Computer Security

github.com/martinopiaggi/polimi-notes

2022-2023

Contents

1 Introduction

It's a lot social engineering:

> "People are the weakest element in a security chain. While technology vulnerabilities are patched, there is no patch for human stupidity."

The attacker always takes the **path with least resistance**.

1.1 CIA paradigm

- **Confidentiality**: information can be accessed only by authorized entities, unauthorized people can not have access to it (can be seen as privacy).
- **Integrity**: consistency and trustworthiness of information over its entire lifecycle
- **Availability**: information must be available to all the authorized parties without external constraints

Other definitions:

- **Vulnerability**: elements that let someone break the rules of the CIA paradigm.
- **Exploit**: method of using one or several vulnerabilities to achieve a certain goal that breaches certain constraints.
- **Asset**: Recognizes the value that someone or an organization places on a particular entity.
- **Threat**: This is a possible violation of the CIA.
- **Threat Agent**: The person or thing that may instigate an attack.
- **Attacker**: The person or thing that executes the attack.
- **Hacker**: An individual with an intricate knowledge of computers and computer networks, and a desire to learn everything.
- **Security Level**: something which addresses the threats directed towards the asset.
- **Protection Level**: security measures put in place to safeguard an asset.
- **Risk**: $Risk = Asset * Vulnerabilities * Threats$

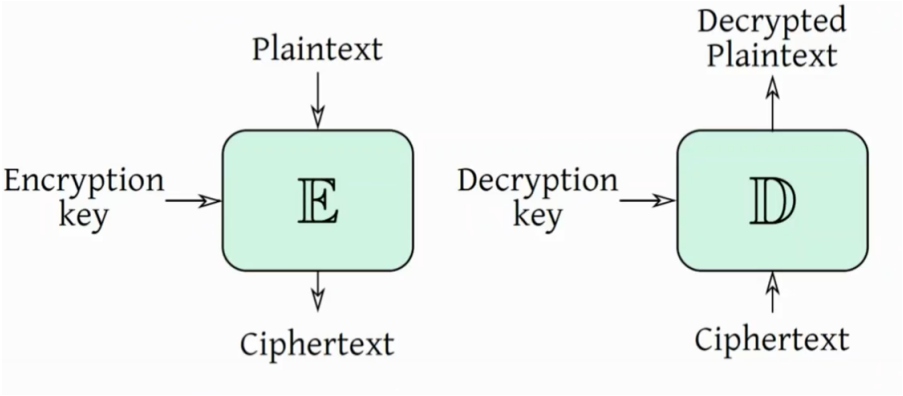
> "A system with limited vulnerabilities but with a high threat level may be less secure than a system with many vulnerabilities but with low threat level."

2 Foundations of Cryptography

We will cover the basics of cryptography that are relevant to discussions about system security. Definitions:

- **Plaintext** space P : set of possible messages $ptx \in P$ (Old words, now $\{0, 1\}$)
- **Ciphertext** space C: set of possible ciphertext $ctx \in C$

- **Key** space K: set of possible keys



A perfect cipher is a cipher where the ciphertext reveals no details regarding the plaintext . For a symmetric cipher, designated as $< P, K, C, E, D >$ with $|P| = |K| = |C|$, perfect security is guaranteed if:

1. Each key occurs with the same probability which means that all keys possess an equal likelihood of being employed.
2. For any given combination of plaintext and ciphertext within the sets P and C, there exists a unique key in set K that will map the plaintext to the ciphertext.

The **One Time Pad** (OTP) exemplifies a minimalist perfect cipher. It is achieved by passing a random key of identical size as the message through an XOR operation. However, this key must not be reused. Hence, despite confirmation of the existence of a perfect cipher, practical usage is unattainable. An easy way to understand the concept of perfect cipher:

```
M = UGO (21 7 15)
K=+3
C = XJR (24 10 18)
```

In the case of the non-perfect cipher example, the attacker would need to bruteforce all possible keys by shifting each letter and comparing it to a dictionary of known words or phrases to find the correct key.

"Toy" Perfect Cipher Example

```
M = UGO (21 7 15)
K=+5+2+2
C = ZIQ (22 9 18)
```

In the case of a perfect cipher, the attacker would obtain all possible combinations of the letters through bruteforcing, which are equally probable to be the original text. Brute-force is the upper bound to the robustness of an algorithm, it is the worst case scenario to apply (most time-consuming method) in order to break an algorithm. In real life algorithms the same key is re-used more than once: this is why algorithms are not perfect, furthermore the key is the only unknown thing (the algorithm must be known). At this point, the attacker in order to break the algorithm has to find the key, and all real world algorithms are vulnerable to brute-force attacks.

2.1 Some fundamentals of Information Theory

It's possible to mathematically quantify the difficulty of guessing a changed secret string between two entities using a random variable X. The amount of information which can be transmitted depends on the distribution of X.

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

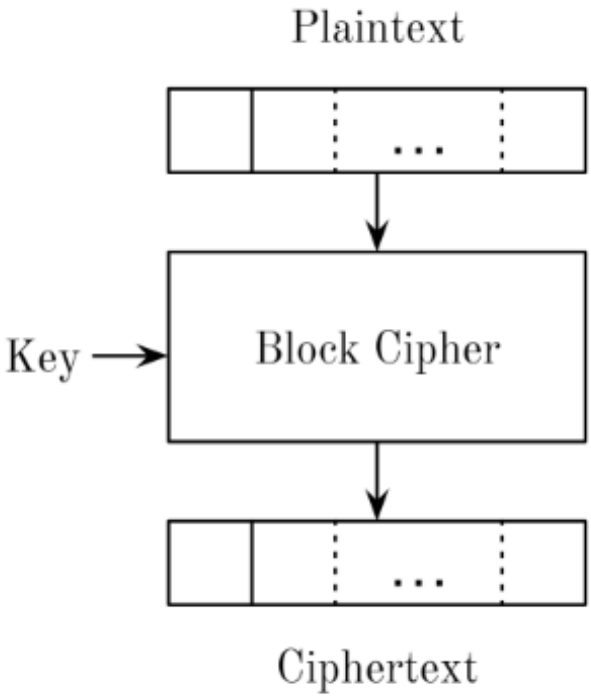
The entropy $H(X)$ is a good way to express the hardness to guess a string.
Compression of bitstrings without loss is impossible: cryptographic hashes discard some information.

2.2 Computationally secure

Our road from perfect ciphers to pseudo-random generators:

0. **Random Functions**: functions that take an input and produce an output that is not easily predictable. The output of the function appears to be random, even if the input is known.
1. **Pseudo Random Functions** (PRF): A function that appears to be random, but is actually deterministic and produces the same output for the same input. This is just a RF which can be used in practice.
2. **Pseudo Random Number Generator** (PRNG): A deterministic function that generates a sequence of random-looking numbers from a seed or key $\{0,1\}^\lambda \to \{0,1\}^{\lambda+n}$ where n is called stretch of the PRNG. Is called "pseudorandom" since it's truly random and are deterministic, meaning that given the same seed, the same sequence of numbers will be produced every time.
3. **Cryptographically Secure Pseudo Random Number Generator** (CSPRNG): Cryptographically Safe Pseudorandom Number Generator (CSPRNG) is a PRNG whose output cannot be distinguished from an uniform random sampling of $0, 1^{\lambda+n}$ in $O(poly(\lambda))$ time. This function stretch the key you use. It's not a perfect tool but it's a practical tool.

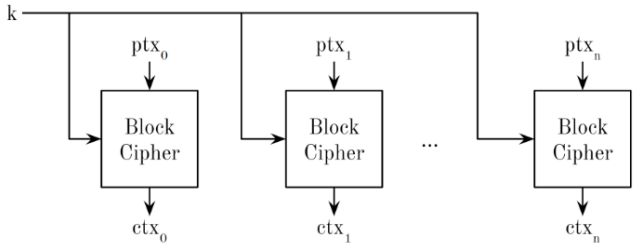
4. Pseudo Random Permutations (PRP) (also called **Block cipher**): a function which maintains input/output length (permutation) where it's not possible to tell apart in $O(poly(\lambda))$ from a Random Function RF. Operatively speaking acts on a block of bits outputs another one of the same size which looks unrelated (but actually it is.. since it's **pseudo** random).



A block cipher is said to be broken if, without knowing the key, the plaintext can be derived from the ciphertext with less than 2^λ operations. Which means that it's broken if exists a way to broke it that is not a bruteforce technique. The standard algorithm is now Advanced Encryption Standard (AES). In AES, the block has a

length of 128 bits, and the key can be 128, 192 or 256 bits long.

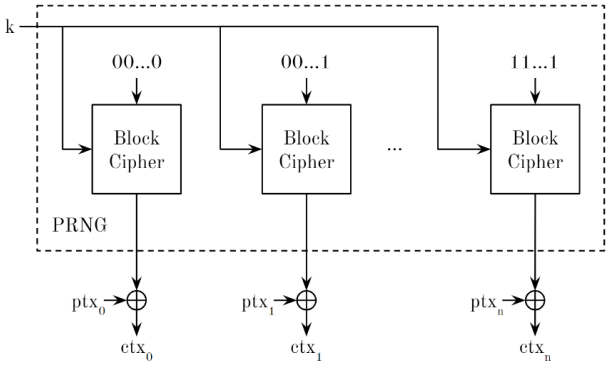
A first attempt to encryption with PRPs are ECB, which are based on the idea of split-and-encrypt.



This method is fast and parallelizable, **but** plaintext blocks with the same value will be encrypted to ciphertext blocks with the same value. Furthermore, if the attacker knows the position of a zero-filled plaintext block, she'll be able to decrypt all the messages sent.

2.3 Counter (CTR) Mode

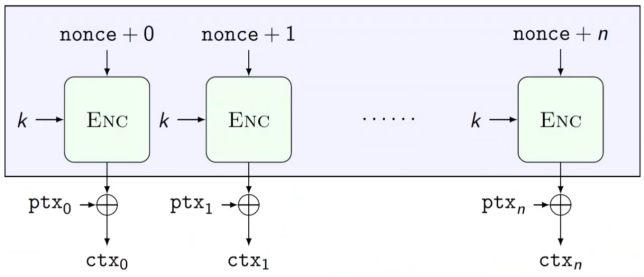
This method uses a counter and encryption to generate a unique output. The counter can start at any number and an increment other than 1 can be used, but 1 is both secure and efficient.



Ideas evoluted from CTR:

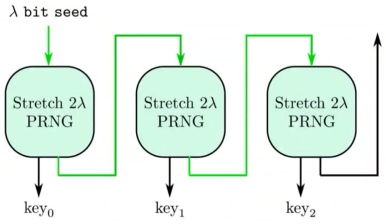
- manipulate the key every time it is used
- choose each time a different number (called nonce) as starting point of the counter
- symmetric ratcheting

The exploited main idea is always the same: adding an extra source of pseudo-randomness.



2.3.1 Symmetric Ratcheting

The idea takes the name from the mechanical component: a plugin to the cipher tool creates a different key for each block starting from a single original key (a seed). "Every single time you call it you move the ratchet".

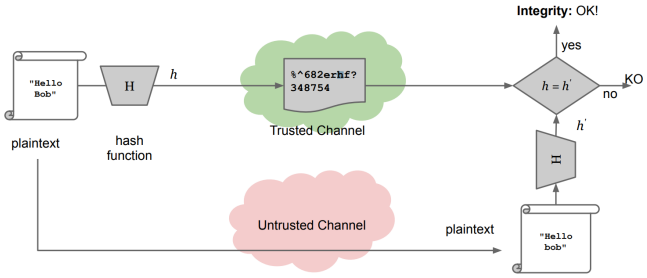


2.4 Integrity

To ensure data integrity, we can append a Message Authentication Code (**MAC**) to the message. The MAC acts as a tag that an attacker cannot forge and it's used to guarantee that the message has not been tampered. From an high-level pov the idea is:

- `computeMAC(string, key) = tag`
- 'verifyMAC(string, tag, key) = {true|false}"

Calculating message tags is time-consuming and costly. Ensuring file integrity requires comparing it bit by bit or computing a MAC by reading the entire file. This is not feasible. Can we create something similar to the ideal situation? Yes, using Cryptographic Hash Functions. However, they are **not perfect** and trade-offs are involved. One trade-off is that every file has a lower bound of compression, as per Shannon Theory.



The main idea is to have a hash function that can hash any binary stream in a way that is difficult for attackers to create collisions between different objects. A good hash function should be resistant to:

1) **first preimage attack**: inverting H. given $d = H(s)$ find s should take $O(2^d)$ hash computations.
2) **second preimage attack**: finding a different s that produces the same digest. Knowing $d, H(s)$ find another object $j \neq s$ with same $H(j) = d$ should take $O(2^d)$ hash computations.

3) **Collisions**: finding two different inputs with the same hash. Find two objects $s \neq r$ which $H(s) \neq H(r)$.

Nowadays, past functions like `SHA-1` (Google found a collission with two pdf files) and `MD5` (this one is totally broken) have been shown to be flawed, but currently, the unbroken and widely standardized `SHA-2` and `SHA-3` can produce 256, 384, or 512-bit digests. SHA-3 exists because SHA-2 utilizes mechanisms from flawed functions. Where are used hash functions? Hash functions are widely used for various purposes such as document signature, HTTPS certificate, version control (git), and ensuring data integrity in backup systems. Command `certutil -hashfile 42.pdf SHA256` uses the `certutil` Windows tool to calculate the `SHA256` (`SHA-2` variant) hash value of `42.pdf`.

2.5 Authencity

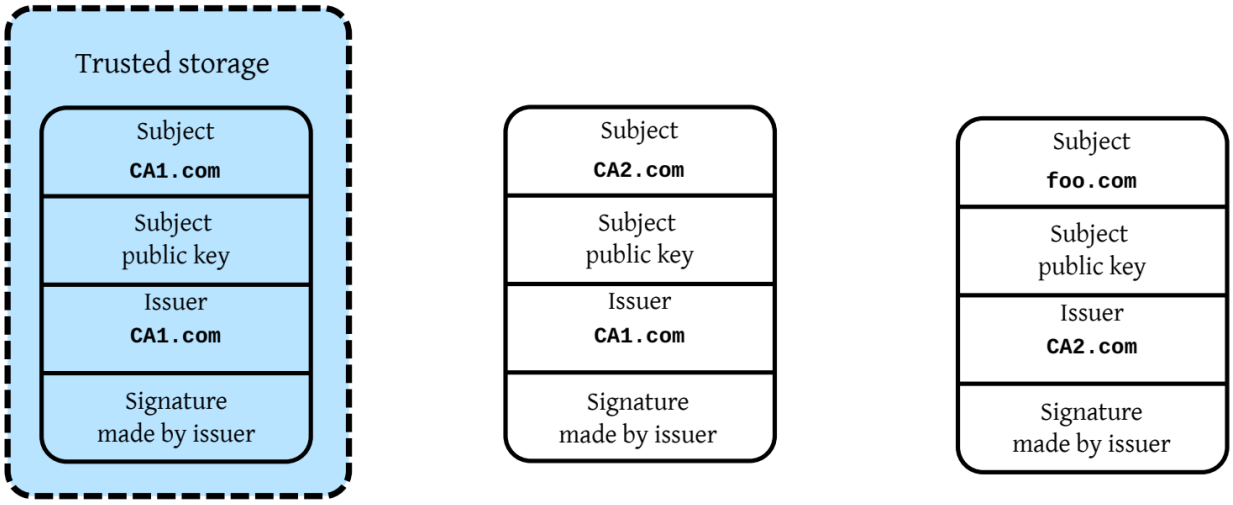
We'd like to be able to verify the authenticity of a piece of data without a pre-shared secret. With an asymmetric cipher, we can actually provide authenticity. To guarantee that the sender of the message is the same as the one who originally transmitted the public key we can use a **Digital Signature**. But a practical issue of this is that encrypt large documents is slow. A solution to slow encryption are **Cryptographic Hash Functions**: generating a fixed-length string of bytes where a change in the input message will result in a completely different output. The RSA standard, other than confidentiality, is built to provide authenticity as well: it ensures that the **message sender is who they claim to be**.

2.6 PKI CA chain

Cautionary (obvious) note: In both asymmetric encryption and digital signatures, it is crucial to bind the public key to the correct user identity.

How to achieve this?

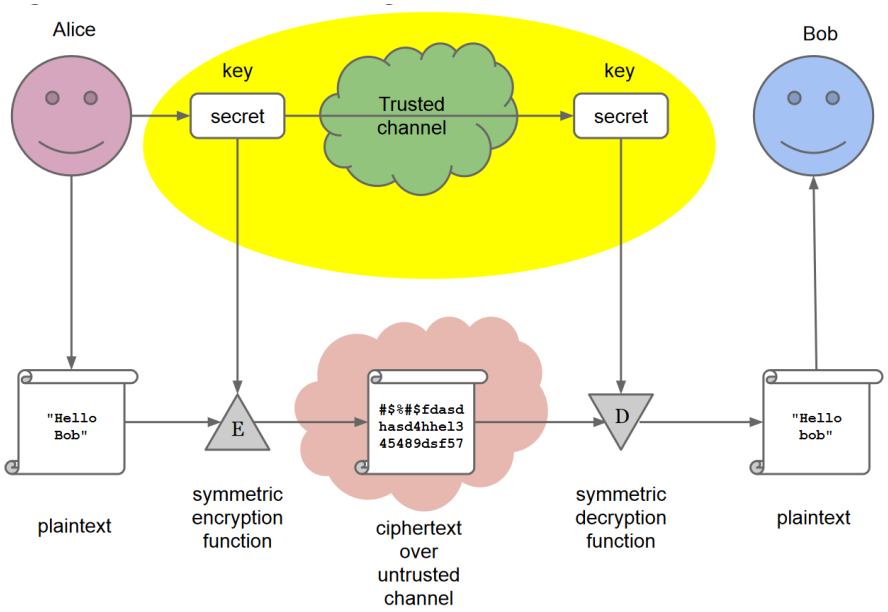
This can be achieved by using a Public Key Infrastructure (PKI) that employs a trusted third party called a Certification Authority (CA). The CA issues digital certificates that bind an identity to a public key. The CA digitally **signs** these certificates and employs a private key to do so. This creates a **chain of certificates**, known as the** Certificate Chain**. At some point in the chain, a self-signed certificate is used (**Root of trust**), which must be trusted a priori.



Generally CA is already installed in a system/browser. The root CA's list can be checked in a system, and all the root certificates installed within it are trusted when an OS is trusted.

The private key is the most important asset. If it is stolen or compromised it can't be destroyed but it's possible to revoke the digital certificate itself using one of the Certificate Revocation Lists that are stored online. The problem is that CRL cannot be verified offline, which poses another security issue.

2.7 Symmetric Encryption



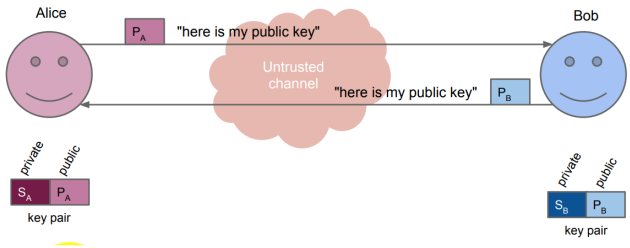
The basic idea of encryption:

- Use a key K (also known as shared) to encrypt plaintext into ciphertext
- Use the same key K to decrypt the ciphertext back into plaintext

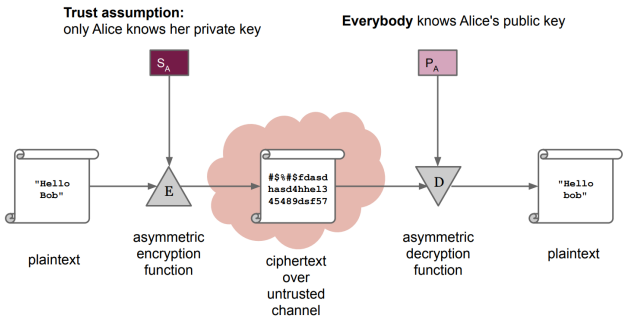
Issues with encryption:

- The key is shared ... we cannot send the key on the same channel as the message.
- An off-band transmission mechanism is needed.
- Scalability problem: a n system needs $O(n^2)$ keys

2.8 Asymmetric Encryption



And doing this we can provide authencity:



We can use only asymmetric cryptosystems but in practice it would be inefficient since they are much slower compared to symmetric cryptosystems. The solution is to use hybrid encryption schemes, where asymmetric schemes are used for key transport/agreement and symmetric schemes are used for encrypting the data. This approach combines the best of both worlds and is the basis of all modern secure transport protocols.

3 Authentication

Three factors of Authentication:

1) Something that the entity knows (to know)
2) Something that the entity has (to have)
3) Something that the entity is (to be)

3.1 To know

Why passwords are appealing?

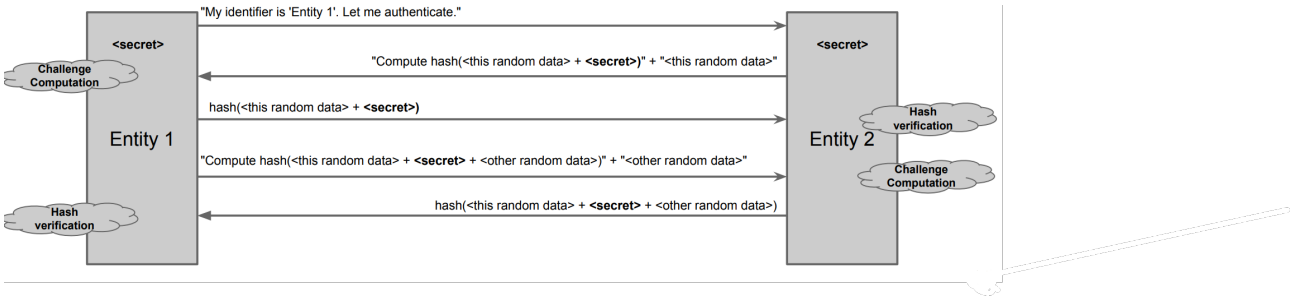
- Low cost
- Easy deployment
- Low technical barrier

Disadvantages	Countermeasures
Secrets can be easily stolen or snooped	Change or expire frequently
Passwords can be easily guessed	Are not based on any personal information of the user
Passwords can be cracked through enumeration	Are long and contain a variety of characters

But countermeasures have costs and humans are not machines:

- They are inherently unable to keep secrets
- It is hard to remember complex passwords

3.1.0.1 Secure Password Exchange How to minimize the risk that the password is stolen? **Challenge-Response** scheme is the solution. Shift from the problem to "have and pass the password" to demonstrate to have the password.



Random data is important to hide the hash of the secret and protect against replay attacks.

3.1.0.2 Secure Password Storage

- Use cryptographic protection: never store passwords in clear text, implement hashing and salting techniques: salting involves adding a random nonce to a password and it ensures that different users identical passwords have distinct hashes.
- Access control policies
- Never disclose secrets in password-recovery schemes
- It's necessary to minimize the cache of the clear password

3.2 To have

Advantages:

- Human factor: forgetting a password is more common than losing a physical key.
- Relatively low cost
- Provides a good level of security

Disadvantages	Countermeasures
Hard to deploy, can be lost or stolen.	none use with second factor.

Some examples:

- OTP: One-time password generators
- Smart cards (also with embedded reader in USB keys)
- TOTP: software that implements the same functionality of password generators

3.3 To Be

Advantages:

- High level of security
- No extra hardware required to carry around since are physical characteristics

Disadvantages	Countermeasures
Hard to deploy	none
probabilistic matching	none
invasive measurement	none
can be cloned	none
bio-characteristics change	re-measure often

Disadvantages	Countermeasures
privacy sensitivity	secure the process
users with disabilities	need alternate

Some examples:

- Fingerprints
- Face geometry
- Retina scan
- DNA
- Voice analysis: not really used since its not "stable feature".

3.4 Extra

3.4.1 Single Sign-On (SSO)

Managing multiple passwords is complex. With SSO it's possible to authenticate user across multiple domains with an identity provider. It's more usable but it's more "dangerous":

- **single point of failure**: if it's down, it's like all the services are down
- **single point of trust**: if compromised, all sites are compromised

3.4.2 Password managers

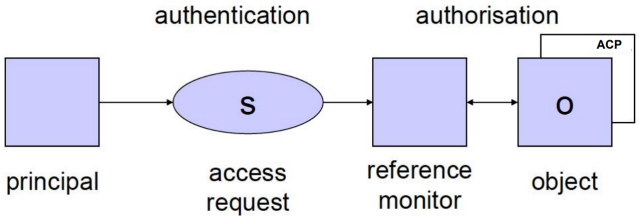
Another tradeoff of security and usability are password managers:

- Password managers eliminate the need to remember all passwords.
- Password managers allow for generating robust passwords.

but again password managers are more "dangerous":

- **single point of failure**: if it's down, it's like all the services are down)
- **single point of trust**: if comprimesed, all accounts are compromised
- Password managers are softwares...

3.5 Access Control



The Reference Monitor is responsible for enforcing access control policies, dictating who is allowed to perform what actions on which resources. It is a crucial component of all modern kernels.

Access control models:

- **Discretionary Access Control** (DAC)
- **Mandatory Access Control** (MAC), it differs from DAC since in DAC privileges are assigned by the owner of the resource while in MAC they are assigned by a central authority.
- **Role-Based Access Control** (RBAC): which is an hybrid of DAC and MAC

Examples of DAC systems are both Windows and Unix OS: which are based on users and groups of users which have privilidges over objects (files).

3.5.1 DAC

3.5.1.1 HRU model The HRU model is a computer security model that stands for Harrison-Ruzzo-Ullman. It is a formal mathematical model or a framework which is used to analyze the effectiveness of systems. Basic operations in the HRU model:

- Create or destroy subject
- Create or destroy object
- Add or remove into [s,o] matrix

Transitions are sequences of basic operations. **Safety problems** can be translated into the question "Can a certain right r be leaked through a sequence of transitions?" or in other words "Is it possible from an initial protection state to perform a transition that steals someone else's file?". If there is no such sequence, then the system is safe regarding right r. Note that this is an undecidable problem in a generic HRU model (with infinite resources).

3.5.1.2 AC matrix The **Access Control** matrix with S rows and O columns. Each cell $A[S, O]$ of the matrix defines the privileges of subject S over object O, where the privileges are read, write and own. The **safety problem** consists on checking the AC matrix to avoid leaking privileges. AC matrix is usually sparse matrix and there are different implementations to solve sparse matrix issue:

- **Authorizations table**: records **non-null** triples $S - O - A$ (Subjects can perform Action over Object).
- **Access Control Lists**: focused on objects, for each object there is a list of subjects and authorizations.
- **Capability Lists**: focused on subjects, for each subject there is a list of objects and authorizations.

The most common implementation in OS is based on **ACLs**.

3.5.2 MAC

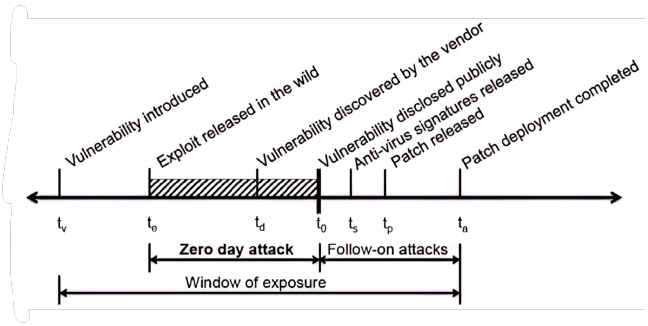
The main idea is to not let owners assign privileges, which are set by an **admin** using a classification. An implementation of MAC can be the Bell-LaPadula Model, which is based on two rules:

- A subject s cannot read an object o at a higher secrecy level
- A subject s cannot write an object o at a lower secrecy level

The main consequence of these two rules is the "tranquillity" property: by design priviledges over objects cannot change dynamically.

4 Software Security

- Bug-free software does not exist.
- Not all bugs lead to vulnerabilities.
- Vulnerabilities without a working exploit exist.
- Vulnerability-free software is difficult to achieve.



To minimize the window of exposure, the following steps should ideally be followed:

1. The vendor should find the vulnerability.
2. The vendor should patch the vulnerability.
3. The vulnerability should be disclosed.

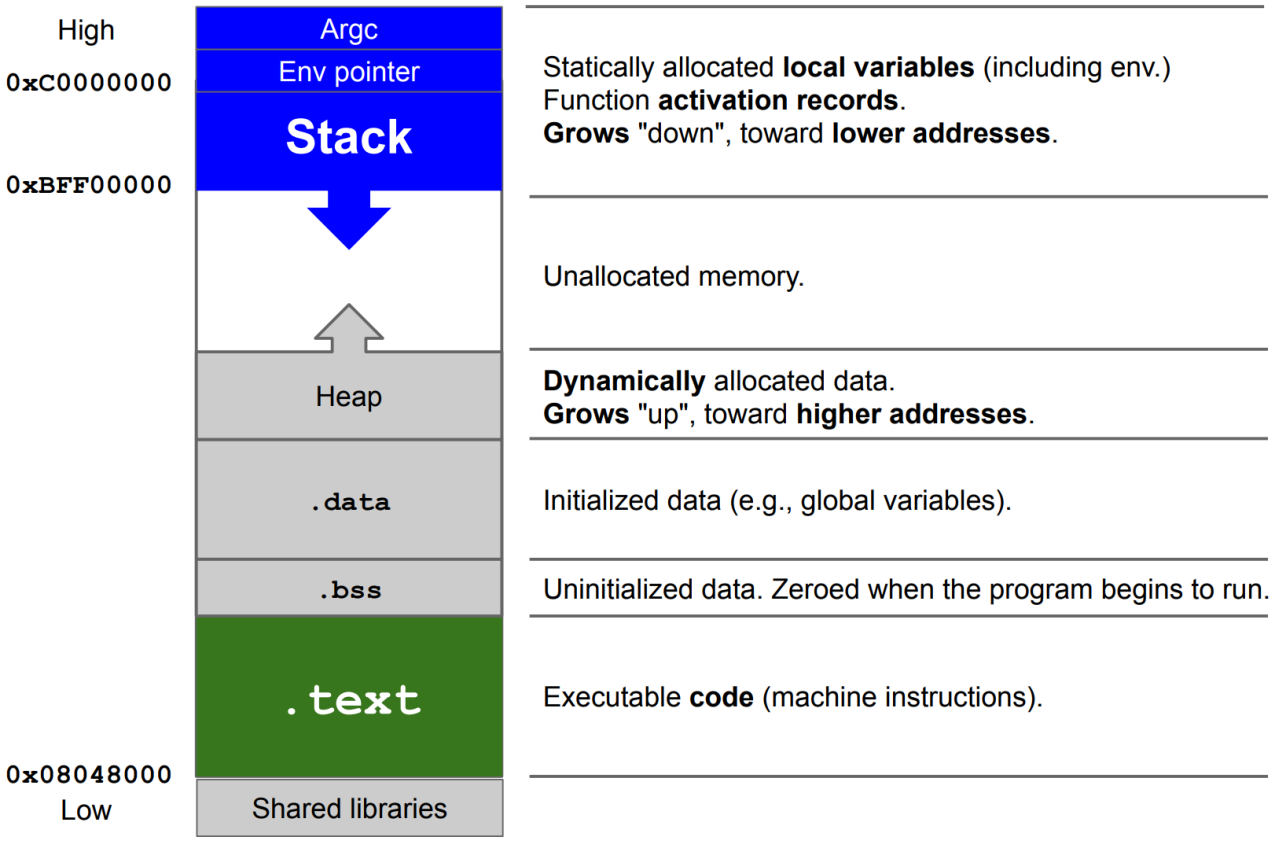
4.1 Buffer Overflow

Buffer overflow caused overwrite of the saved EIP (which contains the next instruction to execute) on stack due to lack of size checks. The corrupted return address can lead to various consequences: the program may crash, execute unexpected instructions or attackers can craft malicious payloads to take control of the program.

Proper input validation and size checks are necessary to avoid buffer overflows.

How to exploit this vulnerability? We have to jump to a valid memory address where there is a valid piece of code.

The method is called **stack smashing** : we place the code that we want to run directly inside of the buffer and rewrite the return address with the address of the buffer itself.



4.1.1 Recap stack function prologue and epilogue

Every time you call a function you will find in the disassembled code the **function prologue**:

```
push %ebp
mov %esp, %ebp
sub $0x4, %esp
```

in particular:

1) `push %ebp` saves the current stack base address onto the stack
2) `mov %esp, %ebp` saves into `ebp` the old top of the stack (`esp`)
3) `sub $0x4, %esp` allocates `0x4` bytes

Regarding the **function epilogue**:

```
leave
ret
```

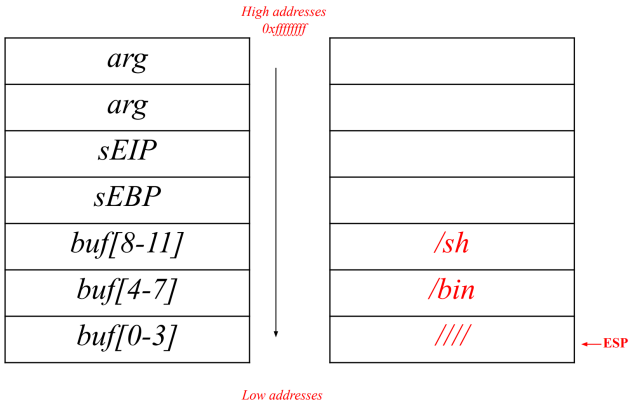
can be converted into:

```
mov %ebp, %esp
pop %ebp
ret
```

where:

1) `mov %ebp, %esp` is the exact opposite action of the prologue ... we are "saving" in the register that stores the top of the stack the `%ebp` .. basically we are **removing the all function stack segment**.

2) `pop %ebp` is used to restore the saved ebp to registry
3) `ret` is used to pop the saved `EIP` and jump there: this `EIP` will be the address of the instruction after the function call (which has just returned).



4.1.2 Stack smashing

To exploit a buffer overflow vulnerability, we need to overwrite the ESP register and have the computer jump to the code we wrote in the buffer. To obtain the assembly code, we write, compile, and then disassemble a program. However, finding the exact address of the buffer can be difficult. We can estimate it by examining the value of ESP from a debugger, but this value may not be completely accurate due to the debugger's presence. This inaccuracy makes it challenging to execute the code precisely. To solve this problem, we use a `NOP` sled. A `NOP` instruction is a command that does nothing and moves to the next cell. By inserting a sequence of `NOP` instructions before our code, called a `NOP` sled, we can jump "somewhere" in the sled since we don't know the exact address. The CPU will execute all `NOP` instructions and eventually reach our code. Historically, the goal of an attacker is to spawn a privileged shell on a local or remote machine. A shell code is a set of machine instructions that basically can do anything, including spawning a shell.

Advantages:

- Can be done remotely.
- Input can be used as code.

Disadvantages:

- Buffer size limitations.
- Memory must be marked as executable.
- Relies on accurate address guessing.

4.1.3 Alternatives techniques

- Memory that we can control
  - The buffer itself
  - Some other variable (for example **environment variable**)
- **Built-in existing functions**
- Heap overflows

4.1.3.1 Environment Variable

```
user@pc: env
HOME=/home/username
USER=username ...
```

We allocate an area of memory that contains the exploit. Then, we put the content of that memory in an environment variable. Finally, we have to overwrite the `EIP` with the address of the environment variable by filling the buffer.

Disadvantages:

- valid only for **local** exploiting

4.1.3.2 Built in existing function In the saved `EIP` we will place the address of a function, for example `system()`. But the CPU is expecting the arguments of the function but also the returning address (after the call of the function).

Advantages:

- Work remotely and reliably
- No need for executable stack
- Functions are usually executable

Disadvantages:

- Stack frame must be carefully prepared (the attacker has to emulate the function frame)

4.1.4 Defending agains Buffer Overflow

Many strategies exist apart from the correct best practices to accept input from the user (for example, the `%7s` syntax is a limit specifier in the placeholder which limits the string accepted to 7). Most advanced countermeasures transparent to the developers are:

- ASLR
- not executable stack (NX)
- stack canary

More specifically:

- Defenses at **source code** level involve the process of finding and removing vulnerabilities:
  - mainly developers cause buffer overflows
  - using safer libraries
  - dynamically memory management (e.g. Java) that makes them more resilient (*You can't mash my stack if the stack doesn't exists*).
- Defenses at **compiler level** are focused on making vulnerabilities non-exploitable.
  - warnings at compile time
  - randomized reordering of stack variables make the stack mashing more difficult.
  - **canary** is the most important mitigation.
- Defenses at **operating system** level aim to increase the attacks difficulty.
  - **non-executable stack** to clearly distinguish data from code. But this actually is bypassed since it is possible to point the return address to existing machine instructions (code-reuse attacks)
  - **address space layout randomization (aslr)** is a technique which translates the stack at each execution at random, which makes impossible to guess the addresses correctly.

4.1.4.1 Canary Canary mecahnism consists of a variable which is usually placed between local vars and control vars (such as `EIP`/`EBP`) so that it can be used as a "flag" which is checked every time a function returns. A **canary** could be read but if it's correctly implemented, every time the program is run, the canary value changes! So it's impossible to get around it. There are different types of canaries that can be used for protection:

- **Terminator canaries**: made with terminator characters that cannot be overwritten by string-copy functions.
- **Random canaries**: random sequence of bytes are chosen when the program is run.

- **Random XOR canaries**: same as random canaries, but they are XORed with a part of the structure that needs protection. This helps to protect against non-overflows.

4.2 Format string bugs

The vulnerability arises because many programming languages provide functions for formatting strings using placeholders that are replaced with values at runtime. For example, in C, `printf()` is used to print formatted output on console. If the user input contains special characters such as `%s` (string), `%d` (integer), `%x` (hexadecimal), etc., and these are not properly sanitized by the program, there is the possibility to leak information from the stack.
For example, if arguments are omitted, we can read the stack:

```
printf("%i, %i, %i", a, b, c); // NOT vulnerable
printf("%x, %x, %x"); // vulnerable
```

Placeholders modifiers of interest:

- `I$` (or `pos$` or `n$`) where `I` is any number (integer) is a placeholder modifier which is place between `%` and the **conversion specifier**. It prints the value of the `I`-th argument passed to the `printf` function, for example: `%2$d` will print the second argument as a decimal number.
- `%n` is another **conversion specifier** which writes in the **address pointed** by the argument, the number of chars (bytes) printed so far in the `printf`. In this example, after the call of `printf`, we will have that `i=5` since `hello` has length of 5 chars. `%n` allows us to write directly on the stack and we can exploit this to change any memory cell, maybe an address.

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

- `%Ic`, where `I` is an integer that specifies how many times the first character argument should be printed.

The idea:

1. Put, on the stack, the address of the target cell to modify
2. Use `%I$x` to go find it on the stack using the "displacement" `I` as `pos`.
3. Use `%n` to write the number of bytes written so far (this can be manipulated by `%Ic`) in the target, which is pointed to by address you have pushed in 1.

The problem with this is that to write an address we will have to manipulate the bytes printed so far with `%Ic` with an **huge** number of characters (billions). This is not feasible on any device (memory limits). So we divide the single 32 bit write in two separate 16 bits writes using only one format string (actually it can be seen as a combination of two format strings):

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

Note that in `%pos$hn` (composed by `%hn` and `pos$`), we use `%hn` instead of `%n` because we are writing 16 bits instead of 32 and we do not want to overwrite the two following bytes with 0s. Also remember that `<lower_value>` rapresents the first 2 bytes of the word that we want to write. The `<higher_value>` will not rapresents in decimal form the second part of the address, but the **difference between lower_value - higher_value>** since the mechanism of `%Ic` ! For the same motivation when we are writing the first value we have to keep in mind how many chars (bytes) the `printf` has already printed at that point: `<low_value> = <lower_part> - len(printed)` .

4.2.0.1 Example We need to write `0xbeefdead` to `0xffbfdd9c`. As in this case `0xbeef` < `0xdead`, we need to **swap**: in the formula we must to write `0xbeef` before so, the first target is not just `<target>` but necessary `<target+1>`. The general format string structure is:

```
<where to write+2><where to write>
```

`%<low_value>c%<pos>$hn`
`%<high value>c%<pos+1>$hn`

- Where to write (inverted) = `\x9c\xdd\xbf\xff`
- Where to write + 2 (inverted) = `\x9e\xdd\xbf\xff`
- If in the `printf` we have already written some characters, we have to count them. We have written a total of 8 characters (bytes) for the target address, which is represented by 4+4 bytes. Along with this, we have also written 16 additional characters (just for this example) within the `printf` function. To calculate the remaining characters that need to be written, we subtract `24` from the total.
- low value = `0x49d0 -> 48879 - 24`
- high value = `0xf7e3 -> 57005 - 48879`

4.2.1 A word on countermeasures

Buffer overflow countermeasures like **ASLR** and **XOR canary** are not always effective to prevent also this vulnerability. Compilers warns when a format string function is used without the right specifications. Sometimes is possible for the attacker to use the format string to leak the value of the canary and rewrite it using the buffer overflow vulnerability.

5 Web security

The basic structure of a web app or website is based on HTTP(S) protocol. It's very important in this context to filter and validate client input carefully. It's common to assume that all the clients are attackers, since the client is the only part of the software that the attacker can use.

5.1 SQL Injection

SQLi can happen every time the input fields are not-sanitized input in web forms. Closing a string with the caracter `'` or the query injecting `;`, as well as commenting (`--`) or using operators such as `UNION` and `INTERSECTION` can be exploited to bypass password checking or completely bypass the query system. Examples:

```
SELECT name, phone, address FROM Users
WHERE Id='userinput';
```

can become:

```
SELECT name, phone,address FROM Users
WHERE Id='' or '1'='1' ;--';
```

injecting `' or '1'='1' ;--` . Also `UNION` operator permits to concatenate the output of multiple queries, even a query on a different table from the original one.

SQL injection prevention:

- **Prepared statement**: generate a query with placeholders where the users input is seen as parameter and not as SQL. **Not** build queries by string concatenation.
- properly sanitize input
- limit query privileges to limit damages

5.1.1 Blind SQLi

Blind SQLi is similar to normal SQLi injection but involves retrieving data not directly but posing a series of queries to the database. An example is:

```
Username = "whatever' AND EXISTS (SELECT * FROM User WHERE username='martino' AND password LIKE 'a%')"
```

This injection uses the `EXISTS` statement to check if a password exists that starts with the letter "a". If username password indeed begins with "a", the login attempt will be successful. Otherwise, the login will fail. To discover the entire password, the attacker can employ a bruteforce technique. By trying different starting letters for the password in the injection code, the attacker can determine the correct value through trial and error.

5.2 XSS

We can call XSS (Cross Site Scripting) any stuff that can be exploited by the attacker to inject into a web page a script. There are three types from an higher point of view:

- **Stored XSS**: the attacker input is stored on the target server's database. This happens if the malicious code is not filtered out.
- **Reflected XSS**: the malicious code is not stored on the server. An example of reflected XSS involves appending a parameter and a JavaScript script to a URL via a GET request. If the input is not sanitized, this can be risky as the script is treated as a parameter and may result in an error page or load a generic page containing the injected script. Example: Let's say the search term gets reflected in the page of the website in case of error. The URL for the search is like this: `www.example.com/search?term=test`. An attacker could craft a URL like `www.example.com/search?term=<script>malicious_code_here</script>` and trick a user into clicking it, causing the execution in the user's browser.
- **DOM-based XSS**: basically the previous one ... the only big difference is that the attack remains completely in the user's browser.

Examples:

5.2.1 XSS bypasses Same Origin Policy

SOP is a paradigm which is based on the rule:

> "all client-side code (e.g., JavaScript) loaded from origin **A** should only be able to access data from origin **A**"

where an origin is identified by the tuple: `<PROTOCOL,HOST,PORT`. It's easy to see that XSS bypass this simple rule and also in nowadays it's not rare that applications permits to share resources to enhance services. A solution could be to blackilist everything that may be misinterpreted but it's not a good approach.

5.2.2 CSP

CSP3 (Content Security Policy) is the most valid defensive measure to mitigate XSS. It's a set of directives sent by the server to the client in the form of HTTP response headers which specifies what should be trusted and what shouldn't. CSP in theory is effective against XSS, but since the policies are written manually by devs and they must be to keep updated, they often are not effective to mitigate such attacks.

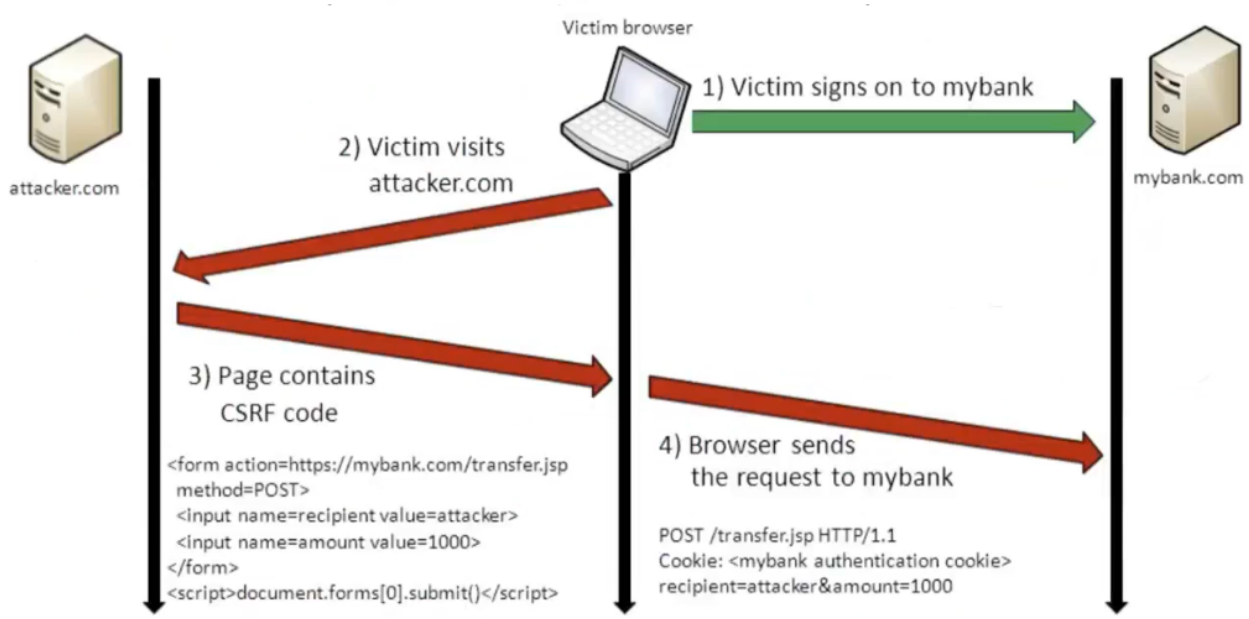
5.3 Cookies

Cookies are commonly used to store information about the HTTP session (making it "stateful"). Cookies are basically small text files on a user's browser. They can also be used to track the user across different websites. However, this can create security vulnerabilities. For example, a server can generate a SessionID as a cookie on the client to identify and store conversational data about them. It is essential to ensure that session tokens are crafted in a secure manner to prevent exploitation. If used for authentication, sensitive data should **not be stored in plaintext**. If used for session identification, they should **not be predictable**. Additionally, session tokens should have a clear and different **expiration date** from the cookie itself.

5.3.1 Cross-Site Request Forgery (CSRF)

A CSRF attack exploits cookies to force a user to execute unwanted actions on a web application in which he is currently authenticated using ambient credentials (**cookies**).
In theory, CSRF attacks occur when a state-changing action is allowed based on cookie validation. The attack involves four steps:

1) victim signs in on a website using cookies
2) victim visits a malicious site or clicks on a link
3) malicious site generates a fake request using XSS techniques
4) the victim's web client executes the malicious web request, tricking the original website's authentication by using ambient variables.



Another way to use CSRF is to **steal** a session cookie: the attacker can gain access to the admin's session by exploiting session cookies. They do this inserting in a vulnerable page a XSS code designed to extract and publish in the website as a comment/post the session cookie. This allows the attacker to collect the session cookie of the admin.

5.3.1.1 CSRF Mitigation Techniques CSRF attacks can be mitigated using techniques such as CSRF token and Same Site Cookies policy.

The CSRF token method involves generating **random** tokens associated with the user's session. These tokens are regenerated for every request. The client sends these tokens to the server, and the requests are only confirmed if they match the tokens stored on the server. It is crucial that the token is implemented correctly! For example in some old exams, the CSRF token was static and not saved in the database, making it vulnerable to SQL injection attacks.

The **Same Site** Cookies policy works by setting an additional 'SameSite' attribute when the cookie is created. This attribute instructs the browser to attach the cookie only to same-site requests, rendering CSRF attacks ineffective.

6 Network Security

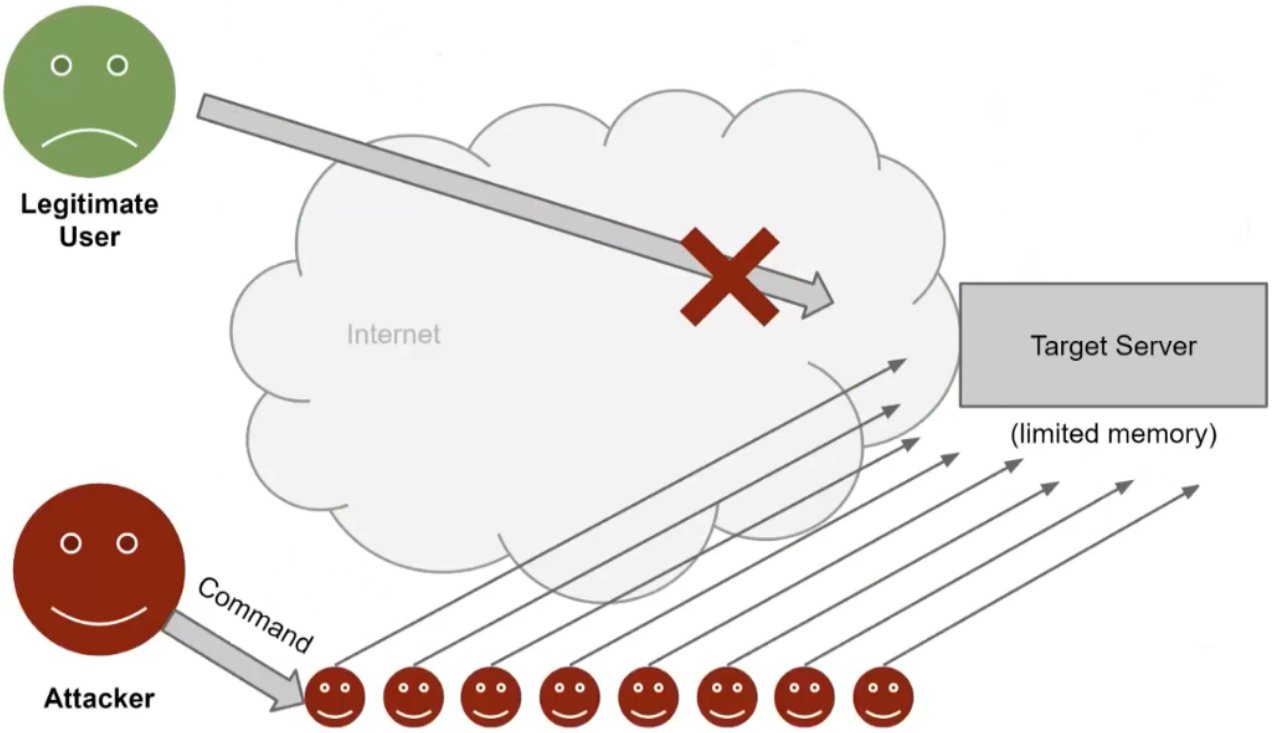
6.1 Network attacks

We can distinguish 3 main types of attacks:

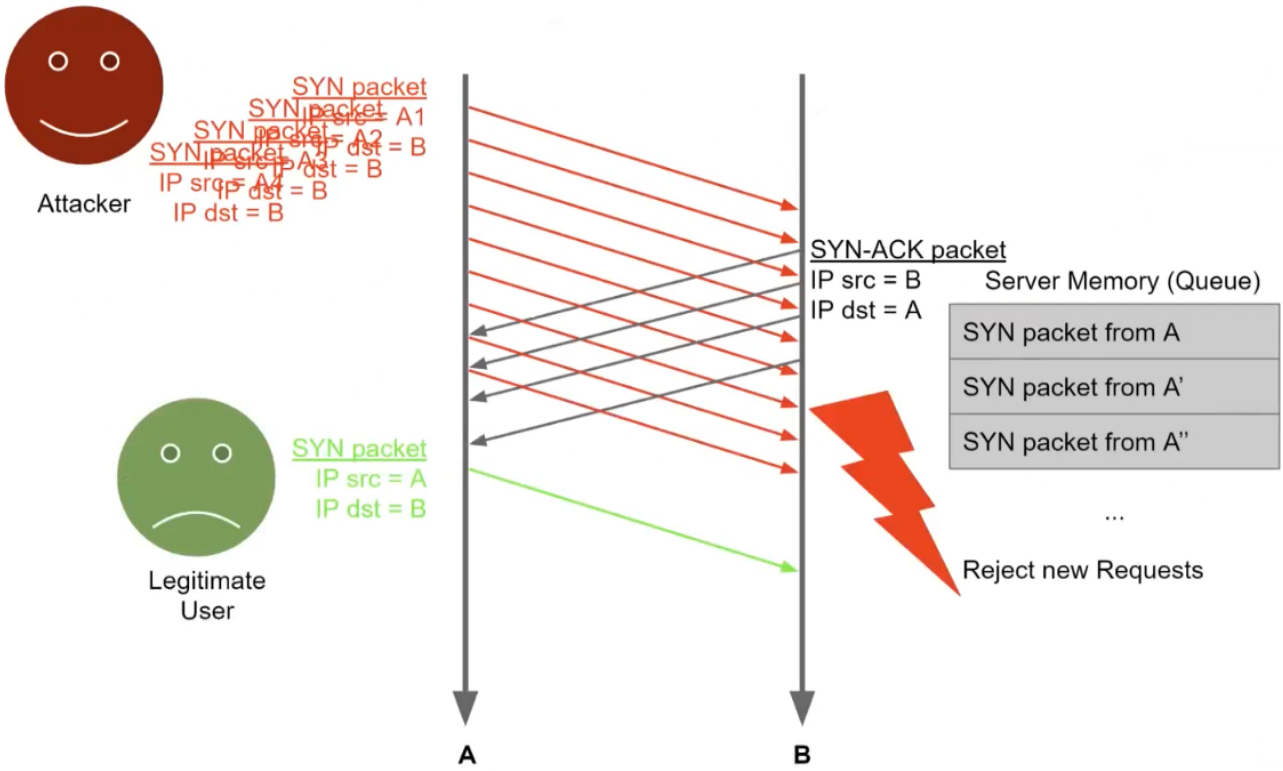
- **Denial of Service** (DoS): make a service unavailable (against availability)
  - killer-packets: malicious packets crafted in specifically to make the application/OS that reads them to crash or become unresponsive.
  - flooding: sending tons of requests to choke the connection to the service.
- **Sniffing**: abusive reading of network packets (against confidentiality)
- **Spoofing**: forging of network packets (against integrity and authenticity)

6.1.1 Dos

Flooding attacks, such as DoS attacks, cannot be completely eliminated.



The security problem increases when there is a way for the attacker to **multiply** his effort in doing the attack: the attacker uses `x` amount of resources but consumes `Nx` resources on the server. We want to take away the possibility of having that multiplier. One example of this multiplier factor is **SYN flood**, which exploits the three-way handshake of TCP. The attacker sends a lot of SYN packets with spoofed source IPs, never acknowledging them. The server needs to reply with SYN-ACK and **store** pending connections in a queue, consuming bandwidth and **memory**. The limiting factor is the server's memory usage.



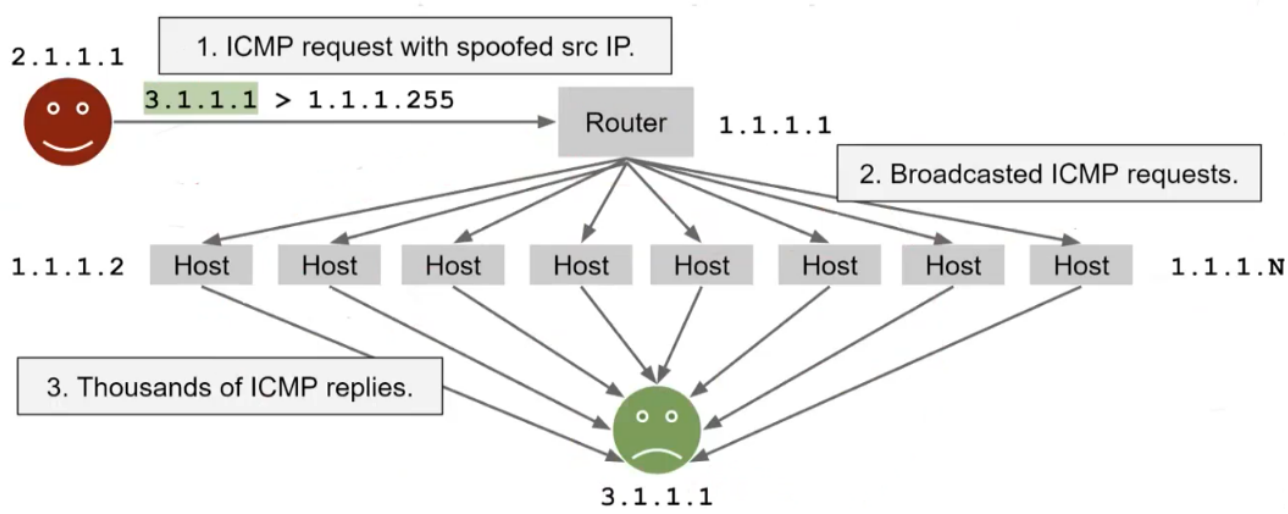
Although this attack cannot be avoided due to how TCP works, mitigation is possible by removing the multiplier (storing connections in memory).

6.1.1.1 DDoS In the case of Distributed DoS (DDoS) the attacker can control a large number of machines (a botnet) and have them send requests to a target server to consume its bandwidth. The multiplier is in the controlled machines: one packet/command will be multiplied by the number of machines. Many protocols on the internet have characteristics that make them suitable for DoS and DDoS attacks (they have some built in multiplier). BAF (Bandwith Amplification Factor) = multiplier of response payload to request payload.

Protocol	all	BAF 50	PAF 10	all	Scenario
SNMP v2	6.3	8.6	11.3	1.00	GetBulk request
NTP	556.9	1083.2	4670.0	3.84	Request client statistics
DNS NS	54.6	76.7	98.3	2.08	ANY lookup at author. NS
DNS OR	28.7	41.2	64.1	1.32	ANY lookup at open resolv.
NetBios	3.8	4.5	4.9	1.00	Name resolution
SSDP	30.8	40.4	75.9	9.92	SEARCH request
CharGen	358.8	n / a	n / a	1.00	Character generation request
QOTD	140.3	n / a	n / a	1.00	Quote request
BitTorrent	3.8	5.3	10.3	1.58	File search
Kad	16.3	21.5	22.7	1.00	Peer list exchange
Quake 3	63.9	74.9	82.8	1.01	Server info exchange
Steam	5.5	6.9	14.7	1.12	Server info exchange
ZAv2	36.0	36.6	41.1	1.02	Peer list and cmd exchange
Sality	37.3	37.9	38.4	1.00	URL list exchange
Gameover	45.4	45.9	46.2	5.39	Peer and proxy exchange

6.1.1.2 ICMP flooding

ICMP is the protocol used for the classical ping (echo request and echo reply). **ICMP flooding** nowadays works only inside the LAN and also the RAM's of devices theoretically can support such overload.



6.1.2 Sniffing and spoofing

Often used in combination: spoof an machine (for example a gateway) to sniff packets of the lan or in general to act as man in the middle (MITM).

6.1.2.1 ARP spoofing

Spoof the MAC address of an ARP reply. Since there is no check, the machine who made the ARP request will trust the first answer that comes, also **poisoning** the ARP cache with the address of the attacker. Now the attacker can use this to pretend being someone else.

Possible mitigations are:

- include a random nonce in each request
- anomaly detection: if someone is constantly sending out ARP replies is weird.
- address conflict: the correct reply will eventually reach the host that could notice the conflict and alert the user.

6.1.2.2 MAC flooding

Fill the CAM table of switches by sending tons of spoofed packets. If the CAM table is full: the switch cannot cache ARP replies and must forward every packet to every port (like a hub does). The broadcasting of every packet permits the attacker to be able to sniff any packet if it's in the LAN.

The main mitigation is:

- Port security (CISCO terminology) tells the switch roughly how many hosts are supposed to be connected to some port and refuse to save more addresses.
- "hard-coded" MAC addresses

6.1.2.3 DNS cache poisoning

The objective is to poison the victim DNS server, which is non-authoritative, by impersonating the authoritative server. This can be achieved through a recursive query of the victim's DNS server by the attacker. The victim after the query, contacts the authoritative server for the requested domain. The attacker has in some way to reply to the victim's DNS server before the legitimate authoritative server does. Once the attacker successfully impersonates the authoritative server, any client DNS request will be redirected to the attacker's chosen IP address.

6.1.2.4 DHCP poisoning

DHCP is another **unauthenticated** protocol since it needs to work without any configuration. The attacker can spoof the DHCP answer to manipulate the information sent to the client:

- IP address assigned
- DNS server
- default gateway

Of course the attacker needs to be on the local network and reply first.

6.1.2.5 ICMP redirect

The ICMP protocol not only includes "ping message" but also the *redirect* one. The ICMP redirect is used by routers to inform hosts about a better route for a specific destination. This allows hosts to update their routing table with the new gateway for that route. Attackers can manipulate these messages to redirect hosts to malicious gateways and basically doing sniffing/MITM (Man-in-the-Middle) attacks. This attack requires the attacker to be in the same network and modern OS typically ignore ICMP redirect messages by default.

6.2 Firewalls

Firewalls are network access control systems that verify packets flowing through them. A firewall is a computer which can be considered as a stupid "bouncer at the door": they cannot perform content analyze (for that we need a proxy.

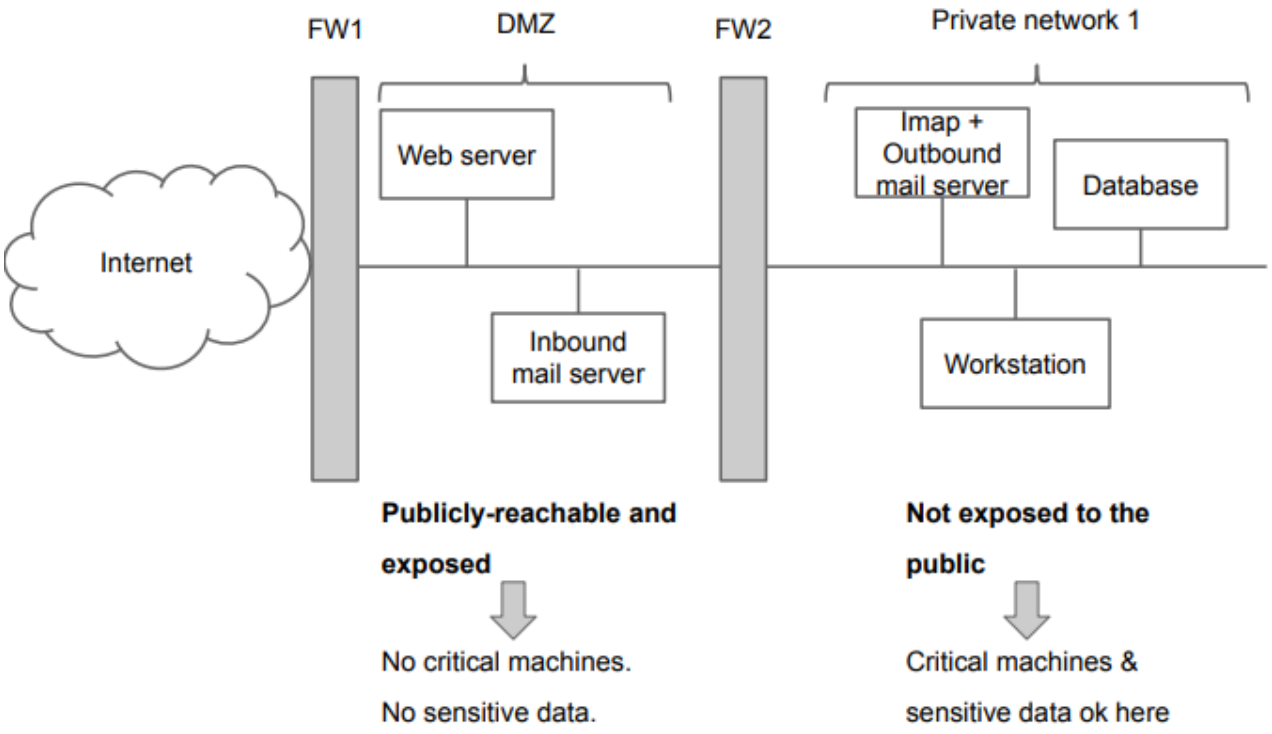
There are different types of firewalls, such as:

- Stateless Packet Filters
- Stateful Packet Filters

The main difference between the two is that the stateful one keeps track of connections and not just packets. This makes deny rules safer but the performance is bounded by connections and not by packets.

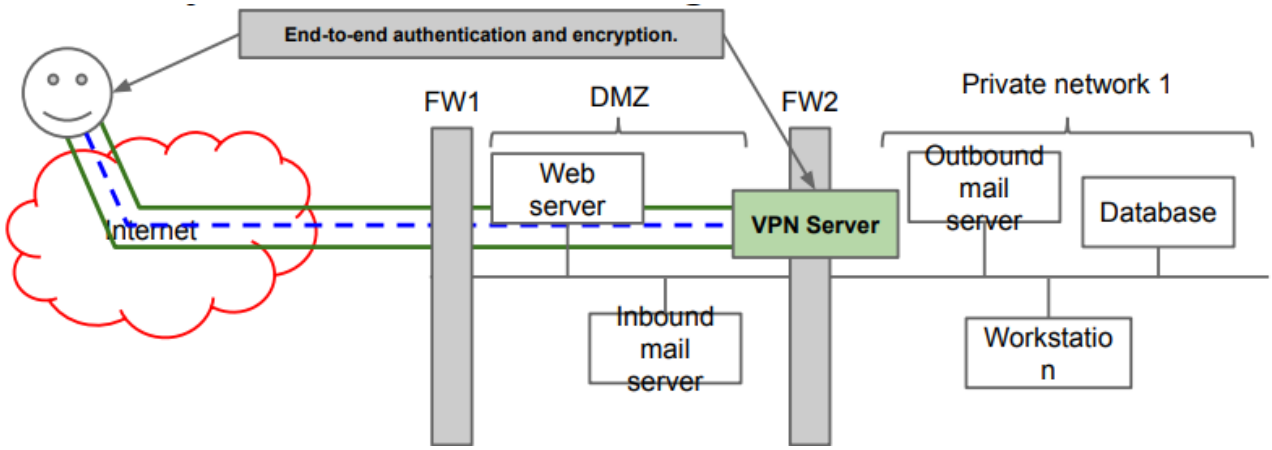
6.3 Network Design

To secure a network, we split the network by privileges zones and we use firewalls to regulate access. The DMZ zone can host public servers (web, FTP, public DNS server, intake SMTP) and is assumed to be exposed as the internet.



6.3.1 VPNs

We can also use a VPN to build an encrypted overlay connection on top of a public network to ensure the confidentiality, integrity, and availability of data transmitted.



A VPN can be configured in:

- Full tunnelling where every packet goes through the tunnel.
- Split tunnelling where the traffic to the Corporate Network is routed through the VPN, while the other traffic goes to the ISP.

7 Malware

Malware violates security policy and can be classified in three types:

- Virus: self-replicates but needs a host program.
- Worms: spread through vulnerabilities or social engineering.
- Trojan: looks benign but has a hidden malicious program and allows remote control.

7.1 Virus obfuscation techniques

- **Metamorphism**: ability to produce different versions of itself. It's often achieved by adding random dead code, such as useless assembly commands, like `nop` or `addi eax, 1` that serve no purpose in the code. If the provided assembly code contains such dead code, it is likely that metamorphism was used. If the traces of the same malware differ between different runs, and here's no "mutation engine" (typical of polymorphism) in the code, then metamorphism is evident.
  - inserting nops
  - reordering sections
  - inserting useless instructions
- **Polymorphism**: ability of malware to encrypt and decrypt itself. Useless assembly commands, once decrypted, can reveal actual working malware. Often this malwares repeatedly perform `XOR` operations on data in memory and stores it elsewhere. Encrypt and decrypt at each time the malware with a different key is useful since it's like there are multiple versions of the same code with the same semantic.
- **Dormant Period**: strategy to do nothing for a specific period of time, using long loops or commands like `sleep(10000);`.
- **Event Triggering**: malware that runs an infinite loop, continuously checking the response received from a specific domain or a specific event.

7.2 Virus evasive techniques

- **Anti- virtualization**: Generally to analyze malware, it can be launched in an isolated virtual machine to observe its behavior. Some malware may have techniques to detect virtual machines, such as by checking environment variables.

7.3 Analysis techniques

Theoretically anti-malware software cannot directly detect if a virus can spread or not. Therefore, they use a blacklisting approach to block known malware samples. These results are basically derived from the Halting Problem.

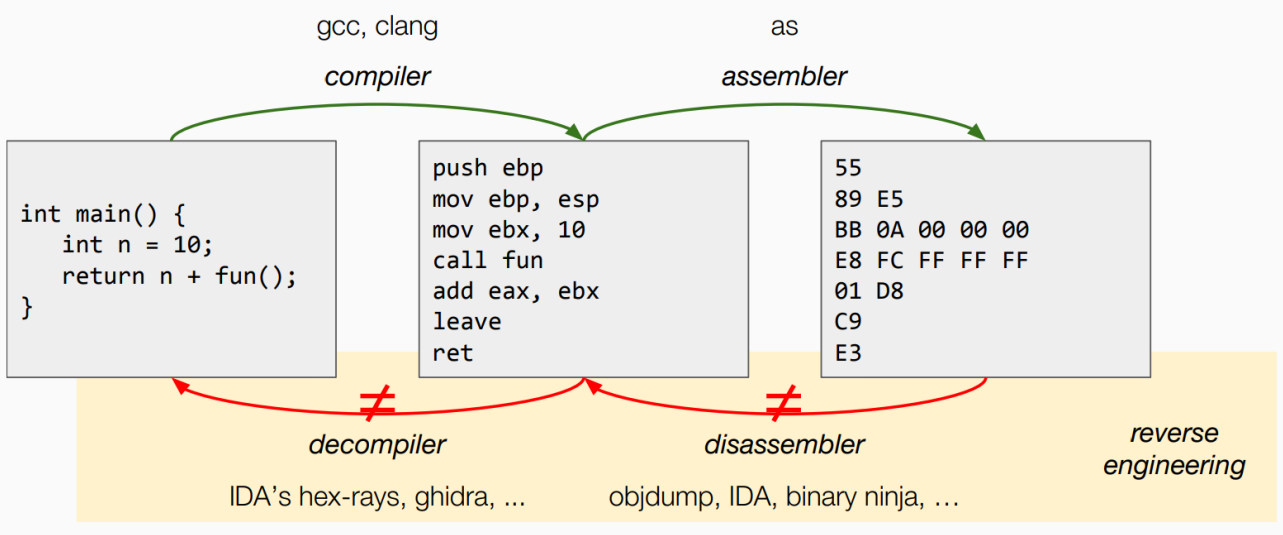
Two main ways of analyzing malware:

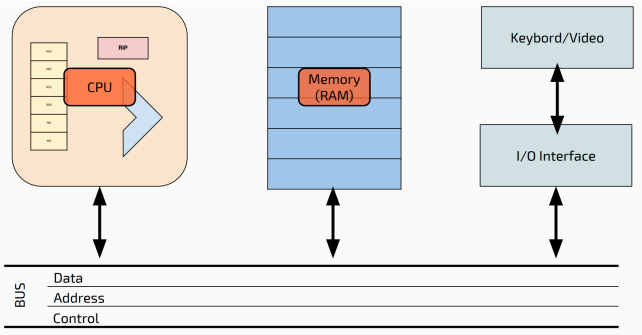
- **Static Analysis**: "manual" analysis generally used for dormant-code and anti-virtualization cases. For example in case of a sleep function with `0x1000` as a parameter, then we have to wait for at least `0x1000` seconds in order to see the true behavior of the malware.
- **Dynamic Analysis**: it's used to "automatically" find relevant information of the malware, such as syscalls, calls to library functions, unpacked instructions executed, and we can feed this information to heuristics or ML algorithms to detect the malware.

7.3.1 Best practices

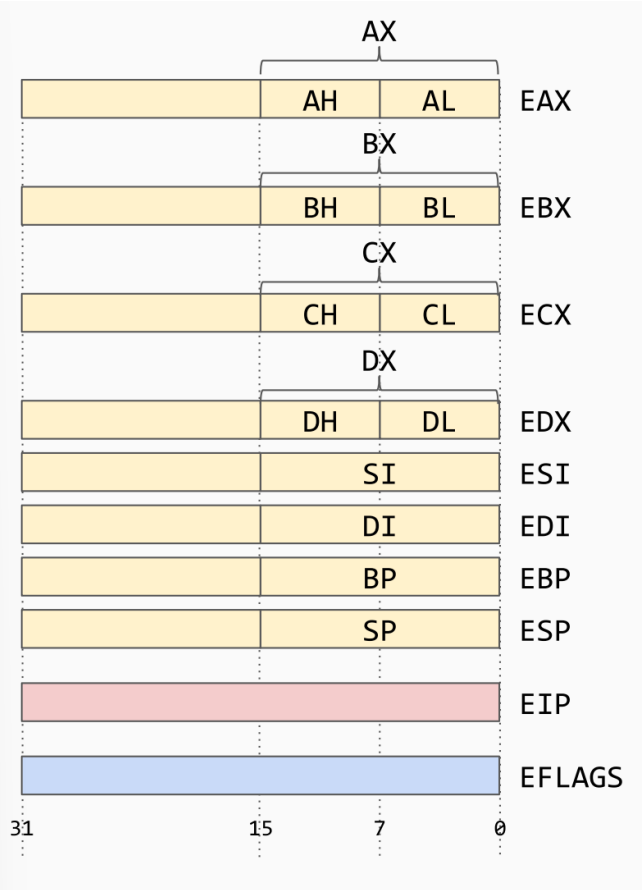
- Polymorphism -> dynamic analysis
- Metamorphism -> dynamic analysis
- Trigger based/Dormant code/anti-vm -> static analysis

8 x86 Crash Course (WIP)





Registers:



- General-purpose registers: `EAX`, `EBX`, ECX, `EDX`, `ESI`, `EDI`, `EBP`, `ESP`
- `ESI` and `EDI` used for string operations
- `EBP` used as base pointer
- `ESP` used as the top stack pointer
- `EIP`
  - Accessed implicitly, not explicitly
  - Modified by jmp, call, ret
  - Value can be read through the stack (saved IP)

`EFLAGS` register is used basically for control flow decision.

x86 Fundamental data types:

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

Moving the value 0 (immediate) to register `EAX`:

```
mov eax, 0h
```

`0h` means 0 in exadecimal

```
mov [ebx+4h],0h
```

To move a **memory** value from one point to another, it is necessary to pass through the CPU by using a register.

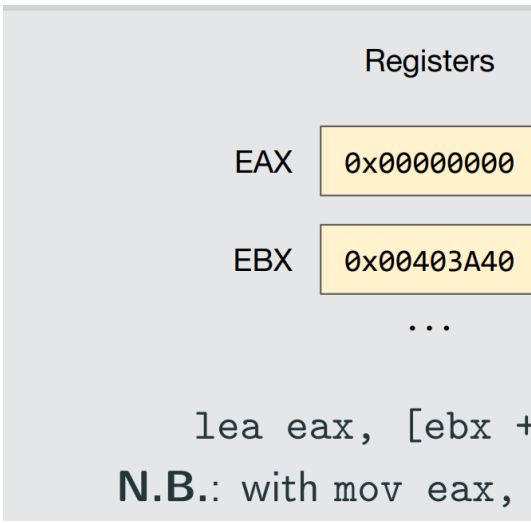
```
mov eax, [ebx]
mov eax, [ebx + 4h]
mov eax, [edx + ebx*4 + 8]
```

8.1 Basic instructions

Instruction = opcode + operand

Most important:

- Data Transfer: mov, push, pop, xchg, lea
- Integer Arithmetic: add, sub, mul, imul, div, idiv, inc, dec
- Logical Operators: and, or, not, xor
- Control Transfer: jmp, jne, call, ret
- And many more...
- `mov` **destination** , **source**
  - `MOV eax, ebx`
  - `MOV eax, FFFFFFFFh`
  - `MOV ax, bx`
  - `MOV [eax],ecx`
  - `MOV [eax],[ecx]` **NOT POSSIBLE**
  - `MOV al, FFh`



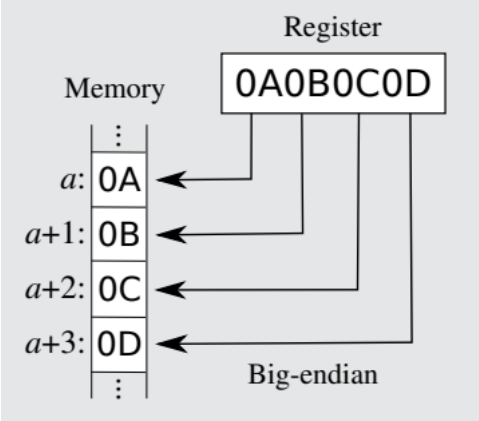
- `lea` **destination** , **source** to store **the pointer** to the memory, not the value
- `add` **destination** , **source** makes: `dest <- dest + source`
- `sub` **destination** , **source** makes: `dest <- dest - source`
- `mul` **source** : one of the operands is **implied** (it can be `AL`,`AX` or `EAX`) and the destination can be `AX`,`DX:AX`, `EDX:EAX` (the results could eventually occupy two registers)
- `div` **divisor** : dividend is **implied** (it's in `EDX:EAX` according to the size)
- `cmp` **op1**, **op2** computes `op1 - op2` and **sets the flags**
- `test` **op1**, **op2** computes `op1 & op2` and **sets the flags**
- `j<cc>` **address** to conditional jumps, reference: http://www.unixwiz. net/techtips/x86-jumps.html

- `jmp` **address** is unconditional jump
- `nop` no operation, just move to next instruction.
- `int` value is software interrupt number.
- `push` **immediate** (or register): stores the immediate or register value at the top of the stack and obviously decrements the `ESP` of the operand size.
- `pop` **destination**: loads to the destination a word off the top of the stack and it increases `ESP` of the operand's size.
- `call`: push to the stack the address of the next instruction (not the function called) and move the address of **the first instruction of the callee** into `EIP`
- `ret` : it's the opposite of `call` function ... restores the return address saved by `call` from the top of the stack. It's equivalent to `pop eip` .
- `leave` restores the caller's base pointer and it's equivalent to say: `mov esp, ebp` and `pop ebp` ... basically you are "deleting" the func's frame.

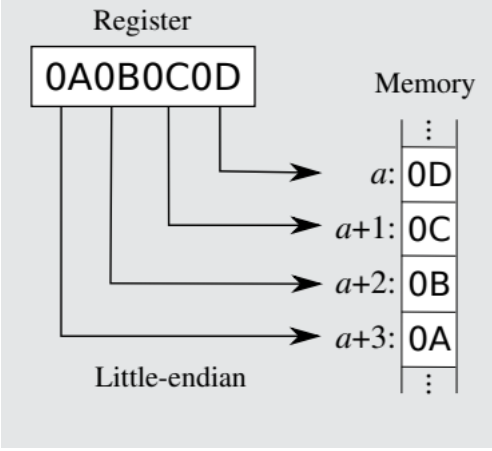
Endianness

Endianness refers to the order in which bytes of a data word are stored in memory.

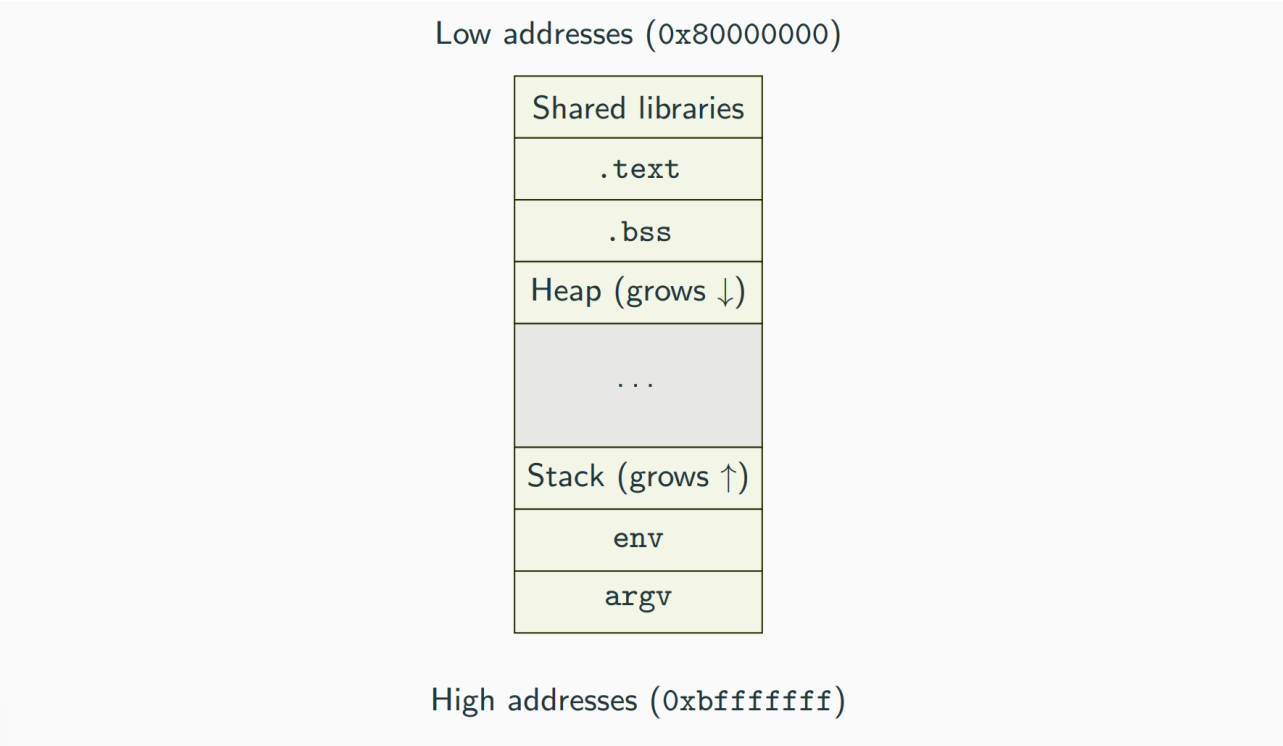
Big endian (left)



Little endian (right)



8.2 Program Layout and Functions STACK



PE (Portable Executable): used by Microsoft binary executables • ELF: common binary format for Unix, Linux, FreeBSD and others • In both cases, we are interested in how each executable is mapped into memory, rather than how it is organized on disk.

- PE is used by Microsoft binary executables while ELF is common in Unix, Linux, FreeBSD, and others.
- The focus is on how the executable is mapped into memory rather than how it is organized on disk.

How an executable is mapped to memory in Linux (ELF) ?

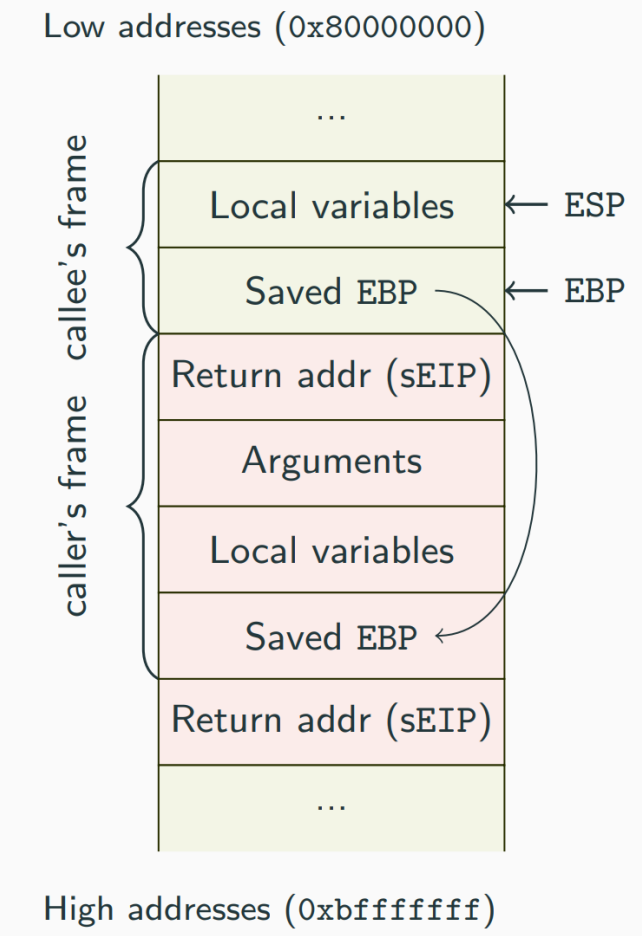
Executable	Description
.plt	This section holds stubs which are responsible of external functions linking.
.text	This section holds the "text," or executable instructions, of a program.
.rodata	This section holds read-only data that contribute to the program's memory image
.data	This section holds initialized data that contribute to the program's memory image
.bss	This section holds uninitialized data that contributes to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run.
.debug	This section holds information symbolic debugging.

Executable	Description
.init	This section holds executable instructions that contribute to the process initialization code That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called main for "C" programs).
got	This section holds the global offset table.

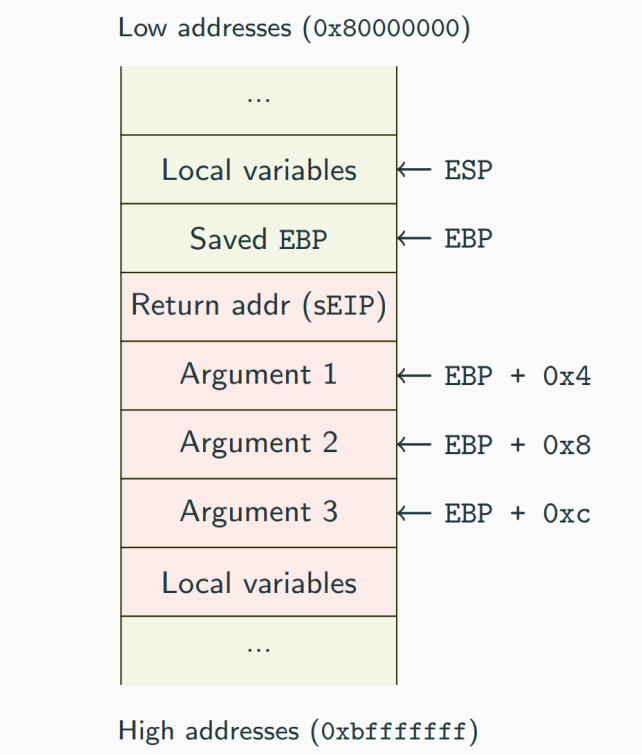
Stack and heap like always used. The stack pointer is the register `ESP`. The **stack grows towards lower addresses**.

`EIP` is an x86 register that stores the "Extended Instruction Pointer" for the stack. This register directs the computer to the next instruction to execute. Remember that we can't read or set `EIP` directly.

The concept of stack frame refers to the stack area allocated to a function: basically the ideas is that each function called has its own area on the stack dedicated to the local variables used by the function. To refers this variables we used `EBP` which is called "base pointer" since it points to the start of the function's frame.



So the `EBP` is used to access local variables easily and the local variables stored in stack frame, at lower address than `EBP` (negative offsets). Depending on the calling convention `EBP` may be used to access function arguments which are at a higher address than `EBP` (positive offsets).



Calling conventions

- Calling conventions determine the mechanism for passing parameters, either through the stack, registers, or both.
- They also define who is responsible for cleaning up the parameters.
- Additionally, they specify how values are returned from functions.
- Lastly, calling conventions determine which registers are saved by the caller and which ones are saved by the callee.
- Up to two parameters can be passed through two registers (`ECX` and `EDX`) the others are pushed to the stack.
- Return is the register `EAX`

To debug we use `gdb <name>` . We use `pwndbg` which is a GDB plug-in that makes debugging with GDB suck less, with a focus on features needed by low-level software developers, hardware hackers, reverse-engineers and exploit developers.

GitHub - pwndbg/pwndbg: Exploit Development and Reverse Engineering with GDB Made Easy

We also see IDA and Ghidra.