Computer Security

Elia Ravella

May 10, 2021

Contents

Part I
Introduction

1 Security

What is security? First, we need to distinguish *security* itself and *safety*; the first represents a system that protects from the outside, the second a system that does not harm when it's used.

1.1 What is a secure system and how to engineer one

Famous triangle problem, this time with CIA properties

- Confidentiality: only authorized entities can access information
- Integrity: only authorized entities can modify information;
- Availabilty: data must be available to all authorized entities in a determined time constraint.

"You can take only two" problem: A conflicts with C and I.
System security is an engineering problem: it's crucial to find an *optimal tradeoff* between properties to be ensured, and costs to be faced when securing such properties.

1.2 Entities in a secure system

- Vulnerability: something that allows to violate one of the CIA properties. It can be something as a design errors, a construction error, or a use defect. Even an *inherent property* of a system can be a vulnerability.
- Exploit: a specific way to use a vulnerability (or more) to violate one of CIA properties.
- Asset: values of a system, properties that must be protected. Depending on the context a system operates in, different assets can be defined (soldier - armor - life).
- Threat: **possible / potential** violation of CIA constraints. A threat is linked to a threat agent, that's the actual physical person / system that performs the potential attack.
- Attack: **intentional** use of an exploit to violate CIA properties.
- Risks: combination of assets, vulnerabilities and threats. Formally (using economic value) they're A*V*T. Risk management takes care of the first two properties. The security problem is to manage the reduction of vulnerabilities and the recovery / mitigation plan.

A vulnerability can be present without any exploits. An exploit depends from a set of vulnerabilities.

Security vs Cost Balance As usual, costs can be separated in direct and indirect costs. In this scenario, direct costs are related to the actual put-in-place of a solution. Indirect costs, instead are all the user experience costs, all those experience-heaving procedures, for example. So EVERY SINGLE SECURITY SOLUTION ADDS A COST.

Trust and Boundaries The security problem must have boundaries, otherwise it will cover over the universe. So, subparts must become *trusted* and be *assumed secure*. All the system that are beyond (below) the boundary are assumed secure too. The trusted element is the one to be tested most.

Part II
Cryptography

2 Cryptography, cryptoanalisys, cryptology

Cryptography is the building of secure mathematical systems. Cryptoanalisys is the discipline of analyzing such systems, and trying to break them. Cryptology is the union of both.
Cryptography *as a discipline* was formalized by Claude Shannon in 1949. The (still actual) formalization of a cryptosystem is composed of 3 main components:

1. the *plaintext*, the message to be encoded
2. the reversible *key*
3. the *cyphertext*, the encoded message

2.1 Confidentiality and Integrity

"Why a cryptosystem"? The two main features that a cryptosystem must have are

1. confidentiality: only the recipients can read and utilize the information in the message.
2. integrity: data can't be altered in the message passing. If so, the recipient can detect the change.

The Kerchoff's principles states that the security of a cryptosystem relies *solely* on the secrecy of the key, rather than the secrecy of the algorithm. This, in fact, should remain public. This means that a secure system is *transparent* inherently.
The key must be the trusted element of the entire cryptographic system.
Formally, a perfect cypher does not leak any information about the message if and only of

$$P(M = m \,|\, C = c) = P(M = m) \tag{1}$$

As theorized by Shannon. So "the probability of observing message m given cypher c is indipendent from the cypher given". Shannon also proved that perfect cyphers exists: they're *one time non reusable key systems, with the lenght of the key equal to the lenght of the message* (also known as the **Shannon Theorem)**.
The only perfect cypher implementation is OTP, One Time Pad. Classic XOR based encryption with a one time password (pre shared) that is long as the message. The cool thing about perfect cyphers such as OTP is that bruteforcing the cyphertext simply returns all the possible combination of plaintext, rendering so useless the brutaforce approach.

2.2 Cryptoanalysis

All the "brutaforce" fuss is about the problem of determining "how much a system is breakable". Bruteforce "always works", but it's a costly approach. So, it formally represents an *upper bound* to the decyfering provedure complexity for that cypher. Other (more smart) attacks are

- Cyphertext attack - only cyphered text are available, a key/algorithm leak of information must be exploited
- Known plaintext attack - comparation between the plain and crypted texts should lead to the key
- Chosen plaintext attack - reverse engineering of the algorithm

distinguished by the materials available to the attacker, and orderd in increasing complexity.

2.3 Remarks on Cryptosystems

1. Security of a CS is based on the robustness of the *algorithm*
2. No algorithm is invulnerable to brute force attacks, exceptionmade for *one time pads*
3. An algorithm is said to be broken if there's at least one way *faster than bruteforce* to bypass it
4. THe only way to prove a cypher is robust is to try to break it

3 Symmetric Encryption

Classic: Alice uses key K to encrypt message M into cypher C, then Bob uses the very same key K to decypher cypher C into message M. Problem: exchanging keys. We need a *trusted separated channel* to exchange the key. What is a trusted channel? It generally is a different channel that both the recipients *consider trusted* and generally *requires a different attack to be violated.*
We so must try to find a way to send to all recipents the key *in a secure way.*
Another issue with symmetric encrypted communication is scalability, each pair of user in a network should have a unique key.

3.1 How to build a secure symmetric system

Three characteristics are fundamental to ensure the robustness of a symmetric cypher:

1. *Multi Alphabetical Cyphers*: to mask structure and repetition in the plaintext several target alphabets can be (must be) used at the same time.
2. *Transposition*: also called diffusion, it consists of "swapping the value of bits", messing with the order in which the plaintext is translated.

Keyspace With keyspace is intended the *set of possible keys*. There's an exponential correlation between the size (rank) of this set and the temporal complexity of the brutaforce effort. This means that more possible keys (generally translated in "how many bits is the key made of") more the time needed to bruteforce the cypher. *This is valid in the current silicon architecture computers. Quantum computations could change this.*

4 Asymmetric Encryption

Cyphers with *two* directional keys instead of one. What is encrypted with key A can be decrypted with key B and viceversa. What you can't do is

- decrypt the key with the *same* key you used to encrypt the key
- deduce (calculate) one key from another

This approach enables "public key scenarios" due to the asymmetricity of the encryption mechanism. The robustness of this methods relies on the function used to encrypt the plaintext, that must be easy to apply *but difficult to reverse*. To make a public key scenario works we must add to the set of trusted elements (that contains the private keys) a *trusted assumption*: "only Bob knows Bob's private keys".
In asymmetric encryption the key lenght is an important measure of safeness. However, while in symmetric cypher the lenght of the key measure the number of decription attempts, in asymmetric cypher it measures the *number of key-breaking attempts*. That implies that asymmetric cyphers are "easier" to crack because of the approach chosen to encrypt. This also means that asymmetric and symmetric cyphers cannot be compared only using the key lenght.

4.1 Hash Functions

A hash function is a function that maps an *arbitrary lenght string* in input on a *fixed lenght string* at the output. Usually, input strings lenght are much longer than output ones $\Rightarrow$ *collisions*. A good hash function satisfies three properties:

1. preimage attack resistance: the function is *hard to invert*.
2. second preimage attack resistance: it must be difficult to find a input string that have the same hash to another one *given* (if the function is not preimage resistant it's not second preimage resistant too. The opposite is not true tho).
3. collision resistance: it must be hard to find 2 inputs with the same output.

4.1.1 Attacks to Hash Functions

Hash function may be broken. This means that it's possible to find the original text from the cyphertwxt *faster than bruteforcing*, just as the definition of broken cryptosystem. Two methods are used to attack a hash function:

Arbitrary Collision This is a first/second preimage attack: the attacker tries to deduce $x \mid H(x) = h$ or, respectively, $y \mid y \neq x \wedge H(x) = H(y)$.
Again, an hash function is broken if this can be completed *faster than bruteforcing it.*

Simplified Collision Attack Exactly the same approach as the arbitrary collision attack, but exploiting some probabilistic aspects of the collision (birthday paradox).

4.1.2 Digital Signature

Classic digital signature mechanism: an hash function produces a digest of a document, that's then encrypted with the private key of the sender. The receiver (to both check for identity of the sender and the integrity of the document) does the same: hashes the document and produces a digest, that's then compared with the decrypted version of the received one.
A big issue in this case is "who to trust". To estabilish a list of trusted elements the current infrastructure relies on Certification Authorities, a tree-like structure (a hierarchical group) of entities that each signs with a trusted key (a trusted role) the certificates.

Part III
Authentication

5 Identification and Authentication

What's the difference between identification and authentication? The first represents the act of "telling your identity". Authentication means to *proof* the stated identity. Authentication must be *bidirectional* or *mutual*: both entities must acknowledge each other identity and the proof.

5.1 Authentication factors

To identify someone, three methods can be used:

1. Authentication by "to know" factor (passwords, pins)
2. Authentication by "to have" factor (cards, ids)
3. Authentication by "to be" factor (fingerprint, voice)

Multi factor authentication uses more than one factor to authenticate.

5.1.1 To Know Authentication

To Know authentication systems relies on something the user willing to authenticate remembers, or is supposed to know. These systems are low cost, are easy to deploy and easy to use (due to the spread of such systems). Systems basing on to know auth are systems that rely on passwords, pins, login data combination etc.
On the other hand, the secrets the user is asked to keep can be easily stolen or guessed, or even cracked (bruteforced).

Countermeasure Against Password Wearing Three different attacks can moved against a passwrod-secured system, each one has a valid countermeasure:

1. against snooping, it's valid to change password often;
2. against cracking, it's useful to enforce complexity of the password;
3. against guessing, it's useful to ask for a nonsense password, or at least not easy to pin to the user;

A system cannot put in place all these countermeasure, because they are *costly*: they heavy the use of a system (a system that asks for a 64 mixed characters every 2 days).

Secure Password Exchange, Challenge Response Authentication phase, exchange of messages between two entities A and B that mutually authenticate each other

1. A to B: This is my identity. Let me authenticate
2. B to A: nice. Compute hash(random-data + secret) + random-data
3. A to B: hash(random-data + secret)
4. A to B: nice. Compute hash(random-shit + secret) + random-shit
5. B to A: hash(random-data + secret + random-shit)

Where secret is the password, generally. The supposition is that both entities *know* the secret.

Secure Password Storage Passwords are the secret upon it's based the security of a system (an authenticating one, at least). How to enforce password security?
To protect a file (data in general) the techniques are always the same:

- Encyption
- Salting (modifying stored passwords to mitigate dictionary attack)
- Access Control
- Password Recovery procedures that not disclose sensitive infos

5.1.2 To Have Authentication

To Have factor relies on *a phisycal device / object* that must be associated to a person willing to authenticate. The problem remains the human factor: this time the security of the system depends on how the key (secret) is handled by the person associated to it.
In any case, to have authentication system are low cost and offer a good level of security. On the other hand, they're hard to deploy, and the tokens / cards are easy to be lost or destroyed / compromised.

5.1.3 To Be Authentication

To be authentication asks the person willing to authenticate to prove to be who s/he declares to be by biometrically satisfy a costraint (like having a certain fingerprint, hand geometry, iris pattern...).
Although this kind of authentication is very effective (it's *really* difficult to emulate someone else's fingerprint) the list of disadvantages for biometric security systems is quite long:

1. They're hard to deploy and set up. Moreover, usually during the setup phase an invasive test must be performed
2. The matching of the feature selected is non-deterministic: accepting or rejecting a person depends on a threshold of accuracy, and this can act differently from time to time
3. They're not *entirely* secure: a fingerprint can be cloned
4. Biometric features *change* over time
5. Privacy issues: people *may not want* to distribute personal features involved in a biometric security system

5.1.4 Single Sign On

Single sign on was proposed as a system to countermeasure to password reusing. This system is based on a 1-2 factors authentication and *a single trusted host*. This is simply the "delegating authentication to someone else" method of authentication, the *external auth* method. Two main approaches:

- SAML approach: an external service serves as authentication factor
- OAuth approach: the external service *just helps the authentication*, giving the user a code in order to sign on to the principal service

This is also complicated to implement: an app should (in order to implement OAuth or SAML) complicate a lot its flow of usage (user experience) in order to just authenticate a user.
Moreover, the trusted factor is a SPOF.

6 Access Control

Access control is all the protocols and procedures that *allow or deny* certain operations to be applied on certain resources. This operation is usually performed by a reference monitor, that carries out the verification of the rules and the roles (lol).

6.1 Discretionary Access Control

The resource owner assigns the priviledges to users. It's the most standard access control system: for every user (or entity) a set of permissions is defined that describe the rights of each one. A DAC is defined by 3 set of entities:

1. subjects: the actors, the users
2. objects: the resources
3. actions: the actions that subjects can perform on objects, that are restricted by the rules

The reference monitor organizes the subjects and the objects in a two entry map that lists what every user can do to each resource. This implementation is called HRU model, due to its designers. What this model also allow is to check if the access control system works as intended: if it can be proven that a user can "upgrade" his permissions *on his own* the system is unsafe by design. This is an undecidable problem btw.
This implementation (let alone the scaling problem) is a little bit difficult to utilize, so alternative implementations are possible, like:

- Authorization tables: DBMS like approach, it's a HRU model with not null entries
- Access control lists: used in networking, each objects records his own owners/users
- Capability list: each user "remembers" the authorization he got

See how the last two implementations are just "read the HRU table coloumn by coloumn" and "read the HRU table row by row".

6.2 Mandatory Access Control

Clearance / Sensitivity access control management. Based on user's roles combined with "objects' secrecy levels". The idea behind is: do not let users assign priviledges. The priviledges are set by an administrator, that assigns levels to users and to resources. Each user level can access only a subset of the resource level. This is usually combined with a labeling system that categorizes in a finer way the resources, orthogonally to the secrecy level.
Then, read/write accesses are granted as rules, for example

- No read up
- No write down

6.3 Role Based Access Control

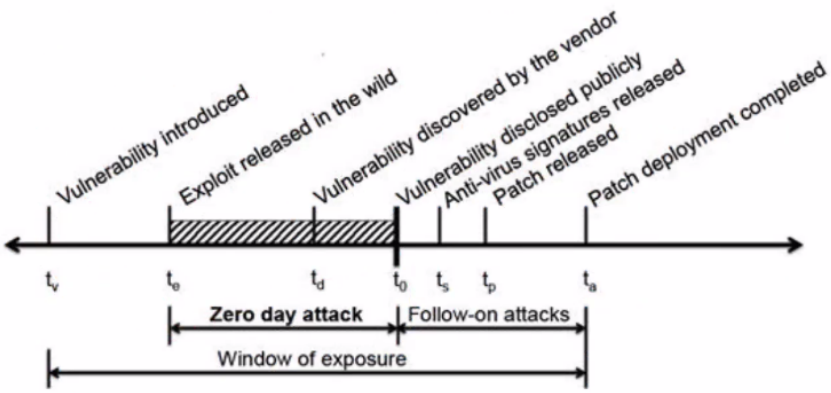
RBAC is kinda a hybrid of DAC and MAC, and can also enforce them: if configured in the right way, it can act as a DAC or as a MAC. The idea is similar to the one that driven MAC design ("no self assigning dynamic priviledges") but the roles are no more structured in a rigid hierarchy but instead are simply *a label to a set of priviledges.*

Part IV
Software Security

7 What is software security?

With "software security" is intended the meeting of the non-functional requirement (obviously for a software) of *being secure*. This is often overlooked in the process of software developing, but to remind the importance of this non functional requirement, it's sufficient to think about the name of the unmet security specifications: **vulnerability**.
Recap on definitions:

- Vulnerability: fault in the target software/system
- Exploit: method to leverage a vulnerability to harm a system



Vulnerability / Attack relationship This graph explains how the vulnerability life cycle interacts with the actual attacks cycle to a software. To be noted:

- zero days attacks happens when the vulnerability is "out in the wild", known only to hackers
- at the moment the vulnerability is *exposed to the public* other security firms can start working on side solutions to limit attacks, weakening the actual attackers margin

- the window of exposure is closed *ONLY* after the patch is installed *everywhere*

8 Principles of Secure Design

We can define a list of principles, guidelines, to create a secure software:

- Reduce the privileged parts to the minimum to avoid permissions vulnerability windows
- KISS principle, reduce the complexity of a software
- Avoid shared resources, or unsafe external libraries. Use instead safe and known secure libraries
- Default deny principle: allow only authorized operations
- Use filters on input and output

9 Vulnerabilities

9.1 Buffer Overflow

Memory vulnerability, one of the most famous technique to exploit a software. Practically, it means that a piece of data is inserted in a buffer that is smaller. This enables the attacker to overwrite "sensitive" portion of the memory, and (in the worst case) inject malicious code. The thing that makes buffer overflow so dangerous is the way a buffer is allocated in 99% of cases, so calling a function that does that: this allows the attacker to *directly overwrite the return address from that function.*

9.1.1 Buffer Overflow HowTo

1. Find the right address
2. Create the shellcode
3. Insert a nop sled

9.1.2 Buffer Overflow Defense

Defending against buffer overflow means defending against a super powerful type of vulnerabilities that can harm a whole software. This kind of defense can be put in place in three main regions:

Source Code Level The source code is where the vulnerabilities is rooted. Making the source code safe from potential buffer overflow means *educating developers* and *using safe libraries* or safe version of already written libraries. The technology stack should be secure too!

Compiler Level The compiler can inform the developer of the possible vulnerabilities, butthis still rely on the human. Another approach is to change the order (randomizing) of the arguments onto the stack. *BUT* this is only temporary, as the buffer overflow is still there and can be found by exploiters by analyzing the output of the compiler, the binaries.
The "canary mechanism" is also used: a spy is put on top of the stack (the canary is put by the prologue of the function) and then verified by the epilogue. If the canary is different, then the function has been overflowed, and the program has been compromised. Multiple approaches can be applied to canaries:

- Random-per-run canaries: each run of the program the canary is changed
- XORandom canaries: the value of the canary is obtained *from the data it's trying to protect*, preventing further unwanted memory modification
- Terminator canaries: the canary is entirely made up of terminator characters (so cannot be stringcopied)

OS Level The OS has two main mitigation that can use:

- Non executable stack, so flagging some portion of the memory as data-only. This does not mitigate the "return to a function" type of attacks.
- Address space layout randomization: scrambling the memory pages layout. This renders quasi-impossible guessing rightly the return address.

9.2 Format String

Format string vulnerabilities creep out near printing function, when these are "badly formatted": no format (or a wrong one) is provided to print the parameters. Classical example: passing a placeholders-only string to a printf without formatting will print the content of the memory filling that placeholder, *disclosing information.*

9.2.1 Exploiting Format String Bug in a ANSI C Program

We have several placeholders and placeholders extensions we can use to make a format string vulnerability super dangerous.
Placeholders:

1. %d and %i to print integers
2. %u to print unsigned decimals
3. %X and %x to print unsigned hexadecimals
4. %o for octals
5. %c for chars, %s for strings

Placeholder modifiers (of interest):

1. N$: if inserted between the % and the placeholder character tells the formatter to go to the N-th parameter.

2. %n: this placeholder *writes the number of chars printed* so far to the address pointed *by the argument*. The important thing is: whatever value is in the memory *where the argument is searched*, it will be interpeted as a memory location. We will also use his brother %hn, that writes a shortint.

3. also, the %c placeholders for chars can be extended as %Nc where N is the number of times the first char argument must be prented.

So, how do we exploit this?
First of all, we can build a "memory scanner" with just "%N$x" and iterating over N: this will print the hexadecimal code of all the "arguments" (memory slots) after the one passed as an argument. Remember, we're sure to be on the stack of the application, because this bug targets *printing functions*, and the exploit string will be the parameter of such function. This is already a vulnerability: we can exploit this to make tha pplication leak information by *directly accessing its memory space*.
To also write, obviously we need to use %n. We must combine the character-generating placeholder %Nc and the target-sniping placeholder N$ in order to obtain the right alignement of the memory and to write *exactly there*. This also highlights the main difference between a buffer overflow vulnerabilty and a format string one: while the BOF acts as a tank overwriting all the memory it traveses, the format string modify exactly the portion of memory needed, leaving the rest unchanged.

Format String in Practice What we need:

- A target address, the one that represents the block of memory *to be over-written*;
- An arbitrary piece of data (could be another address, an arbitrary number, a set of flags...) *that we want to write in memory*

What do we need to do:

1. suppose we already know the target address of the memory portion to tamper, we need to put this address *exactly* in the position that the %n will point. To do so, we use the memory-scanner we've presented previously, to finde the offset of it. For now, our string is composed only by the target address.

2. at this point we need to introduce in our malicious string the data we want to write. To do so, we concatenate the target address with the malicious data generated by the %Nc placeholder. We now have, as exploit string, "hex of target address + % (malicious data in decimal - address dimension) c".

3. lastly, we need to combine the previous step with the writing placeholder %n. Where do we write (so, which value do I insert in the N$ parameter of the placeholder)? We need to write in the target address
   ⇒ we have put the target address on the stack
   ⇒ we have discovered the offset of such address by scanning the memory
   ⇒ we put that offset here.

Our final string is something resembling

$$\begin{cases} \text{target address } TA \\ \text{malicious data } MD \\ \text{offset of the target address } OFF \end{cases} \Rightarrow TA\,\%MD\,c\%\,OFF\,\$n \qquad (2)$$

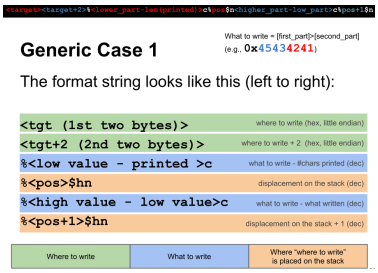
So a valid exploit could be "\xcc\xff\x86\x92%420c3n": it writes "420" in the \xcc\xff\x86\x92 memory cell, and the address is memorized as third parameter after the function.

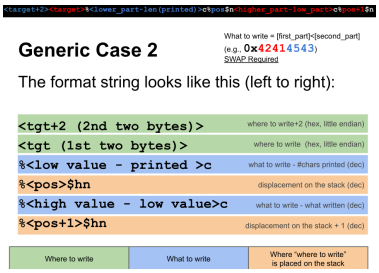
Writing Multiple Locations What if, for example, we want to write into memory (as the malicious data) a valid address, and this address occupies more than one word? Can we write *multiple times* with the same exploit? Yes sir we can. This procedure involves a little bit of additional math, but it's pretty straightforward. First of all, the abstract of the metod: *we use two times th %N$n writing mechanism putting two consecutive addresses on the memory, as parameters for N$.* Whats the problem here? The problem here is that the second write is bounded to go *over* (address speaking) the first one. So, we need to split our malicious data in two part, and write first the "litle" (in module) part.
More difficult said than done, trust me. It's very intuitive: the %Nc placeholder can elongate the string of N characters, then (if we repeat the mechanism afterwards, %Mc) add another M characters. Oviously, M cannot be less than zero, so we need to write *first* the "lower" (in absolute value) part of the malicious data and then the "higher" (as remaining part).
The final exploit turns out to be something like

$$\begin{cases} \text{target addresses } TA,\ TA+1 \\ \text{malicious data } MD \\ \text{offset of the target address } OFF,\ OFF+1 \end{cases} \qquad (3)$$

$$(TA)(TA+1)\,\%low|\,MD\,|\,c\%(OFF)\$n\%(high-low)|\,MD|\,c\%(OFF+1)\$n \qquad (4)$$

Quick Note on this Part So I think I was not totally accurate in writing these notes, so I'll put the professor recap from *his slides* right here in the notes.





9.3 Final Remarks on Vulnerabilities

The two vulnerabilities we've seen (format string and buffer overflow) are two common memory vulnerabilities that can be exploited to manipulate the be-

haviour of a system, tricking it into leaking information or worse, like executing malicious piece of software. These kind of vulnerabilities all originate from a "irresponsible" programmer that does not have in mind how a software can be exploited; however, against these vulnerabilities several countermeasures can be taken and put in place, and many of those eventually finish embedded in operating systems. What's more important is to consider safety a semi-functional requirements for software.

Part V
Web Security

10 What is Web Security?

The main difference between a "traditional" software attack and a web attack is often only the exposure of the app itself. This is also the motivation that renders the web attacks so impactful and web vulnerabilities so crucial to solve.

10.1 Architectural View of an Attack

Let's talk about the classic 3 tiered architecture:

1. Client and presentation layer
2. Controller layer
3. Persistence / Data layer

HTTP connects the first two layers (and often also the two back ones). The statelessness of HTTP is also the major securty issue: it provides *determinism* in the behaviour of the web applications. Also, it's text based; easy to counterfeit.

10.1.1 The Untrustworthy Client

The client is the only piece of software that the attacker can use, most of the times. The golden rule for web programming is to assume that *all clients are attackers*, and thus give that part of the software the minimum power[1] possible.

11 Filtering

Filtering is the first line of defence wrt any kind of attack. It's obvious: if an attacker *cannot add malicious stuff* or *cannot even access dangerous services* it cannot harm. BUT filtering is hard: what do you filter? What do you blacklist? This is the classic filter system build:

1. Start with a whitelist approach (deny all except allowed)

[1] With *power* in this case is meant "possibility to harm". This usually translates in limiting as much as possible the client actions and doublecheck server side all the data that comes from them.

2. Proceed "a regime" with a blacklist method (deny bad people)

3. Perform *escaping*, so neutralize possible malicious inputs. This procedure is often referred to as "hygienizing input"

Filtering also comes into play when it's time to separate the data from the executables: when the data can be executed, when it's just data?

11.1 Filtering Problems: Cross Site Scripting

Cross Site Scripting (XSS) is a kind of attack that just occurs when *data is not differentiated enough from code*. MSN allowed you to enter HTML an JS code into comments, but a JS-written comment will *executed by all the people that visit that page*. We can distinguish three type of XSS: Stored, Reflected and DOM-based XSSs.

Stored XSS The MSN example. The attacker can memorize malicious code in the server and this is then propagated to other clients. This happens if the malicious code is not recognized as such; all the clients that download the malicious resource find valid code to be executed, and blindly execute it.

Reflected XSS Reflected XSS does not rely on the server *to memorize the exploit*: the attacker usually craft a malicious packet (or request) and sends it directly to the victim. This request will *trigger an existing vulnerability* in a site (that can also be totally unaware of it). The classic attack of this kind exploits a site that *does not hygienize the content it receives through a GET* and presents it as-is to the user; for the attacker it's enough to craft an ad-hoc request and have the user trigger it (mail attack).

DOM based XSS DOM based attacks are similar to reflected attacks (in fact, the idea is the same; often also the implementation). The main difference is that the server is completely cut off from the attack: the reflection of the malicious code is handled by the *client side code* such as javascript scripts built in pages. The target page (the one that in classical reflected attacks actually *hosts a vulnerability*) now can be totally embedded in the request. These kind of attacks are just pumped up reflected XSS attacks.

XSS is Powerful JavaScript and other web scripting languages used in dynamic pages are built and executed with the client golden rule in mind: *do not trust clients*. This does not mean that they cannot harm: they have access to personal information such as cookies, the session storage (and often the LocalStorage too) and they can perform **drive by downloads**[2].

11.2 Same Origin / Content Security Policy

Same Origin Policy SOP is a principle implemented in all web clients, that enforces the "don't touch other's stuff" rule: all client side code loaded from origin A can access only to resources loaded from origin A. This simple policy

[2] just issuing a download "fraudolently", that sends malicious code to the client.

can be bypassed in several ways, and the modern web programming paradigms are making circumventing this rule easier to attackers: it's not so rare that more sites share the local resources (in order to offer enhanced services) or that client side extensions inibits some of the security countermeasures of the browser itself.

Content Security Policy CSP, instead, is a measure adopted *by websites* that aims to mitigate XSS and other code injection attack by *informing the web browser which are the secure contents* that can be trusted[3] on that page. It's embedded in the HTTP headers of the response. The CSP is implemented by the majority of commercial browsers. CSP is a great mitigation for attacks, but:

1. writing the policies must be done by hand, and it's difficult
2. policies must be kept up to date
3. how many policies are enough? how many break functionalities (for example getting in the way of the loading of a library)?
4. how does CSP realate with browser extension? (same problem of SOP)

12 Vulnerabilities and Attacks

12.1 SQL Injection

Code injection attacks works all at the same way: they add (and then try to execute, or havo other execute it) malicious instructions in a place where these should not be, in order to harm a system. SQL injection is no different: the target this time are not-sanitized input fields in web forms,

12.1.1 Exploiting a Non Sanitized Form

SQL injection is quite easy to understand: take for example a login form, for a newsletter, social network or service that is. The two basic fields to fill are username and password, and will be likely be inserted in a "users" table. *If there's no filter to the input*, I can insert something like " username ';– " in the username field. This will

1. Close the string with the character '
2. Close the query with the character ;
3. comment the rest of the query with –

If the system is simple enough to allow users in the system as soon as this query returns a value then I've successfully logged in withough a password!
Different attack can be performed: instead of commenting, I can inject an always-true condition in order to bypass the password checking (such as OR 1=1) or completely bypass the query system with a UNION attack: some DBMS actually consent to use set operators as UNION, INTERSECTION to filter query results, and this can be exploited by attackers to validate false information.

[3] so, loaded and executed

12.2 URL Tampering

Another surface that's often vulnerable to attacks is the URL of the app we're using. For most web applications the URL is also a string the *encodes the state of the application*, maybe even specifying the exact location (on the **server**) of the document displayed (this is valid for old directory-structured sites, but it's also true for complex modern applications with explicit parameters in the URL).

12.2.1 Path Traversal Attack

For old fashioned sites that are just HTML documents collection, the URL is the path on the server that identifies the desired page/document displayed. Again, if *no filter* is present on the URL, nobody can stop an attacker to navigate through the filesystem of the server just by tampering the URL.

Part VI
Exercises and Examples

13 Exercises

13.1 Identity theft

Which are the risk components in the scenario of identity theft, like in social media platforms?

1. Reputation threathened both of the victim and the company
2. Social engineering attacks
3. Malicious spread of information

And the assets?

1. reputation
2. money
3. personal information

And who are the threat agents?

1. People around you
2. Government
3. etc

So *everybody* with a malicious intent
Why is this scenario possible, and how to mitigate it?

1. 2FA
2. trusted device based auth
3. enforce credentials strongness

13.2 Ransomware attack

A small company specialized in a particular object is infected by a ransomware
What are the threats, the asset at risk and the possible countermeasures?
First:

- data loss
- data of the company
- offline backups

Second:

- data leakage
- data of the company
- encryption

Third:

- denial of service
- production
- redundancy

Who are the possible threat agents? Cuncurrent companies, ex internal employees, ransomers...

13.3 Auto driving vehicles

Consider auto driving cars used as taxies.
Which are the most valuable assets in this scenario?

1. People inside the car
2. People outside the car
3. the vehicle itself

What are the attack surfaces on the vehicles?

1. The VEHICLE itself, you enter there!
2. the connectivity capabilities of the vehicle
3. the company network, or the internet itself (it was an hypothesis)

Part VII
x86 assembly recap and pwndbg

14 Recap of software compiling

Code is translated by the compiler in Assembly code that is the assembled in binary code. That's the workflow. The process is theoretically not reversible, it can be reconduced to the halting problem. This is due to the loss of information (for example about types in the first step) that the process introduces.

15 The x86 ISA

64 bit CISC architecture for original Intel processors.
Register file:

1. EAX EBX ECX EDX
2. ESI EDI for strings
3. EBP ESP base and stack pointers
4. EIP instruction pointer, never explicitly accessed
5. EFLAGS program status and control

The "E" before the register name stands for "extended" and references the 64 bit version of the register. Omitting it will access the 32 bit version.

15.1 Data types

Four main "data types"

1. Byte - 8 bit
2. Word - 2 Bytes
3. Dword - 2 Words
4. Qword - 4 Words

15.2 Instructions

Instructions in x86 have *variable lenght* ranging from 1 byte to 16 bytes. Every instruction is composed of the opcode (representing the operation) and the operand list. We'll be using logic, memory, arithmetic and control flow instructions.

16 Executables and binaries

Binary files, although different for each operative system, all (more or less) follow a fixed schema to encode operations and data. Usually, this schema involves a DATA and a TEXT section, respectively for the data and the instructions; it usually has also a section where offsets in memory (to describe the memory structure *of the program*) are stored. Usually (starting from the lowest and going to the highest memory addresses) this is the schema:

1. Shared libraries
2. .text
3. .bss
4. the heap (that grows *towards high addresses*)
5. the stack (that grows *towards low addresses*)

6. *env*
7. *argv*