would be wrong, in fact we would need a key, that should be saved in memory. However, if an attacker can get the key, it can access all the passwords in the database. Basically, finding the problem of finding the passwords is reduced to the problem of finding the key.

Hashed passwords The solution to the storage-in-clear problem is to hash passwords. Hashed passwords are not invertible (i.e. from the hash we can't get the password), thus when an attacker steals the passwords of a website, he/she only sees hashes and can't get the real passwords. Notice that hashes alone can't be used to authenticate because the authentication phase works as follows

1. The client sends the password in plain text.
2. The server hashes the password and compares it with the hashed password stored.

However, even this method has some flaws, in particular humans always reuse passwords, thus if an attacker can leak a password database, many hashes will be replicated, because many people use the same password. This happens because an hash function maps the same input to the same digest, thus all people with password `password` will have the same hash. In particular, there exists tables, called rainbow tables, that store the most common password-hash couples. Notice that thanks to rainbow tables, an hashed password can be cracked with a simple lookup instead of a pure bruteforce attack.

Salted passwords The current approach to password storage, that deals with the reuse-of-passwords problem, is salted passwords. In particular, for each client that registers, the website

- Generates a random number called **salt**.
- Concatenates the **password** and the **salt**.

- Stores the salt in plain text.
- Stores the hash of the password concatenated to the salt.

Basically a password database entry looks like in Table 8.1. Notice that, after a data leak, an

username	**hashed password**	**salt**
`name.surname@mail.com`	hash(`pwd.salt`)	`salt`

Table 8.1: An entry in a password database (the dot notation indicates concatenation).

attacker knows the salt used for each user but this doesn't compromise the security of the database because the only way to get the password is to try every possible combination. Also notice that rainbow tables don't work anymore because even if a person uses a well known password, thanks to the salt it's impossible to distinguish it. Practically, the hash of `password1` and `password2` (where 1 and 2 are the salt) is completely different and can't be compared to the hash of `password`, which is the real password of the two users.

The social factor

As part of the to know factor we should also mention the social factor. For a small amount of time, Facebook tried to authenticate (actually to check if a user was legitimate after authenticating with the password) some users asking them to say what person, among their friends, was in a photo. However, this method failed because it was too easy for an attacker to find images online (on Facebook) and pretend to recognise the people in the photos.

Password recovery systems

Another important thing about passwords is how they can be recovered by users that forget them. In particular, when using password reset schemes,

- It's good practice to send a unique (i.e., that can be used only once) reset link (by e-mail or message) that allows the user to specify a new password. Notice however, that the e-mail or the phone number are the single point of failure of the schema because if the user doesn't control them, it's impossible to get the password back.
- It's bad practice to send a new password that can be used for accessing the application. Instead, one should send a temporary password that can be used only once to access the application and create a new password. This technique is similar to the one described at the previous point (the mail is replaced by the temporary password).
- It's bad practice to rely only on security questions because the answers to those questions can usually be found on social networks.

Brute-force protection

Protecting against brute-force attacks can be easy on a single device, however the same isn't true for Web applications. In particular it doesn't exist a one-fits all solution,

- **Reverse brute-forcing**. One could put a limit n to the number of attempts a user can do (with a specific username) and block it after that many attempts. However one could do $n-1$ attempts with the $n-1$ most common passwords on all users and, since people statistically use the same passwords (e.g., `password`, `secret`), get access to the Web application with at least one user.
- **Make accounts not-enumerable**.
- **IP blocking**. One could block the IP address of the device from where the request come from. However, some IP addresses are shared, hence we would end up blocking an entire network (which is some sort of Denial Of Service attack). For instance, if an attacker is using the network of an university, then the whole university network would be blocked.

Single Sign On

Users have a hard time remembering a lot of well-formed passwords and websites have to carefully store them. For both the parties it's costly to safely use passwords, thus some websites relies on Single Sign On (SSO) techniques. In particular, a trusted company, the identity provider, provides authentication for the website. This means that the user only has to authenticate on one website (the one of the identity provider) and use such profile to authenticate on other websites. An example of SSO is OpenAuth 2 (OAuth2) used, for instance, by Facebook and Google (that are the identity providers). A typical example of authentication works as follows

1. A user wants to authenticate on a website to use a service.
2. The website redirects the request to the identity provider.
3. The identity provider requires the user to authenticate.
4. The user authenticates using one or two factors.
5. The identity provider confirms to the website that the user is who he/she claims to be.
6. The website allows the user to use the requested service.

Notice that this technology is advantageous for every party, in fact

- The user has to remember only one password.
- The website has to deal with a limited number of credentials or no credentials at all.
- The identity provider can get data about the user (e.g., which websites he/she visits).

SSO isn't however free of issues. In particular

- It's a single point of failure, thus if the identity provider is broken, all the websites that use it are compromised. For instance, the password reset system has to be extremely solid (usually it's based on the mail, that is considered the trusted element).
- It's hard to implement SSO and the libraries that support the development might have bugs and vulnerabilities.

8.1.2 Something you have

The to have factor requires the user to demonstrate to have something, usually an USB shaped token or a smart card. This factor is much easier to use than passwords for humans because, since a very young age, we have been trained to not give away keys. Furthermore, it's much easier for humans to hold a physical object than a secret.

Advantages and disadvantages

Advantages The advantages of the to have factor are

- It's **easier for humans to use** than passwords.
- It has a **low cost**, in fact building smart cards or tokens isn't very expensive (at least per se).
- It offers a **good level of security**.

Disadvantages It's not all good though, the to have factor has some disadvantages too

- It's **hard to deploy**, therefore it's used only by some type of businesses. For instance, banks use token because it's easy to handle them to clients when they create the account. On the other hand, it would be hard (and probably not profitable) for a social network or a website to deliver a token to every user.
- Can be easily **stolen or lost**.

Two factor authentication

Usually, we refer to the to have factor as the second factor because it's widely used as the second factor in two factor authentication (2FA). In particular, the two most common devices used in two factor authentication are

- **Tokens**.
- **Smart cards**.

Tokens Tokens are USB shaped devices that contain a low-spec computer, a secret key and a counter that gets periodically updated (usually every 30 or 60 seconds). Tokens generate a number that has to be inserted by the user on a website. The website queries a server that verifies if the number is correct and, if the number is correct the server knows that the user has the key. Notice that the token never communicates with the server.

The server that verifies if the user has the token, has

- The key corresponding to the one in the token.
- The same counter of the token.

In particular tokens and servers usually use a public key scheme in which the token has the private key and the server has the corresponding public key.

When the user wants to generate a code,

1. The token encrypts its counter with the private key. The result is used by the user.

2. The server can decrypt the code inserted by the user because it has the public key of the token and check if the code corresponds to its local counter.

Notice that this procedure has two problems

- Token and server never communicate, thus the counters have to be synchronised at the beginning. This can be done with a special procedure; usually the system requires the user to insert some codes (not used for authentication) to synchronise the counters.
- The counter in the token will drift because it's usually less precise, thus the server has periodically to re-synchronise with the token. In particular the server can verify multiple values of the counter (e.g. $i-2$, $i-1$, i, $i+1$, $i+2$) and adjust the counter depending on which counter matches the token's counter.

One final comment on the security of tokens. Token are secure if the key is securely stored, because the only way to get the key is to get and broke the token. However, if someone gets the token, there's no need to extract the key because the token is enough to authenticate. Also notice that this means that a token can't be cloned because either

- an attacker has the actual token and uses it or,
- an attacker breaks the token and creates a copy, but the old token can't be used anymore.

Moreover, in recent years, the VSD2 law required banks to add elements of the transaction in the token's code, thus tokens are disappearing. This has been done because, even if it's impossible to mount online fishing attacks, it's still possible to do offline (i.e. real-life) fishing attacks.

Smart cards Smart cards are plastic cards with a small CPU on board (e.g. credit cards). The CPU

- Has a non-volatile RAM with a private key.
- Has no power source, thus the card has to be close to a reader that powers the card up.
- Can compute cryptography operations.

Authentication happens through a challenge-response scheme, in particular the card uses the private key to encrypt a challenge. Notice that, for the challenge-response protocol to work, the card has to expose its public key, however, it should never expose the corresponding private key. This is because the card is secure until the private key is safely stored in the card and can't be obtained without destroying the card. The idea is the same we described for smart token and prevents smart cards to be cloned.

TOPT In recent years, physical tokens have been replaced with apps that have the same functionality. Basically such applications are software-based password generators. However, such systems are application-based, meaning that the system can be attacked exploiting vulnerabilities of the application and of the hardware on which it's run.

Also notice that, in this case, the token is the phone (or better said, the SIM in the phone), however SIM attacks allow attackers to obtain a legal copy of the SIM and generate codes before the user realises that the attack took place.

8.1.3 Something you are

The to be factor, also called biometric factor, requires the user to demonstrate it has some physical characteristics. Biometric-recognition software extracts some characteristics of the user and checks if such user is allowed to access the system. The most common characteristic used are

- **Fingerprints**. Human fingerprints are almost unique (except for identical twins). Fingerprints are measured against a grid of key points, basically the software intercept the scan of the fingerprint with the grid of key points to get the key features of the fingerprint. Such features can be compared with the ones in the database to check if there is match. The scanner used for fingerprint recognition can be a simple camera enhanced with some capabilities to recognise if the finger is made of flesh (to avoid fingerprint cloning).
- **Face geometry**. Every human has an unique face shape and for us it's very easy to recognise a person by its face. However the same isn't true for computers, in fact it's hard to model a face. The scanner used for face recognition is a camera with special capabilities that allow to get 3d features.
- **Hand geometry** (palm print). The shape of an human hand and the set of blood vessels in it are unique for each person. In the former case we can use a simple camera to recognise the shape while in the latter we can use infrared cameras to get the picture of the vessels.
- **Retina scan**. The blood vessels on the back of a person's eyes are unique, thus can be used for authentication. Notice that retina scan can tell apart even identical twins. The scanner is, in this case, an infrared-light camera.
- **Iris scan**. Iris scan is different from retina scan, even if both use some features of the human's eye. In this case we use the colour of the eye's iris, in fact everyone has some unique imperfections that can be used to tell people apart. In this case the scanner is an high resolution camera that can recognise the imperfections in the pigmentation of the iris.
- **Voice**. The voice of a person is a unique characteristic, however, the range of variation in the voice can be very big, thus voice recognition isn't a very precise biometric authentication method.
- **DNA**. Each person has a unique DNA, however the methods used to analyse DNA are too slow for common-purpose use.
- **Typing dynamics**. Typing dynamics are very different to each person, however they can change, thus this feature can't be used for authentication but only for profiling.

Advantages and disadvantages

Advantages The main advantages of biometric authentication are

- The **user needs just to be himself/herself**, in fact biometric authentication works exactly how humans authenticate.
- It can provide an **high level of security**.
- It **doesn't require the user to carry around some hardware or remember a secret**.

Disadvantages The main disadvantages of biometric authentication are

- It's **hard to deploy** in fact the authentication devices need to be fitted with a very specific hardware (e.g., special cameras).
- It's **based on probabilistic matching**, in fact the characteristics collected by the scanner are matched to a machine learning model that returns, as for any ML model, the probability that the characteristic belong to who is authenticating. On the other side, password and token based authentication is deterministic in fact a password is either correct or not.
- Some biometric scanners use **invasive methods**.
- Some **characteristics can be cloned**, thus the hardest a feature is to clone, the better the authentication method is. To solve this issue, scanners can check the aliveness of the subject that is authenticating (e.g. scanning a face in 3 dimensions or checking if the finger is in real flesh).
- Some **characteristics can change in time**. This issue can be solved re-measuring the features every once in a while. This solution works good for frequent authentication, however when the authentications are rare, this solution is not effective nor secure.
- **Privacy**. A privacy problem arises if biometric features are used to authenticate users in an organisation. In particular one might not want to allow some third party to store its physical characteristics.
- It might not work for **people with disabilities**. Some people might not have some characteristics used for authentication (e.g. one might have no arms), thus we should find a authentication method that works for everyone.

These disadvantages highlight why biometric authentication is basically used only on personal devices. Let us consider for instance the last two disadvantages

- Privacy is not an issue if all the information about the user's characteristics are stored on the user's device.
- People with disabilities can choose the device that better fits with their needs.

Probabilistic matching

Let us analyse more in depth the problem of probabilistic matching. Since the decision about the matching is probabilistic we can have

- **False positives**. A false acceptance happens when the system accepts a person who is not the right individual.
- **False negatives**. A false rejection happens when the system doesn't accept a person who is the right individual.

The number of false positives and false negatives are related, in particular if we want to decrement one value we will increment the other. Say we want to reduce the false negative, this means that we increase the threshold used to decide if a person is who he/she claims to be. This means that the model is less precise, thus we will accept some people even if they aren't who they say.

False positives and negatives also lead to another problem related to the fact that biometric authentication is usually used on personal devices. In particular, having an high false acceptance rate is the actual vulnerability for the user because he/she doesn't want another person to access his/her phone. However, a user usually experience false rejections because he/she authenticate on the device. This problem is called base rate fallacy.

Moreover, the base rate fallacy creates issues when designing the system in fact we should obtain a low false acceptance rate, while ensuring a low false rejection rate to keep users happy.

This problem highlights why biometric authentication is used in personal devices such as mobile phones.

Chapter 9

Access control

9.1 Introduction

After authenticating a user, we have to check if that user can (i.e. has the privileges, or rights to) access the resources he/she requests . Even if resources can be accessed in multiple ways (e.g. written, read or executed) we can simplify the access control problem to a binary decision, namely if a user can access or not a resource.

A scale problem Computers have a lot of resources, hence memorising the access right for each couple `(user, resource)` can be very expensive. To solve this problem we can use a set of rules that express the access privileges in a condensed way. The set of rules is called **Access Control Model** (ACM).

Vulnerabilities Vulnerabilities in access control can be in the design of

- The **rules**.
- The **access control model**.
- The **system itself**.

9.1.1 Reference monitor

The part of the operating system that handles access control has to be very reliable, well tested and easy to get right, since it's a very important part of a system. For this reason this functionality is implemented by a small component of the OS called **reference monitor**. It's reduced size allows to it to be easily verified and tested and hard to break. After authentication, every access request is passed to the reference monitor that decides if the issuing user can access the resource or not.

9.2 Access Control Model

An Access Control Model describes the rules that are used to enforce access privileges in a system. ACMs can be divided in

- **Discretionary Access Control** (DAC). In DAC, the owner of a resource decides who has rights over the resource.
- **Mandatory Access Control** (MAC). In MAC a security administrator decides the privileges of users and the security levels of resources.
- **Role-Based Access Control** (RBAC). RBAC combines DAC and MAC, in particular it's DAC with some elements of MAC. Basically we give privileges to roles and then we assign roles to users. This means that if we assign a role to a user, we assign him/her all the privileges of that role. At the same time if we change the privileges of a role we change the privileges of all users with that role.

9.2.1 Discretionary Access Control

In Discretionary Access Control, the owner of a resource decides who can access it. Namely it gives privileges to the other users and it can also decide to give the ownership to someone else.

General model

All modern OSs use this Access Control Model, but with different characteristics. In general, in the DAC model,

- **Subjects** are those who can exercise privileges over a resource.
- **Objects** are the resources.
- **Actions** are what the subjects can do on the objects.

A triplet (S, O, A) of subject, object and action (i.e. a subject S can do an action A on object O), is called **protection state** and can be represented as a matrix. This general model is called HRU (Harrison, Ruzzo and Ullmann) model. An example is shown in Table 9.1.

	object 1	object 2	object 3	object 4
user 1	read	read, write		execute
user 2		read	read	
user 3	read, write, execute	write		execute

Table 9.1: An HRU matrix for 4 resources and 3 users

UNIX In UNIX operative systems

- **Subjects** are **users** and **groups** (i.e. sets of users).
- **Objects** are **files** because everything is represented as a file.
- **Actions** are **read**, **write** and **execute**.

Windows In Windows

- **Subjects** are **users** that have **roles**. Privileges are assigned to roles, not users.
- **Objects** are **files** and other resources.
- **Actions** are **delete**, **change permission** and **change ownership**.

Transitions

When we want to modify the HRU model we have to

1. Create or delete a row r for a subject.
2. Create or delete a column c for an object.
3. Add or remove the permission in cell (r, c).

These actions can be used when a user wants to create, delete or assign privileges. Say for instance a user u wants to create a new file f, then he/she should

1. Create object f.
2. Add the own privilege to f (i.e. in cell (u, f)) because if u creates the file, he/she should be the owner.
3. Add the read privilege to f (i.e. in cell (u, f)).

Notice that we don't create the user row because the user already exists. This procedure isn't however correct, in fact if f already exists and the system can bypass instruction 1 (when the file already exists), then things might go wrong. In particular

- If instructions 2 and 3 are executed anyways after instruction 1 is bypassed, then we are assigning owning privileges to someone who is not the owner (if the file already exists, it already has an owner).
- If instructions 2 and 3 are not executed, then the transition is not atomic.

This means that we have to add an `if` construct to the set of commands for modelling transitions to check if object f already existed.

Safety problem A set of operations is secure if, from an initial configuration, given a sequence of transitions, subject s (not owner) can obtain a right r on object f. If s can obtain r on f even not being the owner of f, then the set of operations is **unsafe by design**. In a generic HRU model, deciding if a set of operations is unsafe is an undecidable problem.

Implementations

Implementing DAC with a matrix is impossible because the system contains a huge number of files. Furthermore, a lot of cells in the matrix are empty (i.e. the matrix is sparse), thus a lot of space is wasted. To solve this issue we use

- **Capability lists**. Privileges are organised by row, namely, for each user we say which files can be accessed.
- **Access control lists**. Privileges are organised by column, namely, for each object we say which user can access it. Access control lists don't support multiple owners, however this problem can be partially solved with groups.
- **Authorisation tables**. Privileges are organised by cell, namely, we list only the non empty triples.

Access control lists are the most common implementation in modern OSs. The reason why is that when we create a file

- With access control lists, we need only to access the new object and add the users that have privileges on it.
- With capability lists we have to access all users that have some right on the added object.

This means that access control lists are faster than capability lists when creating a file. On the other hand (following the same reasoning), capability lists are faster when we create a user. However on a computer, files are created more frequently than users, then access control lists are faster.

Problems

The DAC model is widely because it's very easy to implement, however it also has some drawbacks. In particular

- It **doesn't scale well** with an increasing number of user. In fact, each user is the owner of some resource of the system and thus it can change the privileges and compromise the whole system (e.g. an user might allow all users to execute a file).
- It's **susceptible to malicious user problem**. Namely, a user might share the content of the file instead of the privileges of the file to allow another user to access the file.
- It's **susceptible to the Trojan horse problem**. Namely, a malicious program, if run by the owner can do things with the privileges of the owner.

9.2.2 Mandatory Access Control

In Mandatory Access Control, privileges are set by one or more administrators who define a classification of users (i.e., a clearance) and objects (i.e., a sensitivity).

Classification

A classification is made of

- A set of totally ordered **security levels**.
- A set of **labels**.

An example of security levels is *top secret*, *secret*, *unclassified* each level is strictly more secure than the following, thus we can define a relation of total order between the levels (i.e. *top secret* > *secret* > *unclassified*). Labels represent a topic, hence some examples of labels are *crypto*, *energy* and *finance*. A couple (security level, {set of labels}) can be assigned to a subject (and we call it clearance) or to an object (and we call it sensitivity). For instance an object with sensitivity $(secret, \{crypto, energy\})$ has a security level of *secret* and involves the topics (i.e., the labels) energy and cryptography.

Now that we have all the building blocks we can say that a classification is a partial order relation (i.e. a lattice) between couples (C, L) where C is a security level and L is a set of labels. In particular the dominance ($\geq$) between the couples is defined as

$$(C_1, L_1) \geq (C_2, L_2) \iff C_1 \geq C_2 \land L_2 \subseteq L_1$$

For instance the clearance, or sensitivity, (*secret*, {*crypto, energy*})

- Dominates (*unclassified*, {*energy*}) because unclassified is dominated by secret and {*energy*} is contained in {*energy*, *crypto*}.
- Doesn't dominate (*top secret*, {*energy*}) because *secret* doesn't dominate *top secret*.
- Doesn't dominate (*unclassified*, {*energy, crypto, finance*}) because {*energy, crypto, finance*} isn't contained in {*energy, crypto*}.

Bell-LaPaudla model

An implementation of the MAC model is the Bell-LaPadula (BLP) model. This implementation defines two rules

- The **simple security property** imposes that a subject with a certain clearance can only read an object that it dominates. In other words, a subject can only read down, i.e. read objects with a lower (or equal) secrecy level and can't read objects with an higher secrecy level.
- The **star property** imposes that s subject with a certain clearance can't write an object that it dominates. In other words, a subject can only write up, i.e. it can only write objects with an higher (or equal) secrecy level and can't write objects with a lower secrecy level.

In a nutshell, the BLP model forces an upward monotonic information flow because subjects can only write files with upper secrecy levels. Knowledge is basically moved from the bottom to the top. Together with the rules above, the BLP model also defines a DAC rule that imposes that an access matrix should be used to specify the discretionary access control.

Properties and problems The BLP model offers a good level of security, in fact it doesn't allow privileges to change (tranquility property) and avoids the Trojan horse and malicious user problems by design. However this model is very hard to use and implement and it's used only for critical applications. Even in this case we can spot some problems, in particular

- The more we add levels and labels, the fewer and higher in the lattice are the people that can read all objects.
- Integrity of the objects is not addressed.
- We need someone that can move information down the lattice. This operation is usually done sanitising the object, in fact usually in a file, only some part make it belong to a level. For instance only some paragraphs might contain top-secret information, while the others could be publicly available. Such parts can be removed and the object can be assigned a new sensitivity with a lower secrecy level.

Biba Biba is a model, similar to BLP, that takes into consideration the integrity of objects. In particular, Biba implements the same rules of BLP but in the opposite direction (no write up, no read down), in fact objects are moved downwards instead of upwards.

Part V

Software security

Chapter 10

Introduction

10.1 Vulnerabilities in software

One of the key components we have to define in a threat model is the set of vulnerabilities of the system. In software, a vulnerability is an unmet security specification, in particular

- If the problem is related to the implementation of the specification, we generate a **bug**. Remember that not all bugs are vulnerabilities, but only those that refer to security specifications.
- If the problem is related to the definition of the specification, we have a **design issue**.

In both cases we talk about vulnerabilities. Always remember that vulnerabilities and exploits are different, in fact the former are way to leverage a bug and make a program do something it wasn't designed to. In this sense, exploiting a vulnerability is like programming a computer with a different instruction set than the one it was designed with. Equivalently, an exploit is a way to leverage a bug to do something with privileges we don't have.

10.2 Life of a vulnerability

Between the deployment of a vulnerability (i.e. of a program with a vulnerability) and of the relative patch, we find many intermediate stages. In particular, after a vulnerability is released

0.a The vulnerability is discovered by a person at time t_e.

0.b The vulnerability is discovered by the vendor at time t_d.

1. The vulnerability is disclosed publicly at time t_0. Time t_0 is called **zero day**.
2. Countermeasures are applied.
3. A patch is released at time t_p.
4. The patch is completely deployed at time t_a.

In this sequence of events we have to highlight two important things

- The first two events can happen in any order and the time that elapses between when the vulnerability is initially discovery and when it's completely disclosed is variable. In particular

  - A person might find a vulnerability and keep it for himself or herself.
  - A person might find a vulnerability and disclose it to the vendor that can start working on a patch and finally disclose the vulnerability publicly.
  - The vendor might find the vulnerability first and start working on the patch before anyone else discovers it.

  Also notice that the time at which a vulnerability is first discovered isn't precise in fact one might have found it hand kept if for itself.

- The time between t_p and t_a isn't negligible, especially for older systems, and can be used by attackers to exploit the vulnerability.

Also notice that we can divide the attacks in two categories, depending on when they occur

- **Zero day attacks** occur from when the vulnerability is discovered to when it's disclosed publicly.
- **Follow-on attacks** occur from when the vulnerability is disclosed publicly to when the patch is completely deployed.

10.2.1 Reducing attack risk

Now that we know the different life-phases of a vulnerability, we should understand how to reduce the risk of attacks. We can start saying that the risk is higher if more people know about the vulnerability (because more minds can work on an exploit), in particular the risk

1. Is **low** when **few people** have discovered the vulnerability and increases the more people discover it.
2. Is **maximum** when the vulnerability is **disclosed publicly**.
3. **Decreases** after being disclosed because the community start using countermeasures (e.g. stop using the vulnerable service) and finally the vendor deploys a patch.

This means that to reduce the risk of an attack we have to reduce the time between when the vulnerability is discovered and when it's disclosed because the risk increases in this span of time. In this context, the most critical situation is when a person discovers a vulnerability and has to decide what to do with it. The actions that the discoverer decides to take is called **disclosure policy**. In particular, when a person discovers a vulnerability, the best policies he/she can apply are

1. The person (or, even better, directly the vendor) who discovers the vulnerability discloses it to the vendor only. The vendor can now work on the patch. The vulnerability is publicly disclosed only when the patch is deployed. Notice that a patch contains the description of the vulnerability because one can check what changed between the old and new version and find out what was wrong with the former one. This policy is called **coordinated disclosure** and it exists an ISO standard to define how it should be applied. The name comes from the fact that the discoverer and the vendor agree on how and when the vulnerability should be disclosed.
2. The person who discovers the vulnerability immediately discloses it publicly. This policy is called **complete disclosure**.

Even if these policies reduce the risk of attacks, always remember that, even after patching a program, the time between the release of the patch and the full deployment can be used by attackers to exploit the vulnerability.

Also notice that, originally the coordinated disclosure policy was called responsible disclosure, however this name has been changed because it would make publicly disclosing look irresponsible. However, complete disclosure is a good way to reduce the risk of attacks.

10.3 Design principles

When designing a software, there are no rules to avoid vulnerabilities, however we can follow some principles and tips to reduce security bugs to the minimum. In particular, we should

- **Reduce the privileged parts to a minimum**. Some programs require privileged execution to run some instructions (e.g. to open a file). It's important to give the privileges to the program as late as possible and to revoke them as soon as possible. The reason why is that the user running the program doesn't have such privileges (e.g. the user runs a program on a remote server or the program needs privileges only to read a file), hence the window in which he/she can use them has to be as narrow as possible. In other words, when a user executes a software that has privileges, the only barrier between the user and the privileges is the software. If the software misbehave, the user can do things he/she isn't allowed to do.
- **Keep It Simple, Stupid** (KISS). Anything that is complex, tend to have behaviours we can't notice and that might generate vulnerability. To avoid this problem we should build software that is as simple as possible. Simple software is also easier to test, hence to check if the it has bugs or weird behaviours.
- **Discard privileges definitely as soon as possible**. Operating systems have instructions to activate and deactivate privileges for a program. Say for instance a program has to open a file (can only be done with root privileges), the OS can assign root privileges to the program and after having opened the file it can deactivate them. However, since the privileges are only deactivated, a vulnerability might allow an attacker to reactivate them. For this reason OSs have also commands to permanently delete the privileges from the program so that an attacker can't reactivate them. This operation should be done as soon as possible.
- Design a program according to the **open design principle**. This means that, the security of the program shouldn't relay on the fact that the source code of the program is not known. Notice that, this doesn't mean that the software should be necessarily open source. Even if a software's source code is not disclosed, the designers should always program it thinking that the code is known by the attacker (remember that one can always de-assemble the executable). Being open-source or not should only be a choice in terms of copyright and money, not security.
- **Take care of concurrent programming**. Concurrency in software is very tricky and it's easy to get it wrong. In particular there exists a class of vulnerabilities, called race-conditions, that are related to timing and concurrent access to a resource. Race conditions verify, for instance, when a security check is happening and an attacker might be faster or slower than the check and get access to a system even if he/she hasn't the privileges.
- Design a program to be **fail-safe and default-deny**. This means that, when a security check fails (i.e. it doesn't work), the software should deny access to the user.

- **Avoid the use of temporary files**. When we are writing a program, we should avoid writing temporary information in a file because, the file is used during the execution of the program and in the meanwhile an attacker might be able to access it and to change it, hence compromising the running program. Notice that, OSs have operations to safely use temporary files that handle the complexity of creating a temporary file and check its integrity.
- **Avoid using unknown, untrusted libraries**. This principle scales up also to services, in fact when a machine is using a service of a third party, it's the third party that provides the result of a service. This means that we should be careful about what services we use because we can't control the service.
- **Filter the input and the output**. Input and output are the first things an attacker tries to use to exploit a vulnerability because he/she can use them to interact with the program. I/O is therefore the main surface of contact with the software.
- **Not write any own cryptographic, password and secret management code**. Since it's hard to get cryptographic code right, we should always use trusted libraries that have been tested and verified. Furthermore, we should also use only trusted sources of randomness (e.g. `/dev/urandom` in Unix systems).

Chapter 11

Buffer overflows

To analyse buffer overflows, we will consider

- Machines running Linux (version later than 2.6).
- ELF executables.
- 32-bit x86 processors.

This assumptions are made to analyse the vulnerability easier, however the same concepts apply to whatever system and program.

11.1 Process basics

The operating systems executes a program (i.e. an executable) inside a process in a way that it thinks that it's the only process running on the machine. This means that the process thinks it can use all the memory of the machine. What actually happens is that the process works in a virtual addressing space and the operating system maps the virtual memory addresses of the process to the physical memory addresses of the machine. In other words, the OS allocates physical memory to the process and maps (using a table) the physical addresses to the virtual addresses used inside the process.

This allows, for instance, to map the shared libraries that each process uses to the same physical memory. Also remember that

- **The stack grows towards lower addresses**.
- **The heap grows towards higher addresses**.

11.2 Buffer overflow

Studying buffer overflows requires to look into the memory of a process and to analyse some assembly code so, hold thigh, get some coffee and read Appendix A if needed.

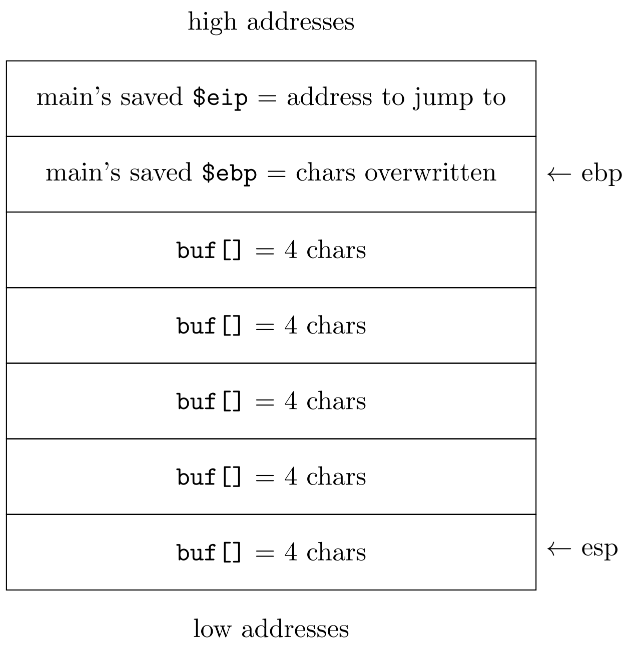


Figure 11.1: Buffer overflow.

11.2.1 Vulnerability

Buffer overflows exploit a vulnerability in how the function epilogue (see Appendix A.3) is executed. In particular, the `ret` instruction can't check the value of the restored `%eip`, hence if we can modify (i.e. overwrite) the stored `%eip` we can force the processor to execute whatever we want (i.e. the instruction at the address we overwrite on the stored `%eip`).

This behaviour is a vulnerability if the function allocates a buffer on the stack (i.e. a vector variable in the function) and there is an instruction that writes into the buffer without checking the size of what's written. Many C functions that take input from the user don't check the size of the input, hence this check should be done by the programmer to avoid this vulnerability. An example of vulnerable program is shown in Listing 11.1.

```c
int main() {
    vuln_func();
    return 0;
}

void vuln_func() {
    char buf[20];
    gets(buf);
}
```

Listing 11.1: A vulnerable function in C.

The vulnerability comes from the fact that we can write more characters (or whatever the buffer contains) than the size of the buffer and since the characters are written in memory from low addresses to higher addresses (i.e. towards the base pointer) we can overwrite the stored `%eip`. The behaviour we want (considering the function in the example 11.1) is shown in Figure 11.1.

11.2.2 Exploit

Now that we know the vulnerability, we have to understand how to exploit it.

Where to jump

The first thing we have to sort out is where to jump, namely what address to write on the stored `%eip`. The idea is to jump to a valid piece of memory that contains a valid instruction that we can execute. Some examples are

- **Another function** of the program in execution.
- A **library function** (usually library functions are loaded in the virtual memory of the process).
- A part of the memory we control, like the **stack**.

We will focus on the third option, namely we will write in the buffer the instructions we want to execute and then we will jump back to the start of the buffer to execute them. However, we don't know where the buffer is (i.e. the exact address of the buffer) because each machine has different processes running and even if we consider machines with the same processes, each machine has different environment variables. This means that the buffer will never be at the same address on different machines. What we can do to solve this issue is to guess the address of the buffer on a machine we control and try to use it on the machine we want to attack.

Guessing the buffer's address To guess the buffer's address we can use a machine that is as close as possible (in terms of processes running and environment variables) as the target machine (i.e. the one we want to attack) and use one of the following strategies

- Use a debugger to check the value of the `%esp` just before the return instruction, since the `%esp` points to the last variable in the function.
- Print the value of the `%esp` directly from the function. Basically we should rewrite the program adding a `printf` that prints the value of the `%esp`.

Notice that, the second option is rarely used because we almost never have the source code of the program. On the other hand the first option can be applied on executables because we only need to attach a debugger to the program.

The value obtained is imprecise because of what we said before, hence we should consider an offset with respect to the guessed address. This is however a problem because if we get the address wrong, even by a byte, we might execute wrong commands and ruin the exploit. To solve this problem we can fill the beginning of the buffer with `nop` instructions so that, if we get the address wrong (i.e. if the guessed address is too big) we can execute some `nop`s and then safely start executing the exploit's instructions. The sequence of `nop`s is called **nop sled**. Notice that the precision of the guess and the success of the buffer depends on the size of the buffer. In particular, the larger is the buffer the more space we have for our instructions and the more imprecise we can be (thanks to the `nop` sled).

To sum things up, the result we want to obtain in shown in Figure 11.2 and the input of the program should be

```
nop nop nop nop nop nop nop nop exploit_instructions guessed_esp
```

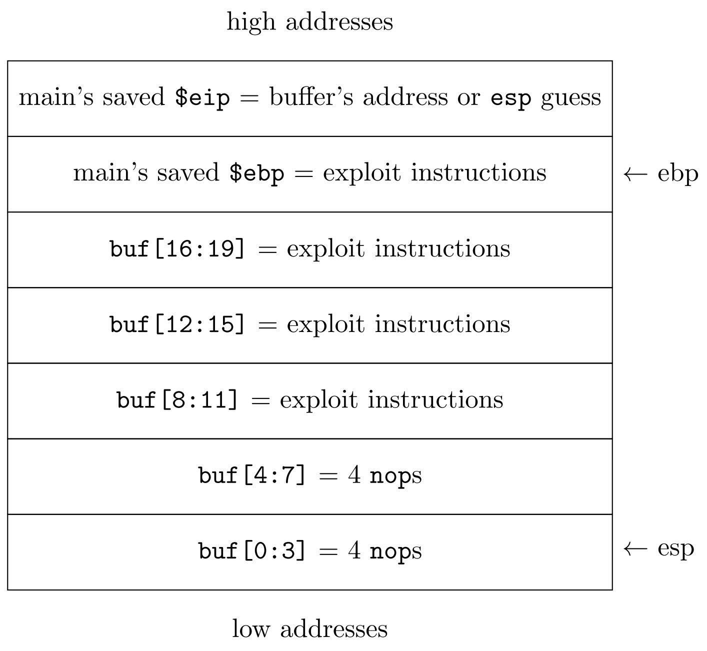

high addresses
main's saved $eip = buffer's address or esp guess
main's saved $ebp = exploit instructions
← ebp
buf[16:19] = exploit instructions
buf[12:15] = exploit instructions
buf[8:11] = exploit instructions
buf[4:7] = 4 nops
buf[0:3] = 4 nops
← esp
low addresses

Figure 11.2: Memory configuration for the buffer overflow's exploit.

What to execute

Now that we know where to jump and how to do it, we should decide what instructions to execute (i.e. the instructions we want to write into the buffer). Usually, an attacker might want to spawn a shell (notice that if the program is running with privileges we don't have, we get a privileged shell) or to execute instruction to open a TCP connection or a VPN (or whatever an attacker might want to do); we will focus on spawning a shell. Remember that these instructions are called **shellcode** because they where originally used to spawn a shell, however the word shellcode usually refers to whatever set of instructions used in a buffer overflow exploit. In general a shellcode is a set of instructions that has to be

- **Portable**.
- **Location independent** (i.e. it doesn't depend on its location in memory).

The easiest way to spawn a shell is to call the system call (i.e. a function that is executed in kernel mode) `execve("/bin/sh")` to spawn a new process and execute the specified program (i.e. `"/bin/sh"`). In Linux, a system call is executed through a software interrupt which is called with the instruction `int 0x80`. Basically the instruction `int 0x80` tells the processor to execute an interrupt (`int`) and that the interrupt is a syscall (`0x80`). The instruction `int 0x80` only tells the processor to execute the system call but it doesn't say what program to execute. In particular, a syscall interrupt needs

- What system call to execute in the register `%eax`. This means that in our case we have to put the code of `execve` (i.e. `0xb`) in the register `%eax`.

```
%eax <- 0xb
```

Remember that `execve` has the following signature.

```
int
execve(const char *path, char *const argv[], char *const envp[]);
```

- The first parameter of the function (if needed) in the register `%ebx`. In our case the first parameter is the name of the program to execute, hence the string `"/bin/sh"`.

  ```
  %ebx <- "/bin/sh"
  ```

- The second parameter of the function (if needed) in the register `%ecx`. In our case the second parameter is the vector of arguments `argv` that should contain only the name of the program (i.e. `"/bin/sh"`) because the first value in the arguments vector is always the name of the program and we don't want to pass any other parameters. Long story short we want to put the address of `{"\bin\sh", null}` in `%ecx`.

  ```
  %ecx <- {"/bin/sh", null}
  ```

  Notice that we have to end the argv vector with a null pointer because the manual (i.e. `man execve`) tells us to do so (quoted, *The argument argv is a pointer to a null-terminated array of character pointers to null-terminated character strings.*).

- The third parameter of the function (if needed) in the register `%edx`. The third parameter is a vector of arguments not passed in `argv` (see `man execve` and `man 7 environ`). Since we don't want to pass any other parameter, we can put `null` in `%edx`.

  ```
  %edx <- null
  ```

Now that we know to put in each register, let us figure out how to do it. In particular, we should store in the buffer all the parameters to pass to the `execve` function. The easiest way to do it is by using the configuration in Figure 11.3. Notice that, this configuration is enough for our purposes

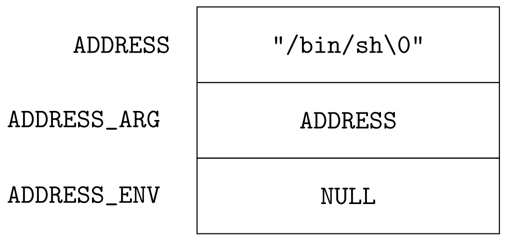


Figure 11.3: Memory configuration for the execve parameters.

because

- At the address `ADDRESS` we find the string `"/bin/sh"`.
- At the address `ADDRESS_ARG` we find the argument vector, in fact the first cell of the vector contains the address of the string `"/bin/sh"` and the next cell contains `NULL`.
- At the address `ADDRESS_ENV` we find the `NULL` that has to be passed to `envp`.

Now we have to understand how to build this configuration in memory. In particular,

1. The string `"/bin/sh"` is passed from input. Notice that this string doesn't contain the string terminator `\0`. The reason why will be explained later.

   ```
   .string "/bin/sh"
   ```

2. We have to add the string terminator `\0` 7 bytes after `ADDRESS` because `"/bin/sh"` starts at address `ADDRESS`.

   ```
   movb   0x0, 0x7(ADDRESS)
   ```

3. We have to add the address of `"/bin/sh"` (i.e. `ADDRESS`) after the string (i.e. 8 bytes after `ADDRESS`).

   ```
   movl   ADDRESS, 0x8(ADDRESS)
   ```

4. We have to add a `NULL` pointer (i.e. 4 zeros) after `ADDRESS_ARG`, i.e. 12 bytes (8 of the string and 4 of the address) after `ADDRESS`.

   ```
   movl   0x0, 0xc(ADDRESS)
   ```

Now that we have prepared the memory we can move the arguments in the registers `%eax` to `%edx` needed by the system call. This can be done with the following instructions

```
movl   0xb, %eax               // store the execve identifier in eax
movl   ADDRESS, %ebx           // store the address of "/bin/sh" in ebx
leal   0x8(ADDRESS), %ecx      // store the address of the arg vector in ecx
leal   0xc(ADDRESS), %edx      // store the address of the environment in edx
int    0x80                    // call the interrupt
```

Jump and call Now that it's all set up, we only need to understand how to obtain the address `ADDRESS`. Notice that, differently from the address of the buffer, in this case we have to be precise to the byte, hence we can't just guess the address and use a `nop` sled. The reason why is that we don't have to guess a point from where start executing code but we have to guess the address of a string. To solve this issue we can use the instructions `jump` and `call`. Before understanding how these instructions are used, let us remind of what they do

- The `jump address` instruction allows the processor to jump to a specific address. This instruction can be used also for relative jumps, namely we can pass to it an offset and it will jump offset instructions after (or before if the offset is negative) the instruction pointer.

- The `call address` instruction
  - Pushes the next value of the instruction pointer `%eip` (i.e. the address of the next instruction) onto the stack.
  - Jumps to the address passed as parameter. Notice that, as for the `jump` instruction, we can do relative jumps, hence pass an offset to the instruction.

We can use the aforementioned instructions to write the following shellcode

```
jmp offset-to-call      // jmp takes offsets
popl %esi               // pop ADDRESS from stack to ESI, from now on ESI==ADDRESS
movb 0x0, 0x7(%esi)
movl %esi, 0x8(%esi)
movl 0x0, 0xc(%esi)
movl 0xb, %eax         // execve starts here
movl %esi, %ebx
leal 0x8(%esi), %ecx
leal 0xc(%esi), %edx
int 0x80
movl 0x1, %eax         // exit(0)
movl 0x0, %ebx         // exit(0)
int 0x80               // exit(0)
call offset-to-popl
.string \"/bin/sh\"
```

Let us analyse step by step this shellcode,

1. The first instruction allows us to jump to the `call` instruction. The `offset-to-call` label is easy to compute since we know the length of each instruction between the jump and the call.

2. After the jump we execute the `call` instruction. This instruction saves the address of the next instruction on the stack, hence if we put the string `"/bin/sh"` after the `call` instruction, then we save the address of `"/bin/sh"` on the stack. The `call` instruction also starts executing the instruction at distance `offset-to-pop` from the current address.

3. The `call` instruction is used to jump back to the instruction after the jump. This instruction pops the value pushed on the stack by the `call` and saves it in the register `%esi` (which is the one usually used for string addresses). The value popped is the address of `"/bin/sh"` (because of what we said at point 2), hence from now on the register `%esi` will contain the address of `"/bin/sh"`. This means that we have found a way to resolve the address `ADDRESS`.

Now we simply have to replace the address `ADDRESS` with `%esi` because after the `popl` instruction that register contains the address of the string `"/bin/sh"`. Remember that, since the string `"/bin/sh"` is at the end of the code, it's added at the end of the buffer, and the memory configuration in Figure 11.3 is obtained going to bigger addresses, hence we do not overwrite any instruction of the shellcode.

Notice that after executing the `execve` we have added some other instructions, in particular we have called another system call. If we check the value in `%eax`, we understand that the system call with code `0x1` is the `exit` function to which we pass `0x0`. Basically, we are calling `exit(0)` to safely quit the process after spawning the shell (creating a child process).

If we assemble the shellcode above we get the following machine code

```
e9 26 00 00 00
5e
89 76 08
c6 46 07 00
c7 46 0c 00 00 00 00
b8 0b 00 00 00
89 f3
8d 4e 08
8d 56 0c
cd 80
b8 01 00 00 00
bb 00 00 00 00
cd 80
e8 cd ff ff ff
2f                  # /
62 69 6e            # bin
2f                  # /
73 68               # sh
```

If we put this machine code in a string we obtain

```
"\xe9\x26\x00\x00\x00\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00
\x00\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xb8
\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xcd\xff\xff\xff/bin/sh"
```

This string can be used as input of a program, together with the nop sled and the address where we guess the buffer is, to exploit the buffer overflow vulnerability and spawn a shell. However, as we can notice this string contains many `\x00` characters. These characters are interpreted as string terminators, hence when the string is read, only `\xe9\x26` is actually read because there is a string terminator after these characters. To solve this problem we have to use different instructions in the shellcode so that the machine code has no `\x00` characters. In particular we can,

- Replace the `jmp` instruction with a `jump short` instruction.
- Use an instruction to compute the number 0 and use it instead of saving `\x00` directly. For instance we can compute `xorl %eax,%eax` and than use `%eax` instead of `0x00` (for instance, `movb %eax, 0x7(%esi)`).
- Use shorter registers.

Applying these tricks to the shellcode above we obtain

```
jmp short offset-to-cal
popl   %esi
movl   %esi,0x8(%esi)
xorl   %eax,%eax
movb   %eax,0x7(%esi)
movl   %eax,0xc(%esi)
movb   $0xb,%al
movl   %esi,%ebx
leal   0x8(%esi),%ecx
leal   0xc(%esi),%edx
int    $0x80
xorl   %ebx,%ebx
movl   %ebx,%eax
inc    %eax
int    $0x80
call offset-to-popl
.string \"/bin/sh\"
```

This shellocode, when translated in machine code, doesn't contain zeros, hence can be used as input of the vulnerable program. Notice that, in some cases we might also want the input to contain only some types of characters, hence we should change the shellcode so that all the constraints are met (luckily there exists tools that do this automatically). For instance, one might want to use a shellcode in a website that only accepts printable characters.

11.2.3 Alternatives

Exploiting buffer overflows by jumping back in the stack has the advantage that the exploit can be executed independently from the location of the buffer because, once we know (even approximately) the location of the buffer, the shellcode doesn't depend on it. However, this technique also has some disadvantages. In particular,

- **The buffer has to be big enough to fit the shellcode**. In some cases, it's possible to use tools to reduce the size of the shellcode so that it can fit in tight buffers, however in some cases the buffer is just too small to fit any useful set of instructions.
- **The area of memory in which we execute the shellcode has to be marked as executable**.
- **We must have a way to find the address of the buffer reliably**.

Reliability of an exploit The disadvantages of jumping in the stack can be summed using the concept of reliability. In particular, an exploit is reliable when its preconditions (i.e. the conditions under which it works) are always met. Reliability is also related to the concept of weaponisable vulnerability. A vulnerability is weaponisable if it's possible to write an exploit every time (i.e. in every sitaution and condition). Basically, the exploit of a weaponisable vulnerability is very reliable and works every time.

In the case of jumping in the stack, the exploit isn't very reliable since it doesn't work for all buffer sizes and it's might be hard to find the buffer's address.

Notice that, reliabilty is very important since, when an exploit is reliable, it leaves less traces (if the buffer overflow is successful the program quits gracefully, otherwise we might get a segmentation fault) and it's harder for the attacker to understand that an attack has happened.

Since jumping back in the stack isn't that reliable (especially on modern machines) we will analyse some other alternatives.

Environment variables

Many operating systems use environment variables, that is variables that can be used by multiple programs and that store all sort of things (e.g. executable paths). These variables are stored at the bottom of the stack at a very predictable address. This means that, if we have the direct control of the target machine (e.g. we only want to do privilege escalation) we can

1. Create a new environment variable, say `$EGG`.
2. Write the shellcode in the environment variable `$EGG`.
3. Send the CPU to the environment variable. This means that the input of the program is just the address of the environment variable repeated multiple times to be sure to overwrite the *eip*.

Notice that, the big advantage of this exploit is that the address of the environment variables can be found precisely. However, this exploit only works on a local machine because we need to create a new environmental variable. Furthermore, the exploit might delete the environment and, like in the return to stack exploit, the memory has to be marked as executable.

System libraries

Another possible idea is to jump to the code of a system library function. Since C programs usually import the `libc` library, this exploit is called **return to libc**. Like for the environment variable case, the address in which the libc is loaded (and the address of a specific function) can be found precisely, hence we can definitely know the address where to jump (i.e. the address with which to override the `eip`). The main advantages, with respect to the other two methods, are that this exploit can be executed remotely (i.e. on a remote machine we don't control) and that we don't have to care about the fact that the memory is executable or not because a library function is always executable. The only issue with this exploit is that we have to carefully prepare the stack before jumping to the library function because the called function requires a well formatted stack to work properly. In particular, we have to

- Place on the stack the arguments to be passed to the function. This means that, we have to fill the buffer with the arguments. Notice that how the parameters have to be passed to the function, depends on the calling conventions of the operating system.
- Replace the `eip` with the address of the function.

Change the overwrite subject

Until now, we have always overwritten the `eip`, however we can also overwrite

- A function pointer, if present in the source code.
- The `ebp`. This technique is called **frame teleportation**.

Heap overflows

Buffers can be allocated also on the heap (instead of the stack), hence there exists techniques, called **heap overflows**, to exploit the same vulnerability of buffer overflows, but on the heap.

11.3 Defence measures

Since it's impossible to have one method that allows to defend against all possible exploits, we have to use a layered structure that, for each layer adds some security mechanism to reduce the attack chances. In particular we can add security layers at,

- The **source code** level.
- The **compiler** level.
- The **operating system and hardware** level.

11.3.1 Source code

C and C++ make it easier to create programs with vulnerabilities because they are low level languages and have functions that do not check the length of the input. To mitigate this issue we could switch to memory safe languages like Java, however this isn't always possible for performance or legacy reasons. In such cases, to reduce the number of vulnerabilities in the source code we can

- **Use safer libraries** that include functions that include a parameter to express how many parameters to read from input. For intance the function `strncpy()` in the standard library takes as input also the number of characters to copy from the source string to the destination string. The same can be done by the function `strlcpy()` in the BSD library. Both replace the less secure function `strcpy()`. The same happens for many other functions that handle user input and strings.
- **Educate developers** to use low level function correctly and avoid the use of unbounded functions without proper checks.
- **Include security in the System Development Life Cycle**.
- **Test** a lot the code to highlight weird behaviours and possible vulnerabilities.
- **Use source code analysis** that allow to highlight possible sources of bugs when writing code.

11.3.2 Compiler

The compiler can help secure programs against buffer overflows, in particular it can

- **Run source code analysis** to check if the code might have some bug and print out a warning to the developer.
- **Randomise the order in which function variables are pushed on the stack**. This countermeasure makes it harder to override the stored `eip` with the correct address (because every time the buffer starts at a different offset from the stored `eip`). However, an attacker might repeat the address many times and hope that one overrides correctly the `eip`.
- **Use canaries to check the integrity of the stack**.

Canaries

A canary is a value placed on the stack by the compiler to ensure that, after the function execution, the stack hasn't been corrupted. More precisely,

1. During the function prologue the caller pushes the canary after the saved `ebp`.
2. During the function epilogue, the called function checks that the canary hasn't changed and
   - If the canary hasn't changed, the function returns normally.
   - If the canary has changed, the function fails and the program exits with an error.

The value of the canary is generated at random every time a function is called and the code to generate it is added by the compiler.

`GCC` has the `-fstack-protector` option to add the canary. This option is active by default, so we don't have to specify it and when we compile a program we already get the canary.

There exists different types of canaries,

- The **random canary**. A random value is added on top of (i.e., at the lower address with respect to) the stored `ebp` during the function prologue and during the epilogue we check if the value hasn't changed. This type of canary is the one we have initially described.
- The **terminator canary**. A canary of all zeros is added on top of (i.e., at the lower address with respect to) the stored `ebp`. This canary works because when the user inserts a string from input, the program reads it until the first character `\0`, hence an attacker can't try to overwrite the canary with all zeros (to pass the canary check) and then overwrite the `eip` because everything after the zeros (i.e. the address to overwrite the `eip`) won't be read.
- The **xor canary**. The xor between a random value v and the value of the stored `eip` (or `ebp`) is added on top of (i.e., at the lower address with respect to) the stored `ebp` during the prologue and during the epilogue, the function checks that the value hasn't changed xoring the value v with the stored `eip` (or `ebp`). Basically, we are checking the integrity of the `eip` (or of the `ebp`). This technique is rarely used because it requires some extra computation (two xors) which has to be avoided in those applications where performance is critical.

The most used technique among these is the random canary because it's a good trade-off between extra computation and security grants.

11.3.3 Operating system

The last layer of security is added by the operating system (and by the hardware). The two most common countermeasures against buffer overflows are

- **Making the stack non-executable**. All modern operating systems use a bit, called NX (Not-eXecutable) bit, that tells for every page if the page can be executed or not. In particular, if the bit of a page is set to 1, the bytes on that page isn't executable (i.e. the bytes can't be interpreted as instructions and executed). This means that the OS can set all the NX bits of stack's pages (including where environment variables are) to 1. When this technique was introduced, it caused some problems because some problems exploited the fact that the stack was executable (e.g. in function pointers). This countermeasure can be bypassed
  - Using **attacks that don't need to execute instructions on the stack** (e.g. return to libc).

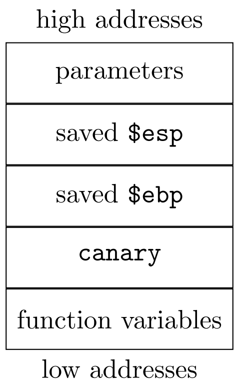


Figure 11.4: A canary added to a function's stack frame.

  - Using **Return Oriented Programming (ROP)** that allows to place pieces of function (called gadgets) followed by the return instruction one above the other so that they can be executed one after the other.

- **Randomising the stack location**. Address Space Layout Randomization (**ASLR**) allows the OS to place the stack (and the others blocks of memory like the heap and the space for environment variables) at randomised locations (i.e. at a certain distance from the position they where supposed to start from) to make it impossible to predict the address where to jump to (i.e. the address to overwrite the stored `eip` with). Usually the stack is moved by an offset of 8 MB. Executables can run in a random position in memory only if they are independent from where they are loaded. Such executables are called **Position Independent Executables** (PIEs).

Chapter 12

Format string bugs

One of the most basic functionalities offered by all programming languages is to print a string to the standard output (or, in general, to a file). Moreover, many languages allow to print something using a format string, which is a string that contains placeholders which are replaced by the values of some variables of the program. Consider for instance C's `printf` that takes as input a string (i.e. the format string) and a variable number of arguments. If we write

```
printf("%x -> %d", i, i);
```

we want the program to output a string that contains the hexadecimal representation of `i`, followed by an arrow and the decimal representation of `i`. Basically, `%x` and `%d` are the placeholders and they are replaced by the values of `i`, interpreted as an hexadecimal first (because that's what the `%x` placeholder represents) and an integer then. Notice that, depending on the number of placeholders we insert in the string, we should pass an equal number of values to the function. However, this number is not known a-priori, because the API doesn't know which format string is used every time the function is called. In other words, the `printf` is a **variatic function** because, after the format string, we can pass an unknown number of values.

Before introducing why the `printf` function is a vulnerability, we should, say that even if this type of bugs is called format string, it can be applied to whatever variatic function (even those that do not use format strings), and the name only comes from the fact that the first function exploited was the `printf`.

12.1 Vulnerability

To understand why variatic functions can be vulnerable, we should understand how do they work. We will focus on the `printf` function, however, the same reasoning is applied to any other variatic function.

When a program calls a `printf`, the function initially counts the number of placeholders in the format string and then puts all the variatic arguments on the stack. When the function has to replace the placeholders with a value it goes on the stack and takes the correct values. For instance if the function has to replace the third placeholder it goes 3 words above (because of how the variables are added to the stack) the place where the first value was placed. To make things clearer consider the following program.

```
int main(int argc, char[]* argv) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int c = atoi(argv[3]);

    printf("%x, %x, %x", a, b, c);
    return 0;
}
```

When the printf function is called, the main pushes the variable `c` on the stack, followed by `b`, `a` and the format string. Basically the configuration of the stack is shown in Figure 12.1. When the function has to assign a value to the format string's placeholders it looks on the stack. In particular, the function replaces the first placeholder with the value in the memory cell above (i.e., at higher addresses), which in our example is `a`.

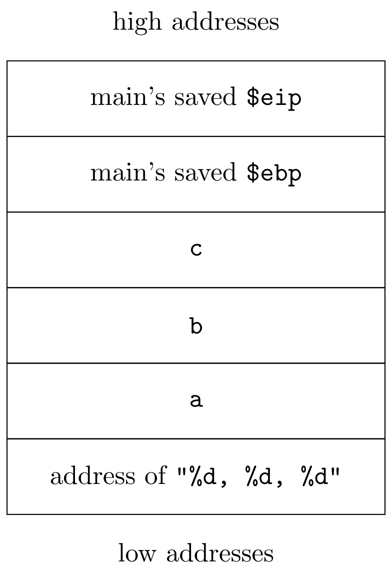


Figure 12.1: Configuration of the stack when a variatic function is called.

Now that we know how the memory is configured and how variatic function works, we can try to do something weird. Say we write the following program that tries to print the first argument passed to the program.

```
int main(int argc, char[]* argv) {
    printf(argv[1]);
    return 0;
}
```

The program listed above is a valid C program and in particular the call to the `printf` function is valid, even if it's used improperly. In particular, the `printf` function is defined as (`man 3 printf`)

```
int
 printf(const char * restrict format, ...);
```

hence the first parameter we pass to the function should always be the format string and not a variable. That being said, the program above compiles and executes correctly and actually prints the first command line argument.

However, strange things happen when we pass a format string (e.g. `"%x %x"`) to the program. What happens is that the program correctly interprets the value of `argv[1]` as a format string, however, since we didn't pass any variable after the format string, it goes on the stack and prints the two words above the format string. If we look at the memory configuration when the `printf` is called (Figure 12.2) we can easily see that the program outputs the saved `ebp` and the saved `eip` because above the format string we have the saved `ebp` and `eip`. This behaviour is a vulnerability

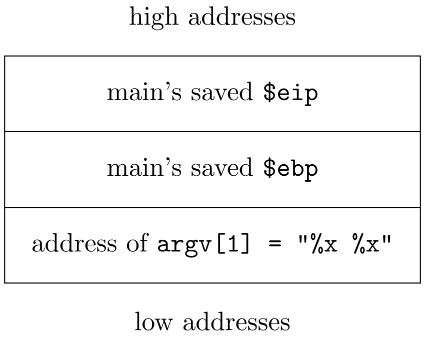


Figure 12.2: Configuration of the stack when a vulnerable variatic function is called.

because it allows an attacker to do things (e.g. read the stack) it shouldn't be allowed. Notice that a `printf` function isn't always vulnerable, but it becames a vulenrability when it's used improperly (i.e. when the first argument passed is a variable and not a constant format string).

12.2 Exploit

Being able to read the stack is dangerous, in fact one might be able to read the encryption keys putted on the stack by SSL or in general any address. However, we would like to execute arbitrary code.

12.2.1 Printing the format string

The first step towards this goal is to try and print the format string inserted by the user (i.e. the string in `argv[1]`). The string is somewhere in the activation record on the main, hence we should try to put as many placeholders as possible to reach the desired string. Here we find our first problem, in fact some function allow the developer to specify how many characters to print, hence we can't write an arbitrary long format string. The solution to this problem is offered by the `%N$x` placeholder (where `x` can be any format) that allows to print the N-th variable. For instance

```
printf("%3\$x", a, b, c)  \\ \ to escape the $ symbol
```

prints the value of `c`. Say that we know that the format string is 20 words above it's address, we can use the following sting

```
"AAAA%20\$x"
```

as input of the program and get as result `AAAB42414141` (where `42414141` is the hexadecimal representation of the string, after replacing `%20\$x` and removing the placeholder). This means that we have successfully printed the format string putted in input.

12.2.2 Writing at a specific address

Printing things can be interesting, however we haven't been able to execute code yet. To achieve this goal we need an interesting placeholder: `%n`. The `%n` placeholder writes, in the address pointed to by the argument, the number of chars (bytes) printed so far. Consider for instance the following code.

```
int i = 0;
printf("hello%n", &i);
```

After it's execution, the variable `i` will contain 5, because the `printf` has printed 5 characters (i.e. the letters of `hello`).

Now we have to understand how to use this placeholder to execute arbitrary code. Say we know that the format string is 2 words before it's address (i.e. we can reach it using the placeholder `%2$x`). If we replace this placeholder with `%2$n`, we are trying to put a number in the cell pointed by the address in the second cell (i.e. the cell that contains the format string). This means that if we use as input the string

```
"\xb\xff\xff\xa5 %2\$x"
```

we can write the number 5 (because we have printed 5 characters) at the address `\xb\xff\xff\xa5`, which might be an interesting and writable address (hence we don't get a segmentation fault).

12.2.3 Write an arbitrary number

Now that we know that we can write to a specific address by putting that address at the beginning of the input string, we have to learn how to write an arbitrary number.

The number we write to the specified address depends on the number of characters printed before the `%n` placeholder. Even in this case, we can use a placeholder; in particular every placeholder allows to specify the number of characters that should be used to represent it. In particular, if we consider the placeholder `%c` (but the same is true for any other placeholder), we can use the syntax

```
%PNx
```

to represent the hexadecimal number with `N` characters and pad it with the character `P`. For instance the instruction

```
printf("%05x", 2);
```

would print the value `00010` because the number 2 in hexadecimal is `10` and we pad it with 3 characters 0 to reach 5 total characters. This means that to print the number `NUM` at the address `ADDR` we have to use the string

```
ADDR%p(NUM-4)c%n
```

as input. Let us analyse this string more in the details,

- At the beginning we put the address `ADDR` where we want to write the arbitrary value. Remember that the address is four bytes long.
- Then we use a placeholder to print a string of (`NUM`-4) characters. Notice that to write the number `NUM` we have to write `NUM`-4 here because we have the four bytes of the address before (which are printed, too).
- Finally we use the placeholder `%n` to write `NUM` (i.e. 4 + `NUM - 4`) characters at the address `ADDR`.

Write an arbitrary big number

It looks like we can write an arbitrary number to a specific address, however we can only specify a 2-bytes number in a placeholder (i.e. `NUM` has to be between 0 and `2^{16}-1=65535`), so we can't actually write some interesting numbers (e.g., we can't write an address) because we need 4 bytes. One solution to this problem could be two use many placeholders, one after the other, however this technique doesn't allow us to write 4-bytes numbers in a short string.

To solve this problem efficiently, we have to use two target addresses. In particular, we have to divide the number we want to write in an upper and lower part (i.e., in the 2 most significant and the 2 least significant bytes). The idea is to use one target address (and a `%n` placeholder) to write the 2 uppermost bytes and the other target address to write the 2 lowermost bytes adjacently to the uppermost bytes so that we have the whole number in a word.

Say for instance we want to write the number `0xbffa76e0` at address `0xbffff6cc`, this technique places (in reverse order, because we are talking little endian) `0x76e0` at address `0xbffff6cc` and `0xbffa` two bytes after (i.e. at `0xbffff6ce`). To achieve this goal we can use the following sting

```
    \xcc\xf6\xff\xbf\xce\xf6\xff\xbf%30214c%pos$hn%18924c%pos+1$hn
```

or in general

```
<target><target+2>%<lower_value-8>c%pos$hn%<higher_value-lower_value>c%pos+1$hn
```

where `pos` is the position on the stack of the format string (i.e. the number of words we have to go through to reach the format string). Before going on, let us point out that we have used the placeholder `%hn` instead of `%n` beause we want to write only 2 bytes instead of 4. Let us explain why and how this string works,

- The format string is pushed on the main's stack when the program starts executing (remember that we are considering that the input string is obtained from `argv`). The memory configuration at this point is shown in Figure 12.3.
- When the naked `printf` (or another variatic function) is called,
  1. `<lower_value>` characters are printed (8 bytes of the addresses and `<lower_value-8>` of the placeholder `%<lower_value-8>c`).
  2. The placeholder `%pos$hn` allows to write `<lower_value>` characters at the second target address which is in the format string. The memory configuration after this step is shown in Figure 12.4.
  3. `<higher_value-lower_value>` characters are printed. Note that this expression comes from $high-printed$, where the printed bytes is 8 (i.e., the initial addresses) + $(lower-8)$, i.e., the characters printed by `%<lower_value-8>c`.
  4. The placeholder `%pos+1$hn` writes `<high_value>` to the first address of the input string (i.e. the one above the second address). The memory configuration after this step is shown in Figure 12.5.

Notice that in this case the decimal value of `0xbffa76e0` (i.e. 30432) was smaller the decimal part of the upper value. This might not happen, hence we have to change a little bit the input string. Say that we want to write `0x76e0bffa`. In this case the placeholder `%<higher_value-lower_value>c` should have written a negative number of characters, which doesn't make sense. To solve this problem we can swap the first and second target and lower and upper values and the general form of the input string looks like follows

```
<target+2><target>%<upper_value-8>c%pos$hn%<lower_value-higher_value>c%pos+1$hn
```

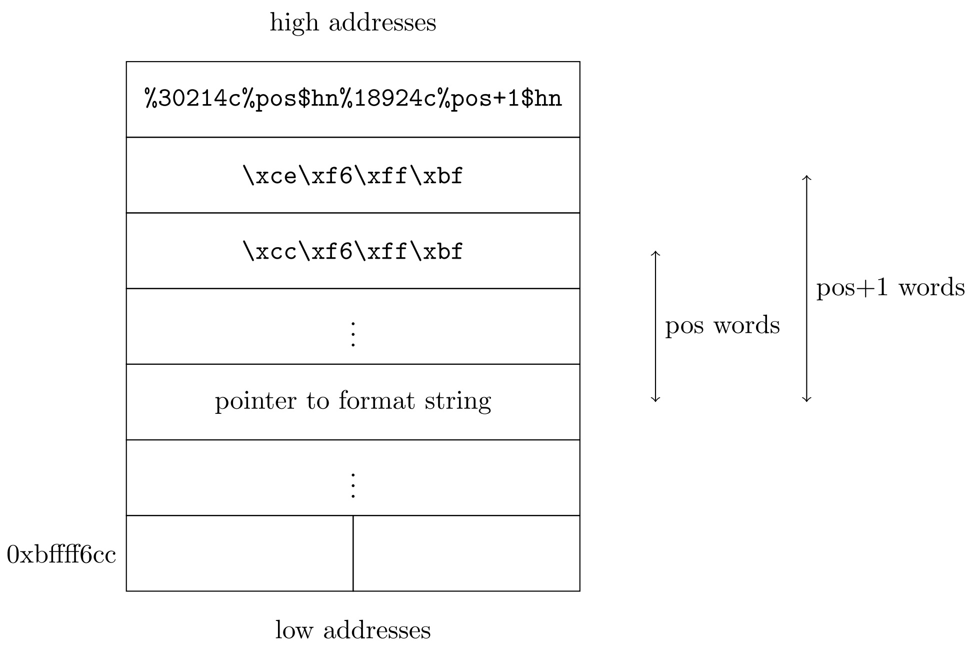


Figure 12.3: Memory configuration before calling the vulnerable print function.

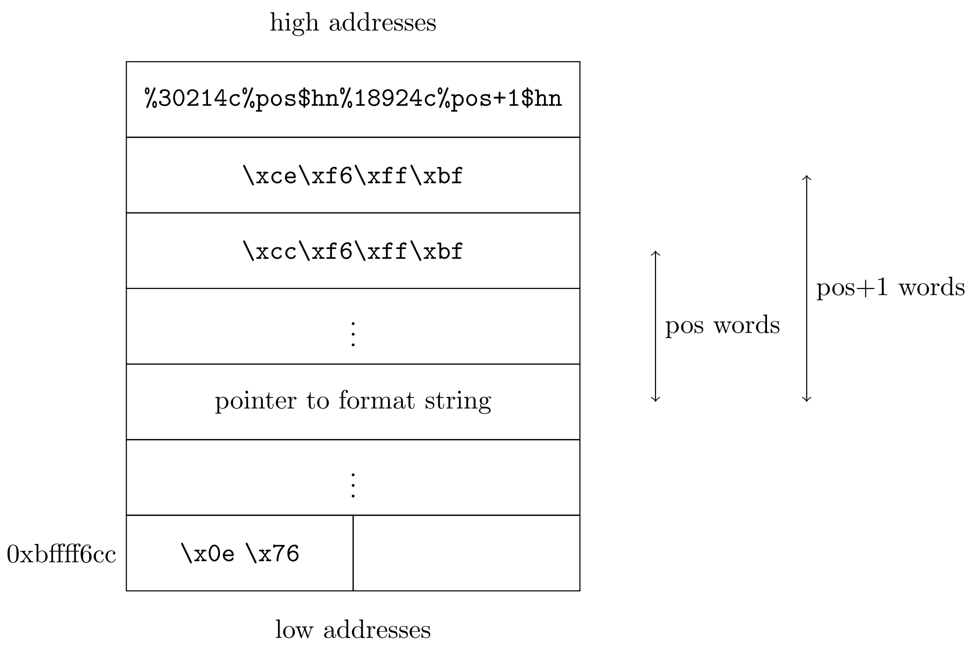


Figure 12.4: Memory configuration after writing the first half of the value.

12.2.4 Where to write

Now that we know that we can write whatever we want, wherever we want, we should understand where to write. We could

- **Overwrite the saved** `eip` of a function, however we would need the exact address (not an approximation like in buffer overflows) where the `eip` is stored, hence this attack is quite hard.
- **Override the Global Offset Table** (GOT). The GOT is used for dynamic relocation of functions.
- **Override exception handlers**.
- **Override C library hooks**.
- **Override function pointers**.

Being able to write on any of this locations, allows us to have arbitrary code execution.

12.3 Countermeasures

Even in this case we should consider a layered approach for securing programs against format strings vulnerabilities and exploits. We will consider

- The **compiler** layer.
- The **operating system** layer.

12.3.1 Compiler layer

A compiler can

- **Randomise the order of function variables on the stack**. This countermeasure can make an exploit unreliable because the attacker might not always find the address of the format string (i.e., the `pos` parameter is wrong), however it's also possible to bypass this countermeasure.
- **Use a XOR canary**. This countermeasure prevents an attacker from modifying the saved `eip`, however, it still allows an attacker to overwrite the GOT or other zones of memory, hence it's not always useful.
- **Warn the compiler that a naked** (i.e., without the actual format string) **format string has been used**. The compiler can only warn the developer because some programs need the possibility to use naked format strings, hence we can't completely eliminate this behaviour.

Also notice that, recently, there have been proposal to fix the libc and include checks on the number of placeholders in format strings so that they can be compared with the number of arguments passed to the function.

12.3.2 Operating system layer

At the operating system level we have two main countermeasures: ASLR and NX. Let us analyse the effectiveness of each one of them.

- **Address Space Layout Randomization** (ASLR) can be very useful because it doesn't allow the attacker to find an address to jump to (because addresses are randomised). Basically, the attack works, in fact one could write an arbitrary value to a specific address, however, it would be hard to find an interesting value to write (e.g., a valid address). That being said, an attacker can always read values form memory (because the attack works) and use them in other exploits. Notice that the attack keeps working because it depends on the relative position of the format string with respect to the function's activation record, and not to the absolute address.
- The **not executable bit** in memory pages is useless in this case because this attack only needs to write in memory, not to execute some code. Notice however that, this countermeasure is effective if an attacker wants to use the format string vulnerability to override a saved `eip` and execute code on the stack (or in an environment variable).

12.3.3 Functions with limited length

Notice that among printing functions we can use in C, some allow to specify how many characters the functions should print. An example is the function `snprintf` (`man 3 snprintf` for more info), with the following signature

```
int
 snprintf(char * restrict str, size_t size,
     const char * restrict format, ...);
```

that allows to print `size` characters of the format string `format` in the string `str`. One might thing that, our exploit is useless when the `snprintf` function is used, since the exploit needs to write a large number of characters (e.g., we use `%30214c`) but the print function prints only a limited number of them. Luckily, the function `snprintf` keeps replacing the placeholders in the format string even if the number of characters to print has been exceeded (simply, it doesn't print them) so our exploit is still effective. The source code for the `snprintf` function for MacOS computers is avalialbe at this website.

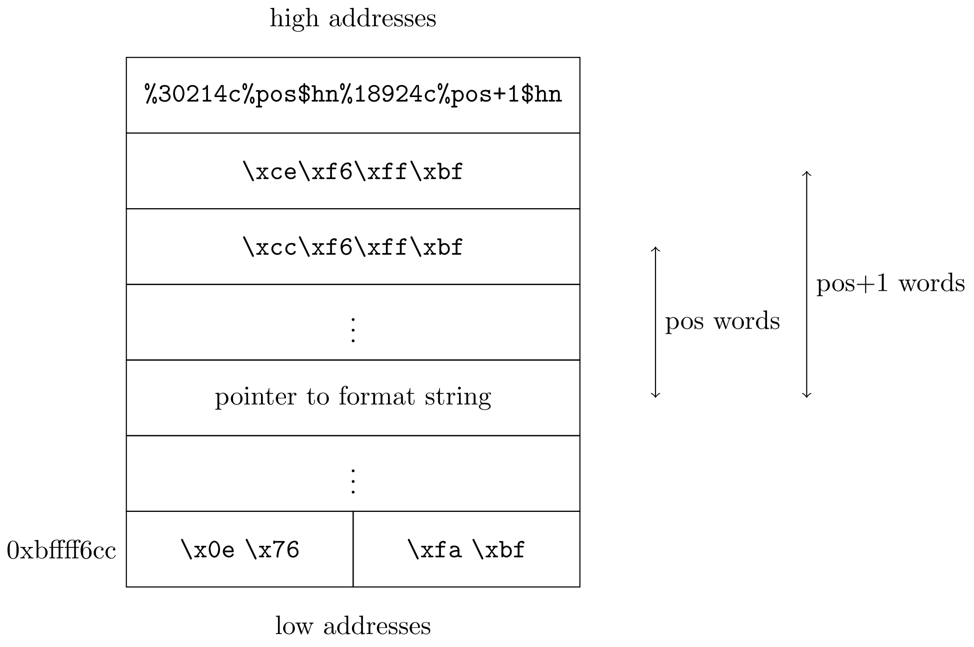


Figure 12.5: Memory configuration at the end of the exploit.

Chapter 13

Web application security

Every computer is, in most cases, connected to the Internet and it uses at least some Web applications, hence we should analyse how to secure such software.

13.1 Introduction

13.1.1 The clients problem

The Internet is based on the client-server paradigm, hence, in every application, part of the software runs on the client's device, which is not controlled by the company that is offering the application. This is important because if the company doesn't control the clients' devices, it can't check what happens there. For this reason **we have to consider the client as not trustworthy**. This means that whatever data comes from the client should never considered worth of trust and should always be checked. This is also true if the client's application performs some checks to validate the user's input, in fact we can't be sure that such checks have been correctly executed or that they have been executed at all. In this sense, client-side checks should only be done to guide the user in inserting the data (i.e., they should be help and convenience controls), while the actual security checks should be done on the server side because it's controlled by the company. For instance, consider a form in a Web application. We can add some JavaScript code to check that the user has entered the correct data type (e.g., a number where a number is requested) but we can't do any security-related checks, because a malicious client could find a way to not execute the validating JS code. Notice that, not only data, but also metadata should be considered not-trustworthy.

Since the client is not trustworthy, **all security checks should be done on the trusted server-side**, even if it means repeating some checks. To sum things up,

- On the **client side** we can do **help and convenience checks**.
- On the **server side** we have to do **security checks**. Putting such checks only on the client side is very dangerous and can lead to vulnerabilities.

13.1.2 The stateless problem

HTTP, the most used protocol on the Web, is stateless. This means that every request issued to a server is handled by the server as a new request and it has no memory of all previously issued requests. However, all applications (i.e., normal applications), are inherently stateful because they follow an execution flow. For this reason, to deliver an application-like stateful experience in a Web-application (which is based on stateless interactions), we have to add some intermediate levels between HTTP and the application layer. This also means that authentication can't be executed by HTTP, hence, it has to be custom-implemented at the application level. This can lead to vulnerabilities, since implementing such a critical element of an application is usually complex (i.e., it's easy to get it wrong and add vulnerabilities), hence it would be better to have authentication build in the lower layers.

13.1.3 Filters

Since the clients can't be trusted, we should always filter the data coming from them. There exist three categories of filters,

- **Allow-lists** (also called whitelists). Allow-lists let us specify what data can pass through the filter (i.e., what data can successfully pass the filter's check).
- **Block-lists** (also called blacklists). Block-lists let us specify what data can't pass through the filter (i.e., what data has to fail the filter's check).
- **Escaping filters**. Escaping filters let us modify the data so that it's not ambiguous. Let us consider a string, some characters have a special meaning (e.g. the `<` in HTML or the `+` in a regex), hence escaping filters replace them with another character (or with a sequence of characters) that allows us to disambiguate the characters. In our example the character `<`, which is interpreted as the opening character of a HTML tag is replaced with `&lt`, which is interpreted as an angular parenthesis (i.e. the actual character `<`).

The cool thing of filtering is that we know which type of filter is better and in which order such filters should be applied. Let's start by analysing which filter is better. Allow-lists work much better than block-lists because it's much easier to define what can pass a check than defining (or enumerating) what can't pass a check. One of the reason for this is that one can always (or at least try to) find another way to pass the security check and bypass the block-list. A practical example of this is anti-viruses. Usually anti-viruses have a list of known malicious programs that shouldn't be run, and whatever program is not in that list can run. However, if a program is malicious but it's not in the database of the anti-virus (e.g., it's an exploit of a zero-day vulnerability), it's executed anyways. Moreover, the database should always be updated and the anti-virus only works on known programs. On the other side, some operating systems (e.g., MacOS and iOS), use an allow-list strategy, in fact they have a list of programs (i.e., of programs' signatures) that are trusted and a program can be executed only if its signature is in the allow-list. This mechanism is much more secure, however it puts strong boundaries on what program can be executed on the device because only trusted application can run. Another thing to notice is that

- Allow-lists are easier to implement and design, are based on the users and are independent from the context in which they are used (e.g., a phone number has the same format in every context, hence once we design a filter that let only well-formatted phone number through, we can use it anywhere).
- Block-lists are harder to design because they are based on the attackers (we want to filter what is bad) and they depend on the context in which they are used.

Based on the considerations above, we can define an order in which the different types of filters should be applied,

1. **Allow-lists**. Allow-lists should be always applied first because they are the most effective one and filter out most of the data.
2. **Block-lists**. Block-lists should be applied after allow-lists because they are less secure and can only filter out known bad data.
3. **Escaping**. Escaping can be applied after the input has been properly filtered to eliminate any type of disambiguation (it would be useless to apply escaping on strings that are then rejected by some filter).

13.2 Cross-site scripting

Cross-sits scripting (XSS) is an exploiting technique that leverages badly designed filters. To introduce this technique, let us consider a Web blog that, among other things, allows a user to add some comments that can be seen by all other users. Let us consider that the server doesn't filter the user input (i.e., the comments posted by the users). If an attacker posts the following comment

```
<script>
    alert("Malicious code here. ");
</script>
```

strange things happen. In particular, when a user loads the page, it will get an HTML page that contains the comment added by the attacker (i.e., the comment is written in plain text in the HTML because it has to be shown to the user). However, the comment is also valid HTML code that executes a JavaScript code for prompting a message to the user. This means that the browser will interpret the code as valid HTML and execute the code inside (because the `<script>` tag executes the code inside it). This means that, we are able to post a comment, containing an arbitrary JavaScript code that will be executed on every client's machine because the comment is loaded by every user when he/she accesses the blog (i.e., when he/she requests the blog's page).

13.2.1 Classification of cross-site scripting exploits

Cross-site scripting exploits can be divided in

- **Stored XSSs**.
- **Reflected XSSs**.
- **DOM-based XSSs**.

Stored XSSs

In a stored cross-site scripting exploit,

- The attacker sends some malicious code to the server.
- The malicious code is stored on the server.
- A user sends a request to the server that handles the malicious code to the client.
- The client executes the malicious code on its local machine.

Basically, the example we saw when introducing XSS is an example of stored XSS because the comment containing the code is stored on the server, and when a clients requests the page, the code (in the form of a comment) is sent to the client that executes it.

Reflected XSSs

Reflected XSSs (also called non-persistent XSSs) work without storing the malicious code on the server. To explain how do non-persistent XSSs work, let us consider a form that let the user insert some text and then asks the user, in another page, if the inserted text is correct and if it can be issued. If the input of the user isn't filtered, an attacker can insert the same string as before.

```
<script>
    alert("Malicious code here. ");
</script>
```

In fact when the confirmation page is loaded, the `<script>` tag is inserted in the confirmation page and it's interpreted as valid HTML, hence the script inside the tag is executed. Notice how the script isn't stored in the server but it works only because it's written in the confirmation page.

In this type of XSSs we also notice a big difference with stored XSSs. In this case, the victim has to send and insert the malicious code himself/herself (i.e., the victim has to exploit himself/herself) while in the latter case, the attacker would send the exploit to the server and the victim had to read it (i.e., to load the page with the malicious code). This behaviour can be achieved forcing the victim to open a link that contain the malicious code. Consider for instance the following snippet of an HTML page

```
<?php
    $var = $HTTP['variable_name']; //retrieve content from request's variable
    echo $var;                     //print variable in the response
?>
```

We can force the user to click on the following link

```
http://example.com/?variable_name=<script>alert('Malicious code.');</script>
```

to execute the malicious script. Schematically, considering the piece of PHP above, what happens is

1. We, as an attacker, send the malicious URL to our victim.
2. The victim opens the URL, making a request to the Website (`example.com` in our example).
3. The request is handled by the server that prints the variable `variable_name` on the HTML (because of `echo $var`) to send as response to the client (i.e., the victim).
4. The server sends the response (which contains the malicious script) to the client.
5. The client receives the server's response and executes the malicious script.

DOM-based XSSs

DOM-based XSSs exploits the Document Object Model (i.e., the in-memory representation of a Web page) to execute arbitrary code on a victim's machine. The DOM has some methods to obtain parts of the URL, hence if a web-site uses those methods to print an URL, we can try and pass some malicious code. Consider, for instance, the following snippet of an HTML page

```
<script>
    document.write("<b>Current URL</b> : " + document.baseURI);
</script>
```

The script tag is used to print the URI of the current web-page via `document.baseURI`. This means that if we force the victim to open the following link

```
http://example.com/#<script>alert('Malicious code. ');</script>
```

we can execute the code inside the `<script>` tag. Notice that, the `#` character is used to jump to a label in the page, hence the page `example.com` is loaded, but the whole

```
example.com/#<script>alert('Malicious code. ');</script>
```

is printed by `document.baseURI`. Schematically, what happens is

1. We, as an attacker, send the malicious URL to our victim.
2. The victim opens the URL, making a request to the Website (`example.com` in our example).
3. The request is handled by the server that builds the response page without adding the malicious code.
4. The server sends the response (which doesn't contain the malicious script) to the client.
5. The client receives the server's response and executes the DOM script. The script adds the malicious code in the HTML page and executes it. Basically, because of

   ```
   document.write("<b>Current URL</b> : " + document.baseURI);
   ```

   the URL

   ```
   example.com/#<script>alert('Malicious code. ');</script>
   ```

   is replaced in `document.baseURI` and the HTML page becomes

   ```
   <script>
       document.write("<b>Current URL</b> : " + document.baseURI);
   </script>
   <b>Current URL</b> : example.com/#
   <script> alert('Malicious code. '); </script>
   ```

Notice that, as for the non-permanent XSS, we have to force the victim into opening the link with the malicious code. Also note that in this case the server doesn't reply with the malicious code as in reflected XSS, but the malicious code is generated locally by the victim's browser when the DOM method (e.g., the `document.write()` in our example) is executed. That being said, the victim still has to do a request to the server to get the Web-page in the first place. In our example, the malicious script could also be executed even after retrieving the page. Say for instance that the victim visits `http://example.com`. The Web-page is handled to the client, no malicious code is executed and the server doesn't see any issue in the request (since it doesn't contain any weird script in the URL). Now we can force the use to click on the malicious URL. Since the only difference is that we have added a tag `#` to jump to a label in the page, the page itself isn't reloaded (i.e., the victim doesn't send a request to the server), yet the malicious code is placed in the HTML page and executed.

13.2.2 Malicious code

Now that we know that we can write arbitrary code with cross-site scripting, we have to decide what code to execute. One might think that, since JavaScript runs in a sandbox (i.e., in an environment separated from the operating system and the machine so that it can't harm them). However, an attacker can still do damages to everything related with the web browser, in particular one might try to

- Steal cookies or session variables.
- Manipulate a session
- Execute a fraudulent transaction.
- Snoop on private information.
- Drive-by download (i.e., download files without the victim's knowledge).
- Effectively bypass the same-origin policy. The same-origin policy imposes that all client-side code (e.g., JavaScript) loaded from origin A should only be able to access data from origin A.

13.2.3 Filtering strings

To mitigate XSSs we have to use filters. Let's try to apply each filter in order, from whitelists to escaping filters to see which are effective and which not.

Allow-lists

Allow-lists can be used to filter out badly formatted strings. However they are not enough to properly filter out all strings that can lead to a XSS. For instance we can design the whitelist to filter only string contains letters, numbers and punctuation symbols, however, in some context we also have to include other symbols like `+`, `-` and `<` (e.g., in a website that talks about maths). If we have to include the `<` and `>` symbols, then we don't prevent the user from writing tags.

Block-list

After having filtered out all badly formatted strings, one might want to use blacklists to filter out all tags so that an attacker can't input a tag. However, as soon as we filter out a tag (starting from `<script>`), an attacker can always find a way to write a XSS without that tags. There even exists cheat-sheets that allow to write scripts that don't contain specific tags or words.

Escaping

The only way to reliably secure an application against XSSs is to escape characters. In particular, we have to replace all characters that can be misinterpreted with equivalent characters that can't be only be interpreted as characters and not as special symbols. For instance, in an HTML document, the character `<`, which would be interpreted as a tag opener, is replaced with `&lt`, which is interpreted as a simple `<` character. Moreover, there exists well-tested tools and libraries to properly escape strings and protect applications against XSS.

Same Origin Policy

The Same Origin Policy (SOP) restricts how a document or script loaded by one origin can interact with a resource from another origin. Usually, one origin is permitted to send information to another origin, but one origin is not permitted to receive information from another origin.

Two URLs have the same origin if the protocol, port (if specified), and host are the same for both. Scripts executed from pages with an `about:blank` or `javascript:` URL inherit the origin of the document containing that URL, since these types of URLs do not contain information about an origin server.

Content Security Policy

The Content Security Policy (CSP) is an enhancement of SOP that allows an administrator to specify the sources from which one can run scripts. A CSP compatible browser will then only execute scripts loaded in source files received from those allowed domains, ignoring all other scripts (including inline scripts and event-handling HTML attributes). Notice that CSP is a specification, hence every browser has to implement it, even in different ways. Moreover, it's a manual process (it requires the admin to whitelist the allowed sources), hence it can be difficult to scale up.

13.3 SQL injections

13.3.1 Vulnerability and exploit

Many Web applications rely on some database for storing information. For instance an application that needs the user to log-in will store the username and password of its users in a database. Let us consider this example and in particular a login page that allows a user to insert its username and password. Once the user has inserted its credentials, the back-end should verify it the given user is present in the database and grant access to his/her personal page. One could write the following JavaScript code to handle this process

```
public void onLogon(Field txtUser, Field txtPassword) {
    SqlCommand cmd = new SqlCommand(String.Format(
    "SELECT * FROM Users
     WHERE username='{0}'
     AND password='{1}';",
    txtUser.Text, txtPassword.Text));
    SqlDataReader reader = cmd.ExecuteReader();

    if(reader.HasRows())
        IssueAuthenticationTicket();
    else
        RedirectToErrorPage();
}
```

The basic idea of the code above is to create an SQL query with the username and password inserted by the user and then execute it on the database. If the user inserts a proper username and password everything goes fine, however we can notice that the username (and the password) is inserted directly into the text, without escaping any characters. This means that we should be allowed to write some SQL code as username and, when it gets replaced, the we can execute it. For instance, if we know the username (e.g., `ceri`) of an user but not its password, we can type the username

```
ceri'; --
```

When the code replaces this username in the format string, we get

```
"SELECT * FROM Users
 WHERE username='ceri'; --'
 AND password='p4ssW0rd?';"
```

This code gets parsed as the query `SELECT * FROM Users WHERE username='ceri';` followed by a `--` which is how comments begin in SQL. This means that the part after the characters `--` is not executed by SQL, and we can access without knowing the password. Notice that one can also use the following string

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

to access the application without knowing any username because the second condition is always true, hence we can always get one tuple from the database and access the website.

Notice that, as for format string bugs, this vulnerability is generated by the fact that some characters (e.g., `--`) have multiple meanings.

13.3.2 Filtering

Let's try to apply filters to mitigate this vulnerability.

Allow-lists

The first filter that we should apply is allow-lists. The design of the list depends on the application, for instance if the username is a password, one can design a filter with only alphanumerical symbols, punctuation and some special characters (e.g., `@`). However, it's not always possible to filter-out SQL characters (especially dangerous ones like `--`), hence we can't fully rely on allow lists.

Block-lists

Block-lists, as for XSS, aren't very useful because, after blacklisting a character, one could always try to find another way to mount an exploit.

Escaping

The last and more efficient line of defence is escaping, in fact if we are able to disambiguate characters with multiple meaning, we can avoid (or at least reduce to a minimum) SQL injections.

13.3.3 Protection mechanisms

To make exploits harder we can

- Use **prepared statements**. Prepared statements allow to tell the DB that we want to use a query with parameters that are passed dynamically. When a parameter is passed to a query, it's not replaced in a format-string fashion, but it's used like in a function. This means that it's not possible to use `--` to comment out some SQL code because `--` is not replaced in the query, but it's used as parameter of the query.
- **Sanitise the input**.
- **Limit the privileges**. In particular, we should use different SQL users, with different privileges, so that a user with more privileges is used only for the operations that require such high privileges while the normal user can be used in all other cases.

13.3.4 Information leaks

An important issue generated by the fact that an user can interact with a database is information leakage. There are many ways (generated by bad design practices) that allow users to leak some information about the database or the source code,

- **Database mapping**. Some frameworks allow to build a Web application starting from the database schema. This means that each page is a way to represent the data on the DB without any filtering or elaboration of the data. For instance the login page is just a nice way to show the attribute `username` and `password` of the table `Users`. This can easily leak the schema of a database. Using table names as field names is also a bad practice.
- **Error messages**. Showing error messages about the error itself can leak information about how the system works and how the database is designed. One should instead tell the user what to do to recover from the error (and not the error itself).
- **Side channels**. Side channels allow an attacker to get some pieces of information about the database. Consider again the login page. If we prompt messages like `user not found` or `password mismatch` we are leaking information because we are telling to the attacker exactly what went wrong. Notice that in this case, this behaviour is a good design choice in terms of user experience but a bad one in terms of security, hence we should be able to find a trade-off between security and UX.
- **Debug traces**. Debug traces are used in the development phase to prompt a huge amount of information about what's wrong when an error occurs (even important information about the machine on which the service is running on). Enabling debug traces outside the development phase is extremely dangerous and allows attackers to get a huge amount of information, hence it shouldn't be used at all (outside development).

13.4 Cookies

HTTP is a stateless protocol, however Web application require states and sessions. To solve this issue, designers have come up with cookies, i.e., client-side pieces of information that the server can ask the client to store on its machine and send back with every request.

13.4.1 Cookie history

Initially, cookies were used for page customisation. The user could specify how the page should look like and such information were putted inside cookies. When the client requested a page, the cookie allowed the server to send the customised page to the user, thanks to the information in the cookie.

After some time, companies started using cookies for profiling users. In particular they used a unique identifier to recognise the user. The identifier was then sent to every Web page visited by the user so that it was possible to trace all the pages visited by the user.

Finally, cookies have been used for authentication. In particular, session and authentication information are stored in cookies to tell a page that a certain user is correctly authenticated.

13.4.2 Cookies for authentication and sessions

Since cookies are used also for authentication, it's important to describe how to use them properly in a delicate context like authentication. A cookie usually contain the session id (SSID) of the user. The SSID should be

- **Strong** (at least as strong as the authentication credentials).
- **Randomly generated**.

Session creation

When a user logs-in to a Web application, a new session should start. This process is initiated by the client that authenticates himself/herself. Upon authenticating the user, the server generates a random SSID, stores it and sends it back to the user as a cookie.

Session termination

Another important part of session management is when a session cookie should be considered invalid (i.e., when a session should end). This is important because the longer the session lasts, the more valuable a cookie is, because if it's stolen, an attacker can use it for a long period of time. This means that the duration of a cookie is a trade-off between security and user experience, in fact a short-duration cookie would require the user to log-in frequently (ruining the UX in favour of security).

Storing authentication credentials

Storing authentication credentials in a cookie is always a bad idea, especially if the password is stored in clear text, in fact an attacker, once stolen the cookie, can use the credential and use the session of the victim. Storing hashed and salted password in cookies can reduce the risk, however it's still not a good idea.

13.4.3 Stealing a cookie

An attacker can try and steal a cookie to obtain the credential of the victim and use his/her session. There exists many ways to steal a cookie,

- **Guessing the session id or the password**. If the SSID isn't strongly generated, an attacker can try and guess it. In this case we aren't properly stealing a cookie, but we are trying to guess its content.

- **Interception**. If the cookie isn't encrypted, it can be intercepted by an attacker that can sniff the communication channel. This type of attack is nowadays not common because most of the websites use HTTPS that ensures that cookies are encrypted.
- **Cross-site scripting**. JavaScript can access cookies, hence an attacker can use a XSS to access a cookie and send it elsewhere.
- **Cross-Site Request Forgery** (CSRF).

Cross-Site Request Forgery

The idea behind CSRF is that a user, authenticated to a website, can always issue request to that website, which are accepted and handled by the website because the user is authenticated. Say for instance that an user is authenticated to a website that offers a form as a service. An attacker can force a user to submit a custom-made form and since the user is correctly logged in the website, the form is accepted. This happens because the website isn't able to understand if the form has been issued and compiled voluntarily by the user or created by a third-party. The website can only check if the user is correctly authenticated. Consider for instance the following interaction

1. The user signs in to his/her bank.
2. The user receives a session cookie.
3. The user wants to send money to someone but instead of going to the bank website, it's forced to go on a fake website (identical to the bank's).
4. The fake website sends the user's request, with the session cookie, to the bank's server replacing the money's destination with the attacker's bank account. This allows the attacker to send money to its account.

To prevent these type of attacks we should be able to distinguish if a request came from the actual form that the user has to fill or from a malicious form. The first way to achieve this goal could be to use the `Referer` field in the HTTP header that specifies the absolute or partial address of the page that makes the request. However, this solution doesn't work, because the script can modify also the `Referer`. To actually solve this problem, we have to use CSFR tokens. When a user requests a page with a form (e.g., the page to send money to someone's account), a unique token is added to the form by the server and stored on the server side. When the form is issued, the token is sent to the server, which verifies if the token is the one generated in the first place. This technique works because the attacker can't obtain the token, because it would need the cookie of the victim (which the attacker doesn't have). Notice that, for the attack to work, the CSFR doesn't have to be stored in a cookie, because the attacker can always issue the cookie with the request when the victim is using the fake website.

Another way to mitigate this problem is to not send cookie when a request originated from a different Website than the one from which the request came from. This can be done using the field

```
SameSite=strict
```

in the request's header. However, this doesn't completely solve the problem, in fact in some cases we want to have cross-site usage and still use cookies. In this cases we can set the field `SameSite` to

```
SameSite=lax
```

to allow only cookies for cross-domain navigation only but not for POST forms, images and frames.

Chapter 14

Malware

14.1 Malware classification

A malware, also called portmanteau of malicious software, is a software intentionally written to violate a security policy. Some types of malware are

- **Viruses**. A virus is a piece of code (not a whole program) that needs another program to replicate itself and be executed. In particular, a virus attaches itself to a program that, when run, executes the virus.
- **Worms**. A worm is a malware (i.e., a complete program) that self-replicates.
- **Trojan horses**. A Trojan horse is a software that seems to do a legitimate functionality but that actually hides an harmful functionality.
- **DDoSes with botnets**. A DDoS attack is a DoS (Denial of Service) attack run by multiple devices that have been infected with a malware. Each infected (usually with a Trojan) device, called zombie, executes a DoS attack on the target machine so that, if the number of zombies is large enough, the target can't receive requests.
- **Ransomwares**. A ransomware is a malware that locks-down a system or encrypts its data and requires the user to pay a ransom to get his/her system back.

As we can notice, not all malware (like Trojan horses) self-replicate (like viruses and worms). Moreover, some software fall in a gray zone, since it depends on the definition of malware (i.e., on what harms a system). For instance, a software that spams ads can be classified as malware depending on the type of ads and rate at which they are spammed.

14.1.1 Malware evolution

Malware, in time, has been written for different reasons, in particular:

1. Initially, malware has been written as proof of concept and for fun.
2. Then, malware has shifted to mass attacks. This shift has been pushed by the possibility to obtain money from the victim. Basically, malware writing has gone from technical reason to money reasons (hence crime).
3. Now, malware is also used for political and activism reasons.

14.1.2 Malware life-cycle

During it's life-cycle, a malware

- Reproduces.
- Infects.
- Stays hidden.
- Runs its payload.

It's important to underline difference between propagation and infection. In particular,

- **Reproduction** means creating more copies of itself.
- **Infection** means to propagate on a different machine using a propagation vector (e.g., using social engineering or vulnerability exploits). A virus, also has to infect a file and not only another machine.

14.2 Types of malware

14.2.1 Viruses

Theory of viruses

After creating the first virus, its creators have stated that

> **Theorem 2** (Perfect virus detector)**.** *It's impossible to build a perfect virus detector (i.e., a propagation detector).*

To demonstrate this theorem, let us consider a perfect detector P and a virus V that calls P on itself,

- If P detects V (i.e., `P(V) == True`), then V is stopped and V is not a virus, but the detector said that V is a virus, hence we have a contradiction.
- If P doesn't detect V (i.e., `P(V) == False`), then V spreads and V is a virus. However, V spread, even if the detector said that V is not a virus and we have a contradiction.

This means that we should try to build something that recognises the virus itself and not the fact that it propagates. Notice that, such program is basically a block-list since we are blocking only some malicious code (instead of blocking all code and allow only some). This isn't ideal but, since we still want a computer to execute arbitrary code whose source is unknown, it's the only solution in many cases. Another thing to remember is that, when we have block-lists, we are running a race between the anti-malware designer (that adds the viruses to the block-list) and the malware designer that tries to build malware that doesn't fall in the block-list.

Infection

Viruses have to infect something to run their payload. In particular, a virus can infect

- A **file** (usually an executable). When infecting an executable, the virus puts itself at the entry point of the program, so that when it's called, the virus executes instead of the host application. Anti-viruses have adapted to this behaviour searching in the entry point. To overcome this adaptation, virus designers started creating viruses that put their code in the cavities of the executable file (e.g., in the zones between pages).
- The **Master Boot Record** (MBR) of a disk to execute itself when the computer boots. Using this solution, it might seem hard to propagate a virus, however, in older computers, the BIOS if the external floppy disk contained a MBR before checking the internal disk, hence if someone forgot an infected floppy in the computer, upon starting the computer, it would look into the MBR of the floppy and execute the virus.
- A **document file that supports macros**. Macros allow to execute code on a document file, hence a piece of code can infect a document and be executed every time the document is opened.

Infected documents and executables are usually passed from one computer to the other, hence they infect the computer on which the files are copied. The first two infection mechanism still work from a technical point of view, however it's hard to see them because now it's very rare that two people exchange executables using a drive. On the other hand, the third method is very dangerous because in almost every job one has to open a document file, hence it's impossible to avoid it.

14.2.2 Worms

A worm is a self-propagating malware that infects other machines using vulnerabilities exploitation and in some cases social engineering.

The beginning The first example of worm has been run on ARPANET (i.e., a small network) and it tried to exploited some vulnerabilities in the all the computers connected to the network to propagate to all the computers and infect them. Since the computers connected to ARPANET weren't that many, it was easy to try the exploits on them all.

Mass-mailers The first large-scale use of malware used mails (i.e., a social engineering technique, not an exploit) to propagate through the internet. This change in the propagation vector is due to the fact that number of possible victim was too large to try time all (in this case the attack considered all nodes connected to the Internet). These type of attacks, are called mass-mailers.

Scanning With more and more nodes connecting to the internet, all IP addresses (IPv4) got taken, hence, given a random IP, it's very likely that it used by a machine. This paves the way for a new type of attack called scanning attack that

1. Picks a random IP address.
2. Tries to infect the machine at that address using a set of exploits.
3. Infects the machine.
4. Start back from point 1.

The infected machine starts executing the same attack, infecting in turn other machines. This attack can be further improved using permutation. This means that, when a machine A infects another machine B, they split the IP addresses to explore (i.e., they don't choose the IP among all IPs but in a set of IPs), so that they won't try to attack the same machine.

The end of worms At a certain point, worms started disappearing from the Internet. The reason being that it became not worthy to run such attacks for the attackers because companies started training people to mitigate worms. However, this generated another problem in fact attackers started thinking about monetise attacks either directly (e.g., credit card abuse) or indirectly (e.g., selling information or selling access to a machine). This generated what is called Exploit-as-a-Service, in fact people, who wanted to attack a victim, started asking attackers to sell them exploits and ways to run them, effectively creating a network of services used to build and use malware.

14.2.3 Bots

A bot is a machine infected with some malware that is able to execute a set of services on the host for the attacker. A bot is usually connected to a botnet, i.e., a network of bots, connected to a Command and Control machine that tells to the bots what to do. Each bot can, among all things,

- Harvest email addresses from host.
- Log all keypresses.
- Sniff network traffic.
- Take screenshots.
- Start an http server.
- Kill a hard-coded list of process.
- Steal Windows CD keys
- Set up a proxy.
- Download a file at an URL.
- Run a shell command.
- Update itself.

It's important to notice that, a bot poses two type of threats:

- To the **infected machine** (see the list above).
- To the **entire Internet**, since all bots in a botnet can be used to mount a DDOS attack.

This could be a problem because, if a bot doesn't attack the host machine, the owner of the machine might not be interested in removing it, hence posing a threat for all the others machines on the Internet.

14.3 Defending against malware

Anti-viruses (note that anti-virus defend against all type of malwares) have different ways to defend a computer against malware. In particular, they can

- **Patch known vulnerabilities**. Patching vulnerabilities can be effective, however this method can't cope with zero-day vulnerabilities. Luckily, this type of vulnerability rarely gets exploited.
- **Use signatures** to recognise malware.
- **Look for suspicious activity**.

14.3.1 Signatures

A signature is a set of rules (or patterns) that represent some characteristics of a program. For instance a rule could be

- A known malicious instruction sequence used by families of malware.
- The byte representation of a piece of code or of a program or its length.

Usually, the process for creating signatures is the following:

1. An analyst receives a set of malicious (or thought to be malicious) programs that show the same behaviour.
2. The analyst analyses the code and tries to extract some common features or characteristics from the samples (e.g., their length, a particular sequence of instructions, data bytes at certain positions or offsets, whole-file hashes, printable strings and so on).
3. The analyst formalises the rules using a formal format (e.g. YARA).

The signature of a program is confronted against the signatures of known malwares to check if the program is malicious. Signatures can be used together with **heuristics** (that check signs of infection) and **behavioural detection** (that check if the device shows the behaviour of an infected machine). The issue with signatures is that one should always keep the list of malicious signatures (which is a black list) and that the same malware can change its form (hence the signature changes) and can propagate faster than security experts can add the signature to the anti-malware lists.

14.3.2 Detection and analysis

When securing computers from malware, it's important to differentiate malware detection and malware analysis. More precisely,

- **Malware detection** aims at checking if a program is malicious. The best way to detect a malicious program is to use block-lists.
- **Malware analysis** aims at running a program and trying to understand it's behaviour. Malware analysis can be
  - **Static**. In static analysis, analysts use reverse engineering to look into the code and understand the behaviour of the program.
  - **Dynamic**. In dynamic analysis, analysts run the program, usually using a debugger, analysis tool and a virtual machine, to analyse the behaviour of the program.

14.3.3 Allow-lists

An important thing to underline is that, an anti-malware (that uses block lists to detect malware), isn't always the best choice. In some cases, we can use directly white-lists to block all programs except for a small set of authorised ones. In such cases, using an allow-list is much more secure. Some examples are mobile operating systems which allow only signed applications to run and block every other program coming from whatever source.

14.4 Malware stealth

An important part of the life-cycle of a virus is hiding from anti-viruses. A malicious piece of code can use two techniques to hide its signature from anti-viruses when replicating:

- **Entry point obfuscation**. Entry point obfuscation is used by multicavity viruses that hijack the control after the program is launched. This technique is implemented overwriting the import table addresses or the function calling instructions.
- **Polymorphism**. Polymorphism allows a program to change its layout (shape) with each infection. Polymorphism can be achieved (but not only) encrypting (called packing) the same payload with a different key for each infection, effectively changing the signature of the malicious program and making signature detection impossible. In case of packing, an anti-virus can detect only the detection routine, which in turn gives little information about the malicious payload itself.
- **Metamorphism**. Metamorphism consists in creating different versions of the same code. A program could change its shape using different instructions, adding instructions that have no effect (like a `nop` or an `inc` followed by a `dec`), or changing the order of basic blocks in the code.

Polymorphism and metamorphism are virus-specific (or self-replicating malware specific) techniques. In the general case, a malicious program can

- **Remain dormant** for a long period of time so that it's impossible for analysts that do dynamic analysis to understand what a program does (because it apparently has no malicious behaviour).
- **Run the payload only if the Control and Command machine (like in botnets) tells them to do so** (or in general, if a certain condition verifies). This way, the attacker can decide when to show the malicious behaviour and avoid dynamic analysis of the malicious program.
- **Use anti-virtualisation techniques**. Anti-virtualisation techniques allow the malicious software to check if it's run on a virtual machine or if it's attached to a debugger and, if so, show a non-malicious behaviour.
- **Use packing**. Packing allows to compress and encrypt the malware. When the malware has to be executed, a small routine is executed and the malicious payload is unpacked (i.e., decompressed and decrypted) and loaded in memory for execution. This technique prevents static analysis since an analyst can't look into the code and can be combined with anti-virtualisation methods to slow down the analysis of a malicious program.

These techniques can be grouped under the name of **evasive behaviours** and are used to hide the behaviour of a malware from an analyst.

14.5 Rootkits

Rooting a machine, means obtaining superuser privileges on a machine. An attacker that roots a machine might want to keep its privileges and to do so it can write, on the victim machine a set of (trojanised) files:

- A backdoor (e.g., backdoored `login`, `sshd`, `passwd`).
- A set of hidden files .
- A set of hidden processes.
- A set of hidden network connections.

The tools and files listed above are called **rootkit** because they are a kit to keep the root privileged on a machine. For instance, to keep hidden files, processes and connections, a toolkit might provide a new implementation, which has to be replaced on the victim computer of some utilities (e.g., `ps`, `netstat`, `ls`, `find`, `du`, `who`, `w`, `finger`, `ifconfig`).

An admin could however have its version of the original utilities an an external drive, hence it can verify if the utilities he/she's using a the original ones or if they have been modified. To circumvent this protection, an attacker could build rootkits that modify directly the kernel, which is harder, but not impossible and can be done using syscall hijacking.

One could also try to make the kernel monolithic so that it can't import kernel modules (e.g., drivers) from external sources (which could be the source of rootkits), however even in this case it has been demonstrated that it's possible (yet very hard) to install rootkits in monolithic kernels.

14.5.1 Recognising rootkits

Rootkits can be recognised using

- **Intuition**, in fact a system that has a rootkit installed usually has a strange behaviour (e.g., a slow network even if no process is using it).
- **Post-mortem analysis**. This type of analysis allows to use a non-infected machine to analyse the hard drive of a computer (when it's off) and check if some rootkits have been installed.
- A **Trusted Computing Base** (TCB), which is a set of trusted signatures for a program.
- **Cross-layer examination**, which allows to check if, doing a task manually returns the same result as doing it with an utility (that might be compromised because part of a rootkit).

In any case, a good attacker can always find more places to hide a rootkit (e.g., in the BIOS, in the Network Interface Card or in a video card). This means that we should always remember to define at which level we start trusting the levels above, as seen when talking about trust.

Part VI

Network security

Chapter 15

Network protocol attacks

15.1 Introduction

Network protocols attacks can be divided in three categories:

- **Denial of Service** (DoS) attacks. A DoS attack allows an attacker to make a service not available to legitimate users. This category of attacks can take different names depending on the context (e.g., for radio transmission they are called jamming). Remember that, DoS attacks on remote communications are always possible, hence we can't make this vulnerability go away. However, we can try to make it harder to exploit it.
- **Sniffig**, or more in general **interception**. Interception attacks allow a third party to read messages that are travelling the network (i.e., they violate the communication's confidentiality). Sniffing is the term used for network packets. In general, for remote communications, it's almost impossible to remove this vulnerability, hence we should try and make it hard to exploit it.
- **Spoofing**. In spoofing attacks, an attacker pretend to be someone else or manipulates a packet so that it looks like coming from someone else. These type of attacks violate integrity and authenticity.

15.2 Denial of Service

Denial of Service attacks can make a service unavailable in different ways,

- An attacker can exploit a vulnerability of the service making it fail. This types of attacks are called **killer packets**.
- An attacker can use **flooding** to occupy all the resources so that a user can't access the service. In this case, the service is still working but there is no way for a user to access it.

15.2.1 Killer packets

Ping of death

To understand how killer packets work, let us analyse a simple exploit. The `ping` command uses the ICMP echo requests to ask the receiver to send back the request received. A ICMP echo request is sent into an IP packet, which maximum length is 65527 bytes. The `ping` command allows to specify the length of the ICMP packet with the option `-s` (`man ping`). One can therefore ask the `ping` command to send an echo request 65527 bytes long so that, when the IP header is added, the packet is longer that the maximum allowed. Notice that, even using IP fragmentation, the IP packet will still be too long because the packet is fragmented to fit in a Maximum Transmission Unit (which is smaller than 65527), but without checking the length of the packet itself. Once the packet reaches the destination, it is buffered, but since the protocol only allows for packets of length 65527 bytes, the buffer is 65527 bytes long. This means that a longer packet will trigger a buffer overflow that can potentially break the system. Basically, we have used a packet to stop a service.

This exploit has been originated by the fact that no one thought at what would have happened if someone sent a packet longer than expected.

Teardrop

Another example of killer packets attacks is teardrop. In this case, an attacker exploits a vulnerability in TCP reassembling. Every link in a network has a Maximum Transmission Unit (MTU) size that specifies the maximum length of a frame at the physical layer. Different links might have different MTUs, hence when a packet arrives at a router, the router might have to chunk the packet so that it can fit in the physical frame. Basically, the payload of the packet is divided in smaller sub-payload, each of which is putted in a packet with the same header of the original header. In the header we also have to specify the offset of the sub-payload from the start of the original payload (e.g., if the length of the original payload is 128 bytes and we divide it into 4 chunks of 32 bytes, the first payload has offset 0, the second has offset 32, the third 64, the fourth 96). An attacker can modify the offset field in the header so that two payloads overlap (e.g., use the offsets 0, 32, 40, 96 so that the third chunk is overlapped with the second), causing in some cases a failure in the kernel and a crash of the system.

Land attack

Land attacks happen on Windows 95 machines when in a packet, the destination IP is equal to the source IP which is equal to the sender's IP and the SYN flag is set (that requires to do a synack and then an acknowledgement). This configuration caused the kernel to hang up communication. Land attacks are originated by the fact that, to optimise the communication, when a packet has the same source and destination IP, it's not set through the whole stack back and forth but it's kept in the kernel. However, if the packet with same source and destination is sent from the outside, it would stay in the kernel forever, because no one will acknowledge it (remember that the SYN flag is set).

Postel's law

Many killer packet attacks are generated by a design principle, called Postel's law, which states that *One should be conservative in what you send, be liberal in what you accept.* This means that a system has to send out packets that are as adherent as possible to the protocol, but it should accept packets that aren't. This principle is very useful for allowing devices with different protocols to talk to each other, however it is very bad from the security point of view because it allows malformed packets to reach a destination and do some serious damage.

15.2.2 Flooding

Flooding consists in occupying all the resources of a service, for instance generating enough requests, so that a legitimate user can't access it. DoS attacks through flooding always happen, even not in malicious way. Consider for instance a booking website that is visited by a lot of users when the tickets for a big concert come out. For this reason, we can't avoid this type of attack and we can only try to mitigate it.

SYN flood attack

The TCP/IP protocol uses a three way handshake to establish the connection between two devices. The handshake works as follows

1. Device C (i.e., the client) sends a SYN packet to device S (i.e., the server).
2. Device S stores the SYN request of the client (because it has to wait an acknowledgement) and sends back a SYN acknowledgement.
3. Device C sends to the server an acknowledgment.

In this interaction, the server needs to have more resources in fact both devices have to send a message, however, the server also has to store some data. We say that the server has a **multiplier factor** because it has to use more resources. Multipliers allow an attacker to easily mount an attack because it has to use less resources. An attacker can exploit this protocol to send a lot of SYN requests (with different source addresses) without acknowledging the SYN-ack so that the server keeps in memory many SYN-requests until the memory gets saturated and it can't accept any more request.

Timers A way to solve this problem is by using timers. In particular, the server can fire a timer as soon as a request comes and when the timer fires, it deletes the request from memory (if not acknowledged before). This solution works, however we have to consider that the timer can't be too short because we have to take into account the delay of the network (i.e., the circumference of the network).

SYN cookies SYN cookies allow to eliminate the multiplier factor. When the server receives a SYN request, it encodes all the data needed to recognise the request (i.e., the data it would have saved in memory and a secret, which is the same for all packets) in a cookie and sends it back to the client with the SYN-ack. The client, has to send the cookie back with the acknowledgment so that the server can check if the ACK is legitimate (that's why the cookie contains the secret). This technique doesn't require the server to store data, hence the multiplier factor is eliminated. This means that, if a server has N resources, the attacker has to send $N + 1$ requests.

Distributed Denial Of Service

In Distributed Denial of Service (DDoS) attacks, an attacker uses multiple devices to consume the resources of the server. Even without a multiplier factor, an attacker can always use a DoS attack if it has enough resources. On a single machine it would be impossible to have enough resources to flood a server, however, if the attacker uses different machines, it's possible to saturate the resources of the server.

Botnets A typical example of DDoS attacks is a botnet. Botnets are networks of computers, called **bots**, that have been compromised with malware and that are controlled by an infrastructure called **Command and Control** (C&C). Since the attacker controls the network, it can make the bots send traffic to a target server and consume its resources. Since botnets can be made of millions of computers, it can be very easy to attack a server, even without the users of the bots knowing that something is happening.

Smurf Not all DDoS attacks require compromised networks (as for botnets) and Smurf is an example. Smurf uses ICMP messages to flood a target machine. Say the target machine has address `3.1.1.1`. The attacker can send a ICMP request to a router with address `1.1.1.1` that has source address `3.1.1.1` and destination address `1.1.1.255`. This means that the router should send in broadcast the request (because `255` indicates broadcasting), hence the request is sent to all hosts connected to the router. Since the request is an ICMP request, all hosts have to reply to the source of the request (i.e., to the source address in the ICMP request), which is the target machine with address `3.1.1.1`. This means that, the target machine is flooded with replies from all the hosts of the network controlled by the router. To partially solve this problem, we can forbid sending broadcast requests from outside the network, however this requires to reconfigure every router, which is hard. Fortunately, nowadays by default routers do not allow broadcast request from outside, however since it is a feature, we can turn it back on and accept broadcast from outside. This means that this attack could still work today.

The problem behind DoS attacks

The solution to DoS attacks is to fix the vulnerabilities on every single machine on the Internet, however an attack damages a single server but the users have to pay to solve the damage (i.e., fixing the machines). For this reason, users don't want to pay a little price for something that greatly affects others.

The Internet is full of protocols that have the same behaviour as ICMP. For each protocol, we can even identify the Bandwidth Amplification Factor, which is how much traffic the protocol is able to generate from a request.

15.3 Sniffing

15.3.1 Network-level sniffing

Network-level sniffing is based on the fact that, everything that passes through a wire or through the air can be captured by a Network Interface Card (i.e., the device that allows the computer to send and receive data). Usually, a NIC keeps only the packets that are directed to it, however we can use the NIC in **promiscuous mode** to let the NIC hold every packet that passes through the communication channel.

To mitigate this type of sniffing we can use switched networks instead of hub-based networks. This mitigates the problem because in hub-based networks, hubs simply broadcast the incoming traffic whilst, in switched networks, switches relay traffic to the correct NIC (we say that switched networks are ARP address based). This solution only mitigates the problem because one can still intercept packets in a wireless network (consider for instance Wi-Fi, every NIC near the sender can intercept the data sent through the air).

15.3.2 ARP spoofing

ARP

The Address Resolution Protocol (ARP) maps IPv4 addresses to physical MAC addresses. When a computer C has to send a message to a device whose MAC address is unknown (but the IP is known), it broadcasts an ARP request to ask where the IP address is (i.e., what is the MAC address associated to that IP address). The first reply received by C is accepted and the received MAC address is associated with the IP in the request. This means that, if an attacker is able to reply with its MAC before the actual device with that IP (call it the victim), then the attacker receives the packets directed to the victim because the frames are sent to the attacker's MAC. An attacker can easily reply before the victim, continuously sending ARP responses (without getting a request) so that when a computer issues an ARP request, the attacker's response comes first because the request isn't processed by the attacker. ARP matches are cached (so that a computer doesn't have to send an ARP request for every message), hence the spoofed address is used for some time, until the ARP request message is sent again.

This type of attack works because the entire process doesn't require authentication (e.g., a nonce in the request that has to be added to the reply). Moreover, switches don't look into the network (IP) header, hence the frames are relayed only checking the MAC address.

Mitigation To mitigate ARP spoofing, a device can

- **Check the rate at which ACK requests arrive**.
- **Put a nonce in the ARP request** and check that the nonce is contained in the reply. This levels the field because the attacker can't simply flood the requester because it has to process the request, just like the victim device.
- **Wait for different responses** to check if some other device replies to the ACK request (which means that something is wrong).
- **Solve the problem at an higher layer**, for instance using TLS.
- **Use authentication**, for instance signing the reply. This solution isn't however feasible because we should be able to assign a key with an IP address, however a device doesn't actually own an IP address.

Switching tables Switches use switching tables to associate each MAC address to an outgoing channel so that, when they have to relay a frame, they check if its MAC address is in the table and they send it to the respective channel. However, if the destination MAC address isn't in the switching table, switches broadcast the frame and wait from a response. The source MAC address of the response is then added to the switching table, associated to the channel from which the frame came from. This means that, if an attacker sends many requests with different MAC addresses, it can fill the switching table of a switch. When the table of a switch is full, it can't add new entries, hence it can only broadcast each message whose destination MAC address is not in the table. With this technique, the attacker can view all the traffic because it has filled the table with its MAC addresses and all the frames with other addresses are sent in broadcast, hence the attacker receives them, too.

Spanning trees If a switched network has loops, when a packet is sent in broadcast, it is continuously relayed through the network (because of the loop). This phenomenon is called **broadcast storm** and is undesirable. To solve this problem, switched networks build spanning trees, so that every packet is either sent to the leafs or towards the root. The nodes of the network use BPDU packets to generate the spanning tree, however BPDU packets aren't authenticated, hence an attacker can manipulate them to change the shape of the tree for sniffing or ARP spoofing purposes.

15.3.3 Security design choices

ARP spoofing and in general network protocol attacks are usually generated by the fact that a protocol doesn't require authentication. It might seem that the designers of a protocol didn't consider security, however this is far from true. Security has been considered, however the designers figured out that it was more important to preserve the functionality of the protocol instead of adding security mechanisms that make the protocol more complex and less robust. The idea is that, such protocols are usually the very foundation of the internet, hence if they don't work, the whole Internet doesn't work (which isn't secure either), therefore it's better to keep the protocol simple and robust and add security mechanism in other (usually upper) layers. That being said, it's important to document and advertise this design choice (i.e., not securing the protocol) so that those who design the protocols of the layer above know that they have to deal with security.

15.3.4 IP spoofing

IP/UDP spoofing

In IP/UDP, the IP source address of a packet is not authenticated, hence an attacker A can change the IP address of a packet he/she sends. In particular the attacker can replace in a packet its IP address with the address of a victim so that it looks like the packet comes from the victim and not from the attacker. This type of attacks can be used to obtain a Denial of Service. Consider for instance a set of servers, each of which offers a service using UDP (e.g., upon receiving a request some data about the server is sent back). An attacker can send a packet to each server with the source IP of the victim (an not the attacker's) so that the replies are all sent to the victim. If the response is much bigger than the request (i.e., we have an high multiplication factor) the victim could consume all its resources for receiving the responses sent by all the servers.

Since the response is still sent to the spoofed device (i.e., the device whose request has been modified), if attacker and victim are on different networks, IP/UDP spoofing is blind because the attacker doesn't see the response. This isn't a big problem if the attacker and the victim are on the same switched network because the former can still intercept the communication.

IP/TCP spoofing

TCP three-ways handshake In TCP, spoofing a victim machine is harder because the three-ways handshake use sequence numbers. Before describing how an attacker can spoof a machine, let us recall TCP's three-ways handshake. Say a machine A wants to initiate a connection with a machine B,

1. Machine A starts by sending a `SYN` packet to B that contains the source and destination IP and a random sequence number (`SEQ`) x.

2. Machine B, upon accepting the connection request, replies to A with a `SYN-ACK` packet that contains A's nonce incremented by 1 $x + 1$ and a new random sequence number y.

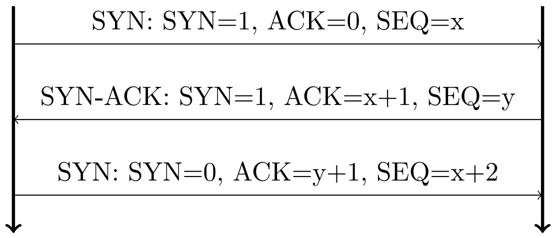


Figure 15.1: TCP's three-way handshake.

3. Machine A acknowledges B's `SYN-ACK` with an `ACK` packet containing $x+1$ and $y+1$. After this step, the connection phase is over and A and B can communicate freely.

IP/TCP spoofing attack Now that we know how the protocol works, let us understand how an attacker A can impersonate a machine M (whose IP address is known by A, i.e., M's IP is spoofed by A) and initiate a TCP connection with a machine B. A can send a `SYN` packet to B with M's spoofed address and a sequence number chosen by A. B's `SYN-ACK` is sent to M (not to A) because the `SYN` packet contains M's IP. This means that the attacker has to do a blind attack because it can't see B's response. This means that A doesn't know the random sequence number y in B's `SYN-ACK`, hence A can't successfully complete the handshake and impersonate M. However, if the nonce y isn't chosen correctly (i.e., it isn't completely random) by B, the attacker can try and guess it and eventually complete the handshake.

Reset messages A way to mitigate this attacks is to use reset (`RST`) packets. More precisely, when the spoofed machine M receives a `SYN-ACK` packet, without having sent any `SYN` packet, it sends a `RST` packet to B informing that M won't communicate with B anymore. However, an attacker with enough resources can DDOS M and prevent it to send the `RST` message. If the attacker can DDOS M, the attacker only has to guess the sequence number y (because A has chosen the first sequence number x), hence the more random it is, the harder it is to guess it.

15.3.5 TCP session hijacking

Say two machines M and S have a TCP connection. When M leaves the network (either voluntarily or because of an attacker), an attacker A can try and keep the connection with S alive so that it can impersonate M. If A can sniff the conversation between M and S, before M leaving, it can record the sequence numbers of each packet and continue the conversation with S without S noticing it because the sequence numbers are correct. An example of TCP hijacking is shown in Figure 15.2.

15.3.6 Man In The Middle

Man In The Middle (MITM) is a category of attacks in which a machine A can impersonate, for each party in a communication, the other party. Say two machines M and S are communicating. A MITM attack by A consists of

- Tricking M into thinking that it is speaking to S, while it is actually speaking to A.
- Tricking S into thinking that it's speaking to M, while it's actually communicating with A.

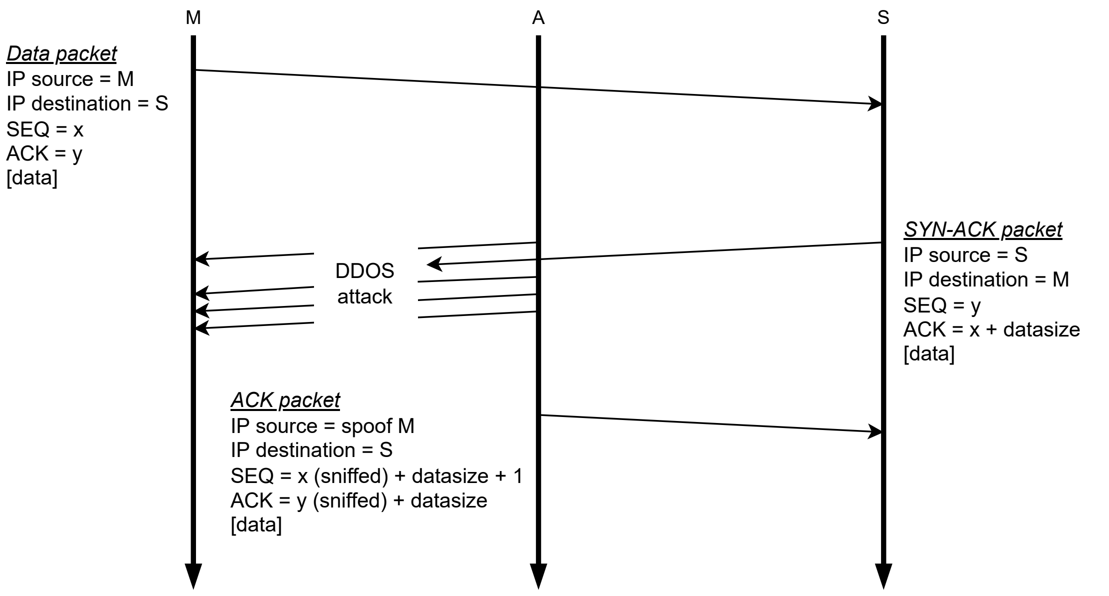


Figure 15.2: TCP hijacking.

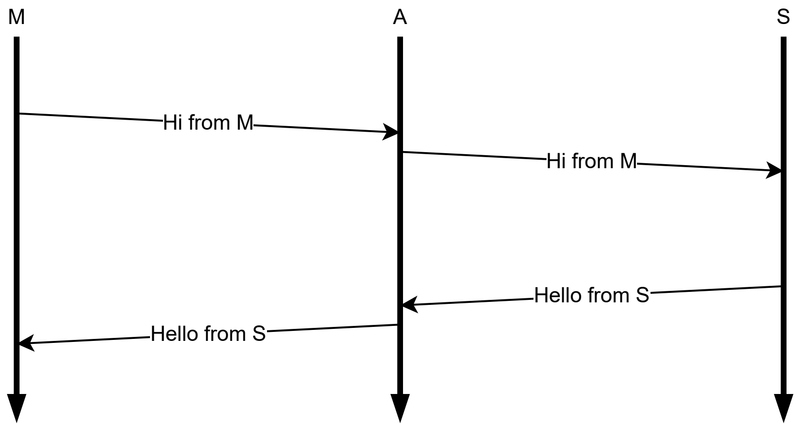


Figure 15.3: A MITM attack. The attacker A lets M think that it is S and since it properly replies to M's requests, M believes it. The same happens for S.

A logical representation of a MITM attack is shown in Figure 15.3.

Internet is full of MITM situations, for instance an Access Point is a MITM between us and the Internet or a proxy is a MITM between our brower and the server we are trying to access, hence it might be hard to detect it. MITM can be divided in

- **Physical** if M and S are physically connected to A.
- **Logical** if the connection between M and S is only logical.

moreover, we can divide MITM in

- **Full-duplex**. If A redirects the communication in both directions.
- **Half duplex**. If A redirects the communication only in one direction (e.g., controls the channel from M to S but not from S to M).

15.3.7 DNS poisoning

The DNS protocol for domain name resolution is not authenticated, hence an attacker can spoof the communication between a client and a DNS server (or between DNS servers). In particular, given the fact that DNS servers cache the address associations, an attacker can inject in the cache an entry that associates a domain name `website.com` to the IP address of the attacker's server so that when someone searches `website.com` it's redirected to the attacker's server instead of the proper server with `website.com`. To mount a DNS cache poisoning attack,

1. An attacker makes a recursive query for website `bank.com` to the victim DNS server.
2. The victim DNS server, which doesn't know `bank.com`, queries the authoritative DNS server.
3. The attacker, impersonating the authoritative DNS server (since no authentication is required), sniffs or guesses the the DNS query ID and spoofs the answer. Basically, the attacker sends a reply to the victim's DNS server that contains the correct query ID (spoofed) and the IP of the attacker's server (instead of the server where `bank.com` actually is). If the attacker's reply comes before the authoritative server one, the victim cache contains the IP address injected by the attacker.
4. From now on, if someone uses the victim DNS server (whose cache contains the injected address), it would be served the IP of the attacker's server. The attacker can now build a website identical to `bank.com`, served on the server with the injected IP, to do all sort of things.

15.3.8 DHCP poisoning

The DHCP protocol allows (among all things) to dynamically assign IP addresses in a network. The part of the protocol used to assign an IP address to a device doesn't require authentication, hence it's vulnerable to spoofing attacks. In particular, after sending a DHCP request, a device waits for a requests that contains (among other things)

- The IP address that the device should use.
- The DNS server that the device should use.
- The default gateway that the device should use.

If an attacker is able to reply to the request before the actual DHCP server, it can, for instance, set the default gateway as its address so that all victim's traffic is directed to the attacker.

15.3.9 ICMP redirect attack

ICMP is used to send debugging information and error reports between hosts, routers and other network devices at IP level. In particular, an ICMP `Redirect` packet tells an host that a better route exists for a given destination, and gives the gateway for that route. When a router detects a host using a non-optimal route it

1. Sends an ICMP Redirect message to the host and forwards the message.
2. The host is expected to then update its routing table.

The attacker can forge a spoofed ICMP `redirect` packet to re-route traffic on specific routes or to a specific host that may be not a router at all. This attack can be used to

- Hijack traffic (the attacker can elect his/her computer as the gateway).
- Perform a Denial-of-Service attack.

Differently from previously analysed protocols, ICMP uses a weak authentication mechanism (an ICMP message includes the IP header and a portion of the payload of the original IP datagram), but it can be bypassed.

15.3.10 Route mangling

In route mangling attacks, an attacker can signal route changes to modify the routing tables of routers to redirect traffic and perform specific attacks like IGRP, RIP, OSPF, EIGRP, BGP. The first three examples do not require authentication while the other two use weak authentication mechanisms that are sometimes even turned off (because router-to-router connections are usually based on trusted relationships).

Chapter 16

Secure network architectures

16.1 Firewalls

A firewall is a filter that filters the messages that pass from two networks and allows to separate two networks so that, when a network is attacked, the effects doesn't spread on the neighbour networks. More precisely a firewall is an applier of rules, each of which defines what can or can't pass through the filter. Rules are implementations of high level policies (e.g., *client C can't get images from a server S*) and are a good examples of allow and block lists. The better the rules, the more effective the firewall is. Luckily, since rules are examples of black and white lists, we know how to design good rules. In particular we should always start from the allow-list (i.e., deny all accesses and allow only a limited set of hosts) and then design block-lists if needed.

It's important to understand that, a firewall is designed to check what goes through it but it can't do nothing about the messages that are exchanged inside a network. In other words, a filter can only filter inter-network messages and not intra-network messages.

Another thing to note is that, if we can find a way around a firewall, the traffic along that route isn't checked. One might think that it's hard to work a firewall around, however it happens many times. For instance, we can connect to a server using Wi-Fi or the Cellular network, effectively bypassing the firewall installed on the Wi-Fi access point in the second case.

An important thing to understand is that a firewall is a computer, hence it can be compromised. However, this rarely happens and usually an attacker finds a way to bypass the (rules of the) firewall instead of attacking the machine itself.

Firewalls offer few or no services (they are simple filters), hence they reduce the attack surface that an attacker can exploit.

Taxonomy Firewalls can be classified based on their inspection capabilities that usually correspond to the protocol layer that they can inspect. Firewalls are divided in

- **Packet filters** (network layer). Packet filters are usually embedded in routers.
- **Stateful packet filters** (network and transport layers). Stateful packet filters are what we usually call firewalls.
- **Circuit level firewalls** (transport and application layers). Circuit level firewalls are legacy firewalls that are not used anymore.
- **Application proxies** (application layer). Application proxies are what we usually call proxies.

16.1.1 Packet filters

A packet filter, being a network-layer firewall, can only analyse a packet's header, hence it can check

- The **source IP and port**.
- The **destination IP and port**.

These information is basically what a router needs, that's why routers usually use this type of firewalls. For instance, Cisco's packet filters are called Access Control Lists (ACLs) and are embedded in the routers.

Packet filters are stateless, meaning that each message analysed is independent from the previous ones. For this reason, packet filters can't track TCP connections because they would need to inspect a packet's payload (but they can only check the header). This means that, to handle a TCP interaction between two devices, we should also write response rules. To better explain this concept let us consider an example. Say we have a firewall that divides the outside network $N1$ with a network $N2$ where there is a server. In this context, if we want to give access to a web server at `10.0.0.4:80` in network $N2$, we should deny all incoming and outgoing connections and allow only packets from the outside that are directed to `10.0.0.4:80`. However, the server should reply to the client on network $N1$, so we have to add a rule that allows packets from address `10.0.0.4:80` of $N2$ to whatever address of network $N1$. This rule can be compromised because the server's address can be spoofed (remember that a firewall only filters what passes through it) by an attacker in network $N2$, since the firewall can't check the packets payload. For this reason, packet filters are just a weak filter for packets.

16.1.2 Stateful packet filters

Stateful packet filters solve the problem of packet filters. In particular, being a stateful firewall, a stateful packet filter can look into the payload of an IP packet and keep track of the TCP state machine. This means that, stateful packet filters can recognise SYS-ACKs and avoid response rules (which can be dangerous). This is cool, however it also has some drawbacks. In particular, when dimensioning a network, the power capacity of a normal router is based on the number of packets per second that it should handle, which is roughly based on the bandwidth of the network. On the other hand, if we have to add a firewall to the router, we also have to consider the number of connections (because it has to store them) which depends on the number of machines connected to the network. This means that, a firewall has more capacity requirements (bandwidth and number of machines) than a router (only bandwidth).

Deep Packet Inspection

Stateful packet filters, can look into the IP packet's payload. This capability, called Deep Packet Inspection DPI, can be used to

- Inspect the content of a communication, for instance to do censorship inspection.
- Apply Network Address Translation (see Appendix B.1).

Network Address Translation NAT has to modify the header in the packet to change the private address with the AP's public address. This means that NAT can be combined with firewalls rules. In particular, when a device want to initiate a TCP connection, the firewall

1. Receives a SYN packet from `192.168.1.4` to `188.12.14.42`.
2. Checks if there is a rule that allows the packet to go through. Say such rule exists.
3. Adds the connection to the list of connections that it's keeping track of. Form now on, every proper answer to this connection will go through.
4. Creates a NAT slot to map `192.168.1.4` to one of the Access Point's public addresses, say `44.125.234.2`.
5. Changes the private source address `192.168.1.4` with `44.125.234.2`. It could also change the sequence number if it thinks that it's not secure enough (i.e., it hasn't been randomly generated).

When the device on the other side of the connection replies, the firewall

1. Receives a SYN-ACK packet directed to its public address `44.125.234.2`.
2. Checks if the destination address `44.125.234.2` is in one of the NAT slots.
3. Checks if the connection is in the list of open connections (added at step 3 when opening the connection).
   - If the connection is in the list, the destination address is replaced with the private address `192.168.1.4` and the packet is forwarded to it.
   - If the connection isn't in the list (or the sequence number is wrong), the packet is dropped because it's a SYN-ACK to SYN which has never been sent.

Basically, the firewall is a MITM because it can change things in the packet's header. It could also change the content of the payload itself. The same idea of tracking connections can't be applied to UDP, because it is connection-less and has no concept of session. To handle UDP, firewalls have to infer the behaviour of two devices that are communicating and can recognise some communication behaviours (e.g., if a device sends a DNS request, a DNS reply should come some time after). The mechanism for handling UDP is the same as for TCP but in this case the firewall is recognising behaviour instead of connections. Moreover, the firewall uses timers so that it can close a session if an answer doesn't arrive before the timer expires.

Application layer inspection Deep Packet Inspection is sometimes necessarily because some application protocols use information of a layer on another layer, effectively violating the protocol stack. For instance, FTP inserts the ports used for file transferring in the application layer (i.e., we are using transport layer information in the application layer payload). For this reason, the firewall should be able to look into the application payload.

Exploit filtering Deep Packet Inspection also allows firewalls to stop some known exploits. If the firewall can look into the packet, we can write some rules that recognise an exploit and discard the packet if it contains a malicious payload. In this case we are creating a block-list, however, it's fine, since we are putting it on top of an allow-list. Firewalls that are able to stop exploits are called **Intrusion Prevention Systems (ISPs)** are very similar to anti-viruses. ISPs work only if a certain exploit is known, hence they can't handle zero-day vulnerabilities, which means that in case of zero-day vulnerability we only have the allow-list to stop the attack.

16.1.3 Circuit firewalls

Circuit firewalls are Man In The Middle firewalls that work at the transport and application layers. In particular,

1. A device A wants to send a message to a device B.
2. A connects to the circuit firewall and sends the message to the firewall, asking to connect and send the message to B.
3. The circuit firewall connects to B and sends the message received by A to B.

This type of firewall is not transparent, since the application, i.e., the device A has to know that it has to connect to the firewall and not to B directly. On the other hand, stateful packet filters are transparent to the application, hence are much easier to use and have practically replaced circuit filters.

16.1.4 Application proxies

Application proxies work like circuit firewalls but at the application layer. More precisely, an application connects to a proxy that connects to the actual destination. A proxy can,

- **Filter some request methods** and allow only some (e.g., allow GET requests but not POST requests).
- **Protect the client**.
- **Protect the server** from malicious clients. In this case, proxies are called **reverse proxies** or **WebApp firewalls**. Reverse proxies are used to protect servers that can't be patched.
- **Speed up the connection**. Proxies can cache replies, hence when an application issues a request to a proxy, the proxy can issue directly the cached reply without connecting to the server.

16.2 Network partitioning

When designing a network, we should always try to apply the castle model that says that **only machines inside a network can access data inside the network**. However, in some cases we have to violate the castle model because,

- A company might want to offer some service to the outside.
- A company might want to allow someone outside the network to access the network, like it were inside it.

16.2.1 Giving access to services

If a company wants to offer some services to the outside, it should allow only some connections to come in. To achieve such goal, the company should split the network in different zones, separated by firewalls, so that if a zone is compromised, it's harder that even the other zones get compromised. Each zone represents a privilege level and innermost zones with the zones close to the Internet having the fewer privileges. This architecture is called **multi-zone architecture**.

Dual-zone architecture

An example of multi-zone architecture is the dual zone architecture in which the network is divided in

- A **demilitarised zone (DMZ)** that contains the machines that have to be accessible from the outside (e.g., the Web server). No critical or irreplaceable data (or machine) should be placed in this zone since this zone is almost as risky as the Internet. In other words, the DMZ is a semi-public zone that hosts servers that can be accessed from the Internet.
- A **private zone** that contains all the machines not in the DMZ zone.

Logically, this architecture is build with two firewalls, one that divides the outside and the demilitarised zone and one that splits the demilitarised and private zones (Figure 16.1). This means that a device on the Internet (outside) can only talk to a machine in the DMZ zone, while a machine in the demilitarised zone can talk to the outside or to the private zone. The internal firewall (i.e., the one that divides the DMZ from the innermost network) blocks external machines from talking to the innermost machines. In practice, the architecture is implemented with one firewall with three ports, one for each zone.

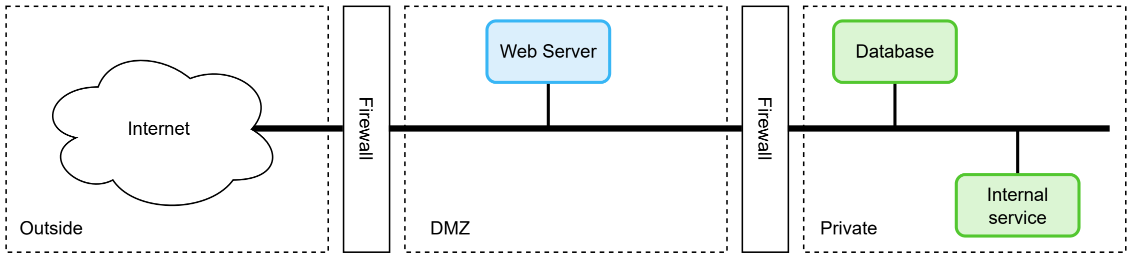


Figure 16.1: A dual-zone architecture.

The dual-zone architecture limits the spreading of an attack in the demilitarised zone, however it's not impenetrable because a device in the DMZ zone can still talk to a machine in the private zone.

Since the demilitarised zone is very important to secure a network, let us remind its characteristics:

- The DMZ **must contain only non-critical and non-irreplaceable data or machines**.
- The DMZ **is a semi-public zone**.
- The DMZ **hosts only services that can be accessed from the Internet**.
- **A device in the DMZ can talk to other device on the Internet or in in other zones**.
- **The firewall the divides the DMZ from the inner network should block any message coming from the Internet**.

Multi zone architecture

The multi-zone architecture is a generalisation of the dual-zone one. Usually, the architecture is the same as the dual-zone case, however the internal network is divided in multiple zones. Note that, the firewall can't check the packets sent inside a zone, however it can check the packets that are sent from one zone inside the inner network to another zone in the inner network.

16.2.2 Virtual Private Networks

Virtual Private Networks (VPNs) are used to

- Connect a device outside a corporate (but not only) network to the network, like if it were inside.
- Connect different branches (i.e., different networks) of a company.

In particular, VPNs create an encrypted and authenticated tunnel that logically connects the user outside the corporate network to the (VPN server of the) innermost firewall (i.e., the one that divides the demilitarised and private zones). In other words, a VPN is an encrypted overlay connection over a public network. Using a multi-zone architecture wouldn't be as secure as using VPNs.

VPNs can operate in two different modes

- **Split-tunneling**. In split-tunneling, the traffic directed to the corporate network, is routed through the tunnel, while the traffic directed elsewhere is sent directly to its destination without going through the tunnel. This mode is more efficient with respect to full-tunneling, however the VPN server has less control over the traffic. Moreover, split-tunneling creates side-channels.
- **Full-tunneling**. In full-tunneling mode, all the traffic coming from the user goes through the tunnel, even if it's not directed to the company's network. This can generate a lot of unnecessary traffic, however, on the plus side, we define a single point of control and application of all security policies.

An example of a VPN is shown in Figure 16.2.

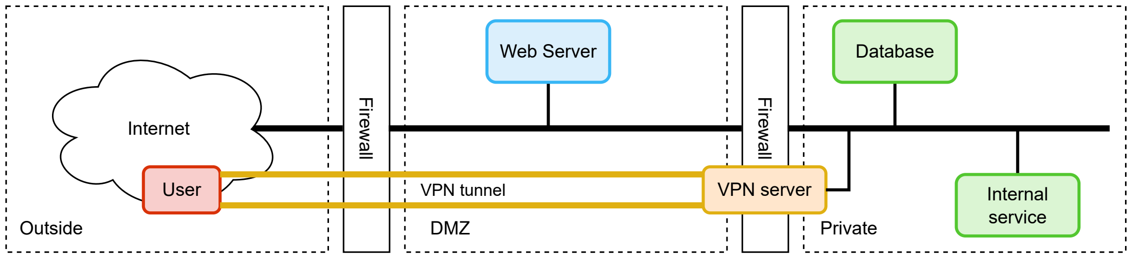


Figure 16.2: A VPN.

Technologies

VPNs can be implemented using

- **Point-to-Point Tunnelling Protocol (PPTP)**. PPTP is a variation of the PPP protocol that adds authentication and confidentiality (through cryptography).
- **SSH tunnel**.
- **VPN over TLS**.
- **OpenVPN**. OpenVPN works above the application layer (i.e., it incapsulate traffic on top of the application layer). OpenVPN is mainly used for client-to-network traffic. OpenVPN encapsulates IP, over TLS, over IP.

- **IPSEC**. IPSEC is an security extension of IPv6 and IPv4 that provides authentication and cryptography directly at the IP layer (i.e., directly manipulating the IP header). IPSEC is a very complex technology which is very hard to use, so it's used mainly on site-to-site (network-to-network) connections between firewalls. IPSEC encapsulates IP inside PPTP over IP.

Chapter 17

TLS and SET

The Internet protocol stack doesn't support encryption by default. This can be a problem, especially because the Internet is used for monetary transactions and to exchange private information that should be seen only be the recipient. In particular, a remote transaction (i.e., a transaction with a remote server), requires

- **Authentication**. A client should be able to very if the server to which he/she's talking to is legitimate, namely if the client is sending its private information (e.g., the credit card security number) to the actual payment server.
- **Confidentiality**. Only sender and receiver should be able to read the messages of a transaction.
- **Integrity**. Every message exchanged in a transaction shouldn't be modified by anyone else but the sender.

To achieve this properties in a transaction, we have to design protocols that work on top of some Internet protocol stack layer.

Transparence and critical mass problem When designing a protocol, we have to take into consideration the critical mass problem (also transparence problem). This problem comes from the fact that a service is used by a client if other clients use it. However, when a service is initially deployed, no client adopts it, hence no other client will start using it, since no client is currently using it. This is clearly a vicious circle that can be solved, if possible, only with big investments to involve users (e.g., discounts or bonuses). An example of this problem is a dating app. A client starts using the app only if many other use it (because it want to find the perfect date), however at the beginning no one use the app, hence a client is less likely to start using it because he/she can't find many dates.

17.1 SSL and TLS

Secure Sockets Layer (SSL) is a protocol designed by Netscape to encrypt a communication over a network. SSL has been usually used with HTTP (HTTPS) to add encryption to HTTP, however SSL is independent from HTTP.

SSL has been revised and standardised by the IEFT to build Transport Layer Security (TLS), hence SSL and TLS are basically the same protocol. The most updated and secure version of TLS, i.e., the one that should be used, is version 1.3.

17.1.1 Guarantees

TLS guarantees that:

- **Communication is encrypted and authenticated**, hence confidentiality and integrity are enforced.
- **The server is authenticated**, hence the client is sure to talk with the actual server it want to talk to.
- **The client can optionally be authenticated**, hence the server is sure to be talking to the client it wants to talk to. This option is rarely used because authentication requires a certificate, which is in general hard to do for a client, in fact a person should install the certificate on each device he/she wants to use.

To guarantee this properties, TLS uses

- **Asymmetric encryption** (i.e., public and private key encryption).
- **Certificates** to authenticate servers and optionally clients.
- **Symmetric encryption** to encrypt the communication, since every message is encrypted and asymmetric encryption would be computationally too expensive for this job.

17.1.2 Handshake

When a client wants to connect to a server using TLS, it has to perform a TLS handshake. The handshake works as follows,

1. The client sends to the server a message containing the **cipher suite** and some **random data**. The cipher suite contains a list of protocols that the client knows and that it can use to encrypt the communication. In particular, it should specify four categories of protocols

   - Key exchange algorithms.
   - Bulk encryption algorithms (for encrypting large volumes of data).
   - Message authentication code algorithms.
   - Pseudo-random functions.

   Allowing client and server to agree on the algorithms to use is very useful since, encryption functions and algorithms many be broken or improved, hence it's useful to be able to change them in time. Notice that, only the algorithms change, while the structure of TLS remains the same.

2. The server, which has a predefined list of algorithms it knows for each category above, chooses an algorithm for each category and replies to the client message with

   - The chosen algorithms (one for each category).

   - Some random data.
   - Its (the server's) certificate. Notice that the certificate can be exchanged only at this moment only because now both client and server know which algorithm to use. Notice that, if the server supports different public key algorithms, it should have a certificate for each algorithm.

3. The client verifies the server's certificate thanks to a Certificate Authority.
4. The client sends a pre-master secret (i.e., a random number) to the serve, encrypted with the server public key, so that only the server can read it.
5. Optionally, the client can also sign the previous message and send it, together with a certificate, to the server so that it can verify that the client is actually who it claims to be.
6. Finally, client and server can use the pre-master key and the random data sent in the first two messages to compute the shared key used for symmetric encryption, i.e., to encrypt the messages exchanged from now on. The random data is used so that the shared key generated is always different and an attacker can't reuse previous messages. Basically, the random numbers exchanged at the beginning make it harder to run replay attacks.

17.1.3 Communication encryption

After the handshake, every message sent between two devices is encrypted with symmetric encryption. If we consider HTTPS, TLS encrypts basically everything and in particular

- The **URL** of the request or the response (initially not the domain name but from v1.3, the domain is also encrypted).
- The **URL of the requested documents**.
- The **contents of the request or response** (i.e., the body of the request or response).
- **Forms' contents**.
- **Cookies**.
- The **http header**.
- The **GET request parameters**.
- The **method used for the request** (e.g., POST, GET).

Basically an attacker can see that we're connecting to a specific IP address and port.

17.1.4 Man In The Middle attacks

On the internet we always have MITM scenarios, hence we should ask ourselves if TLS is safe against MITM attacks.

Say a MITM can intercept messages and change them. Let's check what can happen when the server sends its certificate to the client:

- The attacker could try to change the public key in the certificate and replace it with his/hers. However, the certificate is signed by the CA's hence the client recognises that the certificate has been modified and the attacks fails.

- The attacker could generate a malicious copy of the server's certificate with its (the attacker's public key), signed by an untrustworthy CA (e.g., one controlled by the attacker). Even in this case the attack would fail because the client doesn't trust the CA, hence can't verify the certificate.
- The attacker could generate a certificate by a legitimate and trusted CA for a website `bad.com`, different from the one (say `requested.com`) requested by the client. The attacker can then try and replace the certificate of `requested.com` with the one it generated (i.e., `bad.com`) and send it to the client. Even in this case the client refuses the certificate because, albeit being issued by a trusted CA, it doesn't refer to the website it requested (i.e., `requested.com`).

The cases above represent all possible ways to run a MITM attack and, since TLS can stop them all, we can say that it's secure against this MITM attacks.

17.1.5 Advantages and disadvantages

TLS is widely used on the Internet (nowadays, basically everywhere) because it

- Protects transmission.
- Ensure authentication.

effectively defusing interception and MITM attacks. However, TLS also has some drawbacks,

- It offers **no protection before and after the TLS interaction**. This means that usually data is stolen on the client side (i.e., before the interaction) or on the server (i.e., after the interaction).
- It **relies on the Public Key Infrastructure** for certificates, hence we still have to trust the Certificate Authorities that issue the certificates.
- It's **not foolproof**, especially considering the fact that initially the user interface that notified an error in the protocol showed unclear information and some user, who couldn't understand what the issue was, connected to a website with the wrong certificate.

To overcome this limitations, designers have developed

- **HTTP Strict Transport Security** (HSTS), which is an HTTP header in the HTTP header that tells the browser to use only HTTPS when connecting to a website.
- **HTTP Public Key Pinning** (HPKP), which is an HTTP header that tells the browser to refuse certificates from different CAs for that origin.
- The **Certificate Transparency** CT standard (which is usually adopted by all the CAs) that, among other things, forces the CA to submit the metadata of every issued certificate to a independent, replicated log so that a site owner can check or be notified of certificates issued for the properties they manage. Basically, a website owner can check if some CA is issuing a certificate for that website, without asking the owner. Browsers can refuse certificates not logged in the CA logs.

17.2 SET

The Secure Electronic Transaction (SET) protocol is an encryption protocol designed to protect transactions. Say a client C wants to buy some goods from a merchant M using the payment service of bank B. Ideally,

- The merchant shouldn't get the payment information.
- The bank should only get the payment data and manage the payment, without knowing what goods have been bought.

To obtain this result, SET uses a dual signature. In a dual signature, two messages m_1 and m_2 are hashed to obtain digests d_1 and d_2.

$$d_1 = h(m_1)$$
$$d_2 = h(m_2)$$

The digests are then hashed to obtain the dual signature ds.

$$ds = h(d_1 + d_2)$$

Note that, given m_1 and d_2 we can compute the dual signature, without knowing message m_2 (and vice-versa with d_1 and m_2). This protocol can be used to protect a transaction as follows

1. When a client initiates a transaction, the order information (i.e., what goods have been bought) and the payment data are used as m_1 and m_2 to compute the dual digest. The dual digest is finally signed with the client's private key to obtain the dual signature.

$$ds = sign_{client}(hash(hash(m_{order})) + hash(m_{payment}))$$

2. The order information m_{order}, the hash of the payment information $hash(m_{payment})$ and the dual signature ds are sent to the merchant.
3. The payment information $m_{payment}$, the hash of the order information $hash(m_{order})$ and the dual signature ds are sent to the bank.
4. The merchant tells to the bank that he should receive a payment and sends to B the hash of the payment $hash(m_{payment})$. Since the bank has the full information $m_{payment}$ about the payment, it can verify the hash sent by the merchant. The bank can finally verify the double signature since it has $m_{payment}$ and $hash(m_{order})$. Upon verifying both conditions, the bank can process the payment.
5. The merchant verifies the double signature since it has m_{order} and $hash(m_{payment})$. If the signature is correct it can process the order.

Notice that the merchant and the bank don't know the data used by the other part since they only receive the hash of the information that they shouldn't know.

17.2.1 Failure of SET

The SET protocol can effectively and securely protect a transaction, however it failed to catch on because the client needs a certificate (since merchant and bank have to verify the dual signature, signed with the client's private key). In particular, the client would require a registration to get the certificate. Moreover the client should install the certificate on each device used (i.e., also on mobile phones), which might be hard, especially for those who are not used to technology. This is a good example of critical mass problem, even if this protocol has been designed and used by Visa and MasterCard which account for the majority of the transactions in the world.

Appendix A

x86-64

A.1 Instruction Set Architecture

A.1.1 Syntax

x86-64 instructions can be represented with different two different syntaxes:

- The **Intel** syntax.
- The **AT&T** syntax.

Note that the operands in the two syntaxes are in opposite order. We will consider the Intel syntax because it's less verbose, however, most UNIX tools use the AT&T syntax.

Intel syntax In the Intel syntax

- A register is written with its name alone (e.g., `eax`).
- A number is written as a normal number with a suffix representing the base used to represent it (e.g., `4ah` is number `4a` in hexadecimal).
- A reference is written as `[base + scale * index + offset]` (e.g., `[eax + 4*ebx]`), where
  - `base` is the base register and must be a register,
  - `index` is a register whose value is added to the base register,
  - `scale` is a value and it's multiplied by the value of `index` (the default value is 1),
  - `offset` is a value and it's added to the value of `base`,

For instance,

```
mov eax, [ebx + 42h]
```

AT&T syntax In the AT&T syntax

- A register is written with its name preceded by a `%` (e.g., `%eax`).
- A number is written as a normal number with the `$` prefix and in the `0bN` format where `b` represents the base and `N` is the number to represent (e.g., `$0x4a` is number `4a` in hexadecimal).
- A reference is written as `segment:offset(%base, %index, scale)` (e.g., `(%eax, %ebx, 4)`), where
  - `base` is the base register and must be a register,
  - `index` is a register whose value is added to the base register,
  - `scale` is a value and it's multiplied by the value of `index` (the default value is 1),
  - `offset` is a value and it's added to the value of `base`,
  - `segment` is a segment register used only for the x86 architecture (we will forget about it).

  The memory reference is computed as `%base + (scale * %index) + offset`. Note that constant values should not be preceded by `$`.

For instance,

```
mov 0x42(%ebx), %eax
```

A.1.2 Data types

The main data types used in the x86-64 ISA are

- `byte`: 8 bits, represented with the character `b`.
- `word`: 2 bytes (16 bits), represented with the character `w`.
- `dword`: 4 bytes (32 bits), represented with the character `l`.
- `qword`: 8 bytes (64 bits), represented with the character `q`.

A.1.3 Storing data

The x86-64 ISA supports different instructions to store data from and to memory. In particular, we can use the instructions `mov` and `lea`.

Move instruction

The

```
mov dest, src
```

instruction is used to move the value `src` in the `dest`. For instance

- `mov eax, 42h` moves the number `42h` in register `eax`.
- `mov eax, ebx` moves the value in the register `ebx` in the register `eax`.

- `mov eax, [ebx]` moves the value in memory pointed by register `ebx` in the register `eax`. This instruction can sometimes be seen as `mov eax, DWORD PTR [ebx]` to indicate that the value in `ebx` is a pointer.

Note that a pointer can also be used as the destination of a move instruction. For instance the instruction `mov DWORD PTR [ebx], eax` stores the value of `eax` in the memory location pointed by `ebx`. Also remember that destination and source can't be both memory locations (i.e., no `mov [eax], [ebx]`).

Load effective instruction

The

```
lea destination, source
```

instruction is used to move the address of `source`, which is always a memory location, in the `destination`, which is always a register. Note that the `lea` instruction doesn't access memory. For instance, let us consider two registers `eax = 0x00000000` and `ebx = 0x00403A40`. The instruction `lea eax, [ebx + 0xc]` takes the value in `ebx` (which is a pointer to memory) and adds 12 to it. The resulting value is then saved in `eax`. This means that the new value of `eax` is `0x00403A4c`. Note that, the `mov` would have stored in `eax` the value at address `0x00403A4c` and not the address itself.

A.1.4 Endianness

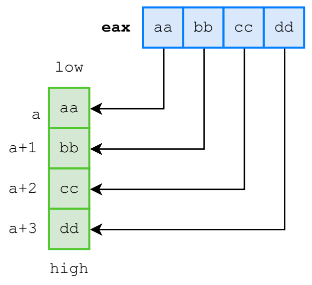
The endianness specifies in which order the bytes of value are saved in memory.

Big endian

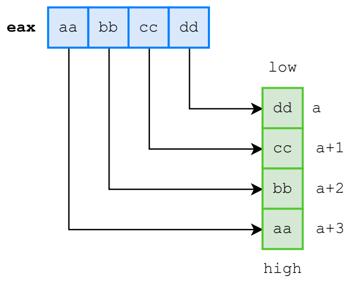
In systems that use the big endian convention, the most significant byte of the word is stored in the smallest address given.

Little endian

In systems that use the little endian convention, the least significant byte is stored in the smallest address.



(a) Big endian convention.



(b) Little endian convention.

Figure A.1: Endianness. Memory is represented in green (vertically), and registers are represented in blue (horizontally).

A.2 Stack

The stack is a portion of memory located just below (i.e., at lower addresses) the kernel space. The stack grows dynamically and in particular it,

- **grows towards lower addresses**, and
- is **written from low addresses to high addresses**.

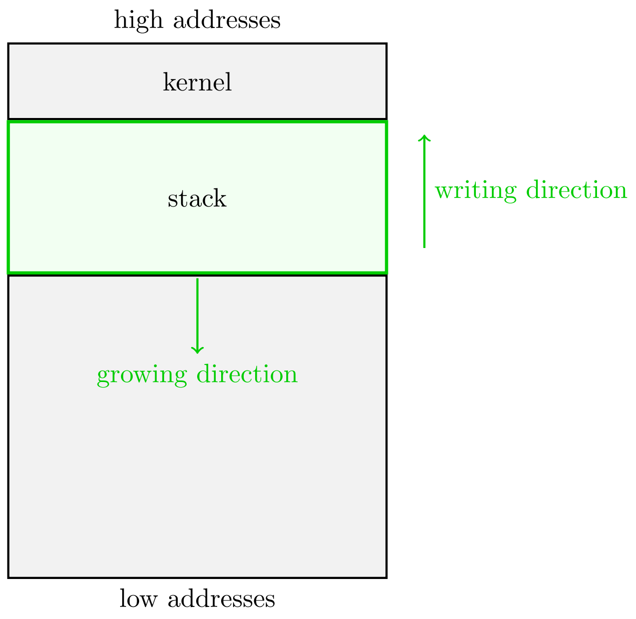


Figure A.2: A representation of the stack.

The head of the stack (i.e., the address of the head of the stack) is stored in the `esp` register. This means that we push a value contained in a register by:

1. decreasing the value of `esp` by 4 (x86) or 8 (x64) (i.e., `add esp, -4`), and
2. moving the value from the register to the address specified by the `esp` (i.e., `mov [esp], eax`).

This means that we pop a value by:

1. moving the value from the address in `esp` to a register (i.e., `mov eax, [esp]`).
2. increasing the value of `esp` by 4 (x86) or 8 (x64) (i.e., `add esp, 4`).

A.3 Functions

A.3.1 Calling conventions

A calling convention defines

- How parameters are passed (stack, registers or both), and who is responsible to clean them up.
- How to return values.
- Which registers should be saved by the caller and which by the callee.

cdecl

The `cdecl` (C declaration) convention imposes that

- Arguments (parameters) are passed through the stack in right-to-left order.
- The caller removes the parameters from the stack after the called function completes.
- The return value is saved in the register `rax`.
- The caller should save only the registers `eax`, `ebx` and `ecx`. The other registers are saved by the callee.

We will use this convention.

stdcall

The `stdcall` convention imposes that

- Arguments are passed through the stack in right-to-left order.
- The callee is responsible for clearing the function parameters from the stack before returning.

fastcall

The `fastcall` convention imposes that

- Up to 2 parameters are passed via registers `ecx` and `edx`. Other parameters are pushed to the stack in right-to-left order.
- The stack is cleaned by the callee.

System V

The `System V` convention (used for Linux x86-64) imposes that

- Parameters are passed in registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` and the subsequent ones on the stack in reverse order (right to left).
- The caller has to clean the stack after the function returns.

- The callee should save registers: `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, and `r15`.
- The caller should saved registers (scratch): `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10`, `r11`.
- The return value is saved in `rax` (or in `rax` and `rdx` if it's a 128-bit value).

A.3.2 Function calling

When a function is called, an activation record has to be put on the stack. The activation record contains all the information about the function (e.g. arguments and local variables). The stack is handled using two registers,

- The register `esp` (i.e. the **stack pointer**) contains the address of the top of the stack (i.e. the address of the topmost valid cell of the stack).
- The register `ebp` (i.e. the **base pointer**) contains the address of the first word of the activation record of the currently executing function (i.e., the record on top of the stack). Below the base pointer, we find all the activation records of the other functions of the process.

When a process calls a function it has to

1. Push the values of the function's arguments to the stack in reverse order (i.e., starting from the last parameter).
2. Call the `call` instruction. The call instruction `call 0x8042DE8E`
   - Saves the instruction pointer `eip` on the stack (i.e., it pushes the `eip` on the stack).

     ```
     push eip
     ```

   - Stores in the instruction pointer the address of the first instruction of the function (i.e., the address passed to the `call` instruction) and jumps there.

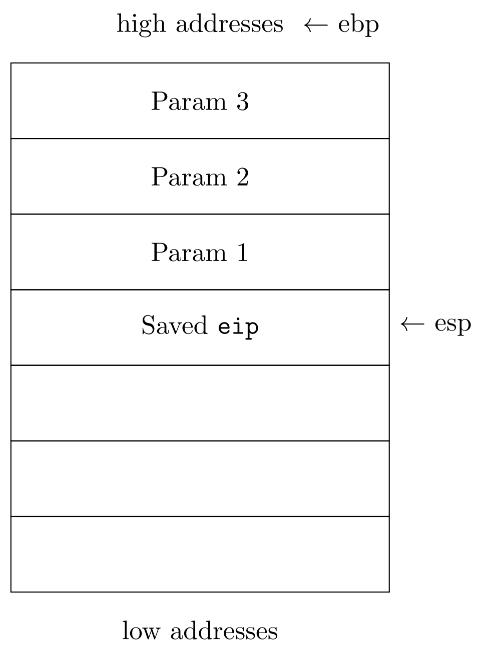
     ```
     jmp 0x8042DE8E
     ```

After the `call` instruction, the stack pointer points to the `eip` stored on the stack by the `call` instruction.

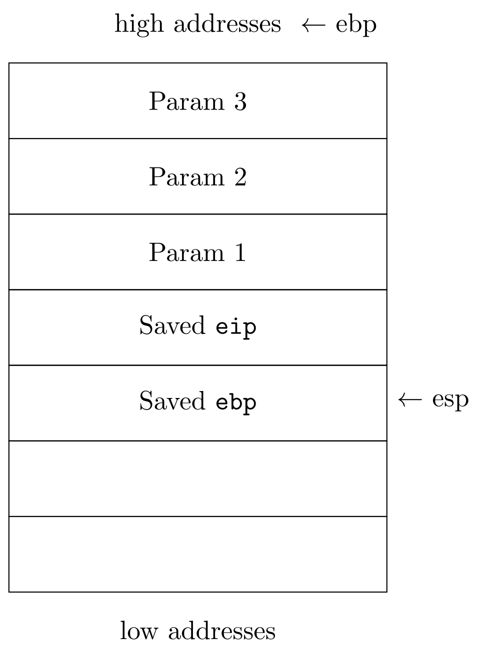
A.3.3 Function prologue

After calling the function, the process has to add some other information on the stack. In particular, the process

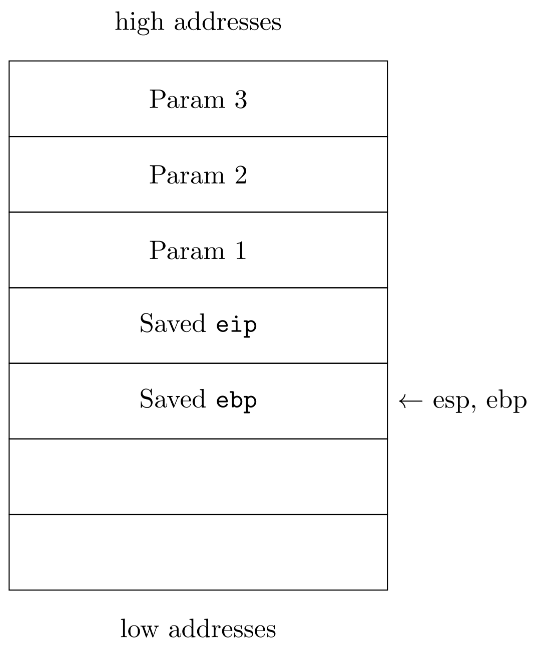
1. Pushes the base pointer `ebp` to the stack. This operation is done because the `ebp` should point to the base of the current activation record, hence we have to change its value. Since we move the base pointer, we have to save the previous value of the base pointer so that, when the function terminates its execution, we can restore the old value of the `ebp`.
2. Moves the base pointer `ebp` to the stack pointer `esp`. Basically, we are saying that the activation record starts where the stored `eip` is.
3. Allocates space for the local variables declared inside the function moving the stack pointer `esp` towards lower addresses.



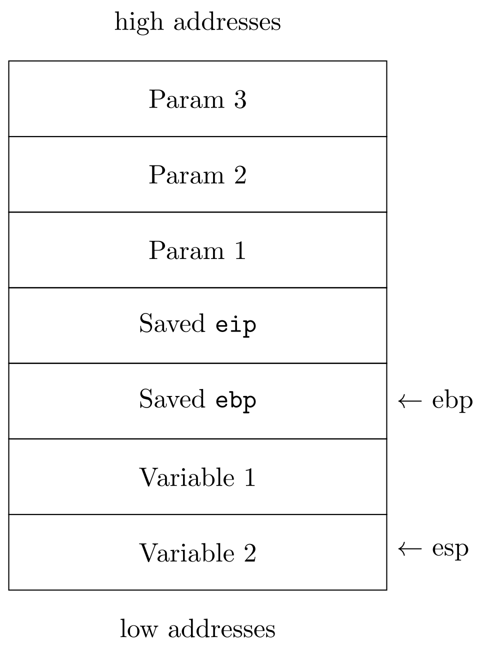
(a) Initial configuration of the stack after the call instruction.



(b) Configuration of the stack after having saved the base pointer.



(c) Configuration of the stack after having moved the base pointer.



(d) Configuration of the stack after having allocated space for the variables.

Figure A.3: A function prologue.

A.3.4 Function epilogue

Before returning execution to the caller of a function, the processor executes the function's epilogue. In this phase the processor

1. Moves the base pointer `ebp` in the stack pointer `esp`. Basically, we are restoring the old `esp` (what was the base of the stack is now the top of the stack).

   ```
   mov esp, ebp
   ```

2. Pops the base pointer saved on the stack in the `ebp`. Basically, we are restoring the old `ebp`. After the pop, the `%esp` is moved 4 bytes up, hence where the stored `eip` is.

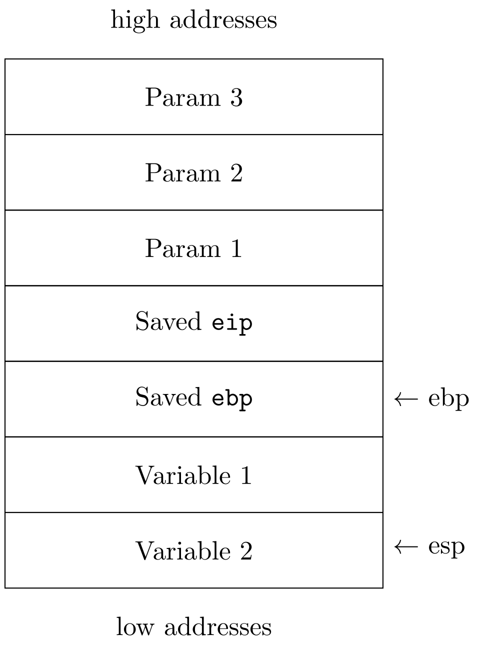
   ```
   pop ebp
   ```

3. Calls the `ret` instruction that

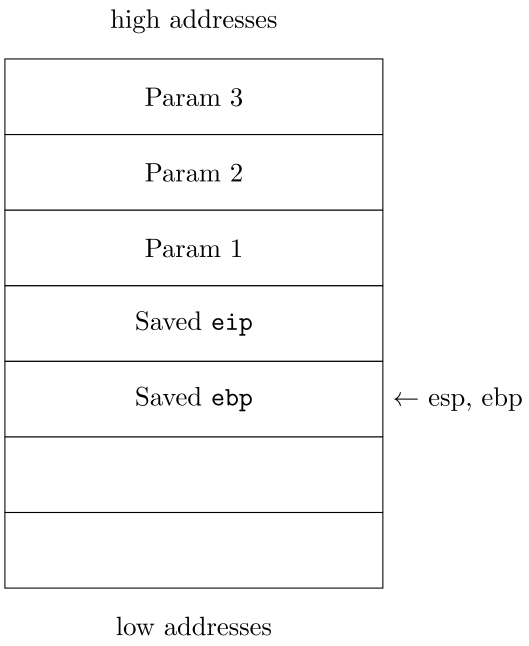
   (a) Pops an address from the stack.

   (b) Jumps to that address and starts executing from there.

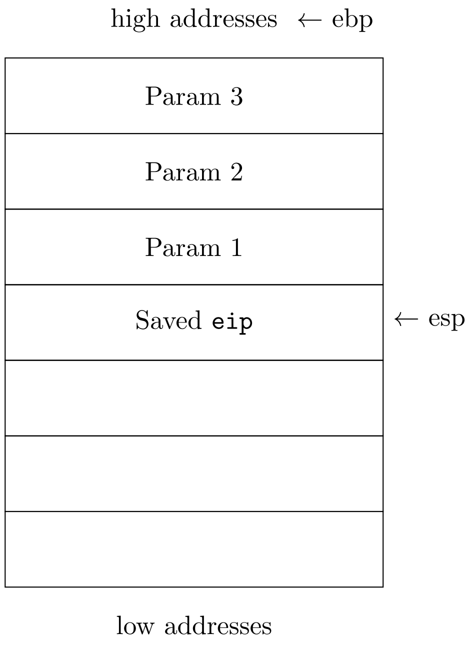
   Since the `esp` is on the stored `eip`, the `ret` instruction starts executing the next instruction of the caller (because the caller saved its next instruction on the stack when calling the function).



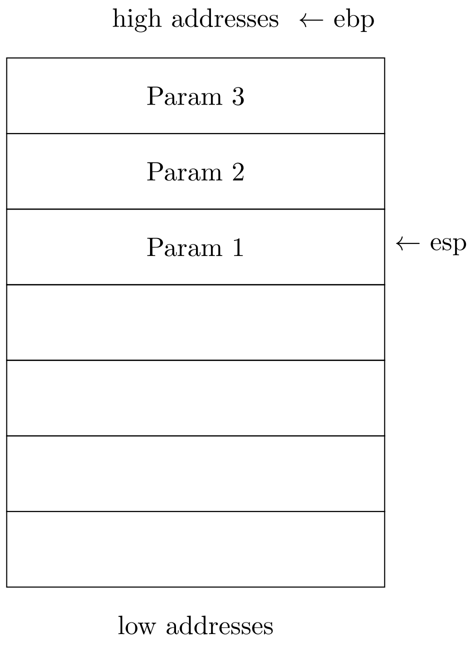
(a) The configuration of the stack before the function epilogue.



(b) The configuration of the stack after having cleared up the variables.



(c) The configuration of the stack after having popped the base pointer.



(d) The configuration of the stack after having called the ret instruction.

Figure A.4: A function epilogue.

A.4 Structure stacking

Structures are allocated on the stack in the opposite order. This means that the last field of the structure is allocated at higher addresses and the first field at lower addresses. In other words, fields are stacked in reverse order (i.e., the last field is stacked first). Every single field is however stacked from low to high addresses.

```c
typedef struct data {
    char name[8],
    char code[16],
    int id
} data_t;

int main() {
    data_t data;
}
```

Listing A.1: A code that uses a data structure.

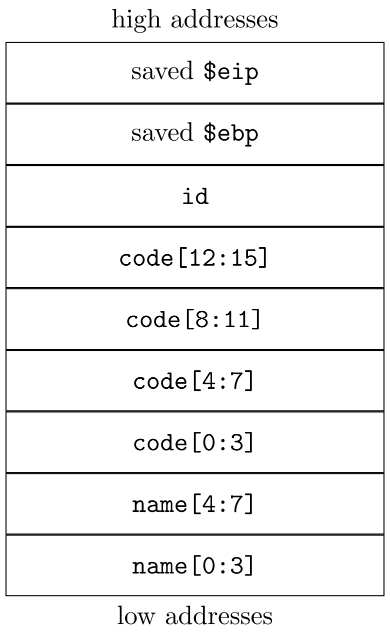


Figure A.5: A representation of the stack for code in Listing A.1.

A.5 Virtual Memory

Virtual memory allows mapping the memory addresses used by the CPU to the memory addresses of the RAM. Basically, a component, usually called Memory Management Unit (MMU), is put in between the CPU and the RAM and handles the mapping of the memory addresses used by the CPU to the address in physical memory. This technology is particularly useful if we remember that multiple processes (i.e., programs) can run on the same CPU. If we can map the addresses used by the CPU to the physical addresses, then we can allow each process to think like it were the only one using on the CPU. Basically, each process can use the whole memory addressing space (i.e., it doesn't have to care about other processing using the same memory) in its code and the MMU, together with the OS, translates such addresses to physical memory addresses. The translation is implemented using a table that maps the virtual addresses of each process to the physical address in RAM.

Virtual memory also allows us to share some common addresses. Consider for instance libraries. Since multiple processes can use the same library, we can allocate the library once in memory and then map the virtual address of the library for each process to the same physical address. For instance, if process P_1 puts libraries at virtual address `0x1000` and process P_2 puts libraries at virtual address `0x2000`, we can map both addresses to the physical address `0x4200`.

Currently, almost all implementations of virtual memory divide the virtual address space into 4 kilobytes pages (i.e., blocks of contiguous virtual memory addresses each with a size of 4 kilobytes) and the mapping is done between pages and not single addresses. The mapping is handled by page tables that map virtual pages to physical pages. Each entry in the page table has a flag indicating whether the corresponding page is in physical memory or not. If it is in real memory, the page table entry will contain the real memory address at which the page is stored. If it is not in real memory, the hardware raises a page fault exception.

Appendix B

Networking

B.1 Network Address Translation

Network Address Translation is a mechanism that allows a device to translate a private address into a public address. More precisely, IP addresses are divided in,

- Public addresses, which are unique (i.e., two devices can't have the same public address).
- Private addresses, which are used inside a network to address only the device inside that network. The most common classes of private addresses are `10.0.0.0/8` and `192.168.0.0/16`.

Every network can internally use private addresses, however, when they want to communicate to the Internet, the private addresses have to be translated to public addresses, otherwise many devices would use the same address, which makes routing impossible. This means that an Access Point, which divides a private network from the internet has to translate the private address used by a device inside the network into one of its (the AP's) public addresses.

Say a device, with a private address `192.168.1.3`, inside a network wants to send a request to the Internet. The source address in the packet's header has to be replaced with one of the public addresses (e.g., `53.1.9.123`) of the Access Point that connects the device to the Internet. The same would be done when the reply from the Internet reaches the AP, in fact, the AP has to replace its public address (`53.1.9.123`), contained in the reply, with the private address of the device inside the network (`192.168.1.3`).

B.2 Local Area Network

A Local Area Network (**LAN**) is a computer network that interconnects devices within a limited area. Computers on a LAN connect to the same network switch, either directly or through wireless access points (APs) connected to the same switch. The two most used technologies used in a LAN are Ethernet (wired) and Wi-Fi (wireless). Each LAN defines a broadcast domain, i.e., a set of addresses to which a broadcast message is sent.

B.2.1 Virtual Local Area Network

Virtual Area Networks can be logically simulated using Virtual Local Area Networks (VLANs). More precisely, a VLAN is a set of devices that define a logical broadcast domain. In other words, a VLAN defines a set of devices that are logically connected one to the other like if they were in a physical LAN. Notice that the devices in a VLAN don't necessarily have to be in the same physical LAN, in fact a VLAN is a logical connection (not a physical one). This is a big advantage because we can move a device from one VLAN to another VLAN without physically moving it. VLANs are used to split a network depending on the functionalities, communication and security needs of the devices in the network.

B.3 Network devices

In a network, some devices are used to route (or relay) traffic. These devices can be classified depending on their capabilities and on the network layers they can look into. In particular,

- **Hubs** work at the physical layer, hence they can only relay the traffic coming from each port to the other ports.
- **Switches** can look into the data-link layer, hence they can only relay packets depending on their destination MAC address.
- **Routers** can look into the network layer, hence checking the IP address of the receiver. This means that a packet can be routed depending on the IP address of the receiver.

Other important components of a network are gateways. Gateways allow different networks to communicate using protocols of different layers (i.e., they are focused on one layer only).

B.4 Protocols

B.4.1 Address Resolution Protocol

The Address Resolution Protocol (ARP) is a protocol used by devices in a LAN to associate an IP address to a MAC address. When a device connects to a LAN, it's assigned an IP address (either statically or using DHCP). When a device D wants to send a message to a certain IP address, it has to send a message in broadcast (i.e., to MAC address `ff:ff:ff:ff:ff:ff:ff`) containing the requested IP address. The message, being sent in broadcast, is received by every device of the LAN and the device T that recognises its address in the request replies (in unicast) to the D. When D receives T's response, it knows that the unknown IP is associated with T's MAC address. Note that, ARP messages are communicated within the boundaries of a single network, and never routed across nodes of different networks (inter-networking nodes).

B.4.2 Simple Mail Transfer Protocol

Simple Mail Transfer Protocol (SMTP) is a client/server protocol used to send and receive emails (between mail servers) using TCP on port 25. SMTP defines two types of agents:

- User Agents (UAs). UAs are the programs used by users to write emails.
- Message Transfer Agents (MTAs). MTAs are the servers that receive the mails from UAs and send them to the MTA of the recipient.

Say user A, which is using mail-server S_A wants to send an email to user B using mail-server S_B. The protocol works as follows:

1. User A writes an email to B
2. A's UA opens a TCP connection on port 25 with S_A and sends the message to S_A using the SMTP protocol. In this interaction, A is the client and S_A is the server.
3. S_A's MTA opens a TCP connection on port 25 with S_B and sends the message to S_B using the SMTP protocol. In this interaction, S_A is the client and S_B is the server.
4. S_B stores the message received from S_A.
5. B uses a mail access protocol (e.g., POP3, IMAP or Webmail but not SMTP) to retrieve the message stored on S_B. In this interaction, B is the client while S_B is the server.

B.4.3 Post Office Protocol 3

The Post Office Protocol 3 (POP3) protocol allows pulling e-mail messages from a mail server using TCP on port 110 of the mail server. The main limitations of POP3 are that the user can't create folder hierarchies on the mail server and messages are deleted from the server once pulled (hence they can be stored only locally).

B.4.4 Internet Message Access Protocol

The Internet Message Access Protocol (IMAP) protocol allows pulling email messages from a mail server using TCP on port 143 of the mail server. Differently from POP3, IMAP let the user define a folder hierarchy on the server that can also store messages (i.e., messages aren't deleted after being pulled).

B.4.5 Webmails

Webmails allow accessing mails in a mail server using plain HTTP (or HTTPS).

Definitions

Theorems and principles