Computer Security

Elia Ravella

June 12, 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. We should always think about which components are damaged in the case of a successful attack: can be
  - the users of a system
  - the company that produces the system
  - the physical environment the system is in
  - also intangible assets, like the *reputation* of the users/owner of the system
- 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 encoding function, which usually depends on a *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. The algorithm itself, 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 cyphertext 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 deciphering 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 decipher 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. *Substitution*: replacing bytes. Ceasar's cypher is the perfect example of only-transposing symmetric cypher.
2. *Transposition*: also called diffusion, it consists of "swapping the value of bits", messing with the order in which the plaintext is translated.
3. *Multi Alphabetical Cyphers*: to mask structure and repetition in the plaintext several target alphabets can be (must be) used at the same time.

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 ***only*** 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 brute-forcing 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

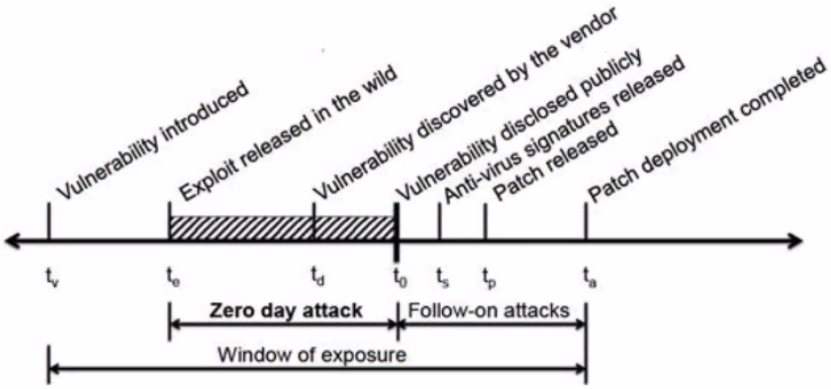
RBAC is kinda a hybrid of DAC and MAC, and can also enforce them: if configured in the right way, it can act as a DAC or as a MAC. The idea is similar to the one that driven MAC design ("no self assigning dynamic priviledges") but the roles are no more structured in a rigid hierarchy but instead are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

9.2.2 Format String Defence

Being another memory vulnerability, many of the solutions explained for the buffer overflow are still valid. Being the format string dependant strictly on some printing functions, it can be easily patched by modifying this very functions (indeed, a secure version of libc exists). Moreover, compilers have learnt to detect "unsafe calls" to printing functions and signal them.

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 behaviour 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)
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?

[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.

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 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.

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

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.

12.3 Cookies!

Cookies were born to help web sites build a user experience that was tailored to the *single client*. The abuse of the cookies (as nowadays web) renders them a weapon of mass surveillance: every action can be stored in a cookie on the client and then accessed to model the client behaviour.

12.3.1 Cookies for Session Management

Cookies are used to store information about the HTTP *session*, a technique used by websites to keep track of the state of an interaction with a client. For example, to keep conversational data about a client, the server can generate a SessionID that than stores as a cookie on the client to identify it, and the data associated to it.
OBVIOUSLY these session tokens should be crafted in a way to not be exploitable:

- if used for authentication, not storing plaintext sensible data
- if used for session identification they should not be "predictable"
- they should have a clear (and near) expiration date that ALSO should not coincide with the cookie one

All these things must be kept in mind when designing session interaction in order to avoid impersonating attacks: if an attacker is able to spoof sensitive infos from the cookies he can easily impersonate someone else.

12.3.2 Cross Site Request Forgery

Forces an user to execute unwanted actions (state-changing action) on a web application in which he or she is currently authenticated with ambient credentials (e.g., with cookies).
So, this formal definition states that a CSRF exploits the *client ambient credentials* to fake a set of action to perform on a site.
The classic CSRF attack can be carried out in 4 steps:

1. The victim signs in on a website that (for example) takes information from the client through a form that *exploits ambient variables*.

2. The victim visits a malicious site (or clicks on a link)

3. The malicious site generates a fake request (following the form example, a well-crafted POST HTTP request is enough) and with XSS techniques make it execute on the victim web client

4. This way, the web client executes a malicious web request that *fills out the original form* and it's able to get through because the original website uses the *ambient variables* to authenticate the user

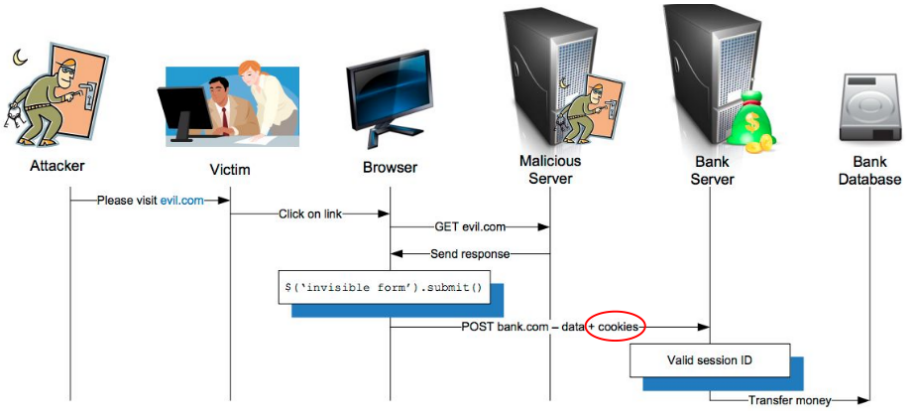


Figure 1: Sequence diagram for a CSRF attack.

CSRF Mitigation Techniques A first form of mitigation for this kinds of attacks is the CSRF token: a set of *random challenge* tokens that are *associated to the user's session* that are regenerated at each request. These tokens are sent from the client to the server (but are not stored in the cookies) and the requests are "confirmed" only if the tokens (client-sent and server stored) match. This technique is called *synchronizer token pattern*.

Another possible mitigation is the "Same Site Cookies" policy: from Wikipedia, "An additional 'SameSite' attribute can be included when the server sets a cookie, instructing the browser on whether to attach the cookie to cross-site requests. If this attribute is set to 'strict', then the cookie will only be sent on same-site requests, making CSRF ineffective. However, this requires the browser to recognise and correctly implement the attribute, and also requires the cookie to have the 'Secure' flag."

Part VI
Transport Layer Vulnerabilities and Attacks

13 Security at Network Layer

In the internetworking scenario, the attacks are usually put in place targetting design issues of communication protocols, as UDP or TCP. These protocols are not "theoretically" vulnerable, but in order to work they expose information that can be used in a malicious way. Three kinds of attacks are studied in literature:

- Denial of Service (attack to availability)
- Sniffing (attack to confidentiality)
- Spoofing (attack to integrity and authenticity)

14 Denial of Service Attacks

DoS attacks threaten the availability of an app or service. This can be done in multiple ways, depending on the level that is being attacked (the attacker can deny a service directly on the user's client, deny the access to the communication infrastructure that links client and service or directly making the service unavailable) or the kind of attack carried out.

14.1 Killer Packets

14.1.1 Ping o Death

Killer packets attacks leverages errors is the *protocol implementation* on certain OSs. The classic attack of this kind is the so called "ping of death": a pathological ICMP packet (pathological because "too large") that overflows the receiver buffer, causing it to crash.

14.1.2 Teardrop

Teardrop attacks exploits a vulnerability hidden in TCP packets, the possibility to specify the offset of the data segment of a fragmented packet[4] in an inchoerent way, like overlapping packets. This makes impossible to reconstruct the original long packet, making the kernel hang or crash.

14.1.3 Land Attack

Little bit of history of hacking: Win95 was vulnerable to a particular type of attack that simply consists in crafting an IP packet with the source address equal to the destination address.

[4] a TCP feature that enables splitting a network packet into multiple packets

14.2 Flooding

Flooding a network means saturate it with requests, above the bandwidth that the infrastructure can hold and manage.

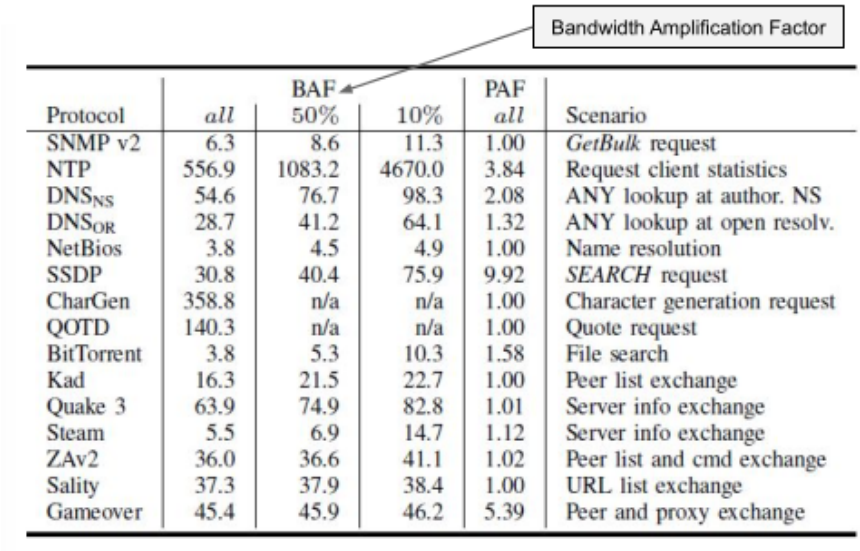
14.2.1 SYN flood

SYN flood attacks exploits the way a three-way-handshake (the TCP method for estabilishing a connection) is designed. The server *must hold the information* about all clients he's waiting an ack from. So, an attacker can saturate the server by never sending these. The attacker must use different IP addresses to do this (easy enough: a handful of seconds of spoofing gives you the most frequent IP addresses used on a network). This can be easily mitigated by moving the actor that holds the SYN information (storing a SYNcookie, of sorts).

14.3 Distributed Denial of Service

Oppositely to the DoS classic attacks, DDoS attacks exploits a big infrastructure (such as a botnet) to saturate a network/infrastructure/service. The idea remains basically the same as the flooding attacks, but this time the "power factor" is the dimension of the attacker network instead of the use of the resources of the server. Notice how these attacks are *far more difficult to mitigate* because if the hacker can put together a large enough attacking infrastructure there's no way to stop it.
The attacker can either use a network of computers at his disposal (botnets or C&C infrastructures) or leverage an existing network, as the Smurf attack:

Smurf Attack If the attacker can spoof the victim address (whichever address it may be) it can use it on a network for simply issuing an echo request with at "sender" the spoofed victim address. All the networks echo replies will be redirected to the victims, flooding it. This is a table that calculates the so called *amplification factor* for each protocol.



Protocol	BAF *all*	BAF 50%	BAF 10%	PAF *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

Figure 2: The bandwidth amplification factor (BAF) computes the bandwidth multiplier in terms of number of UDP payload bytes that an amplifier sends to answer a request, compared to the number of UDP payload bytes of the request.

15 Sniffing

Sniffing is more a "hearing what's not destinated to you" kind of attack. It's a passive kind of attacks that aims only to retrieve information.

16 Spoofing

16.1 ARP Spoofing

ARP is the protocol that converts IP addresses into lower level adddresses. To be fast (it's deisgned to be only fast) it's unauthenticated, with belief that authentication is done at higher levels. ARP just provides a request response broadcast service in a LAN to do this linking of addresses. ARP spoofing is rather easy: if an attacker detect an ARP request he can send a malicious ARP response in order to *deviate traffic* to another direction, or just interrupt it. This kind of attacks are usually called "ARP poison" attacks.

16.2 IP Spoofing

The IP source address is not authenticated. This render the IP protocol extremely vulnerable to spoofing attacks. An attacker can spoof an address and use it as the source address in an UDP or ICMP packet. However, he won't be able to receive the responses! He can still sniff them tho...
For TCP protocol (due to the way the sequencing and the handshake are organized) things change. If the attacker can guess the InitialSequence Number

he can *blindly complete* a three way handshake, but here's the difficult part: guessing the semi random number chosen by the victim. This is quite difficult now but TCP sequence numbers were chosen with a very low randomness back in the 90s.
Where's the danger? *Man in the middle* attack. A fully impersonating attack, the hacker can fake himself to be someone else and it's very difficult to tell the other way!

16.3 DNS Poisoning

DNS poisoning attacks insert malicious name entries in the DNS server of a network, exploiting the non-authenticated update protocol of DNS. This is done by impersonating the higher level DNS server. This can be done to deviate traffic on a network.

16.4 Attacks against DHCP

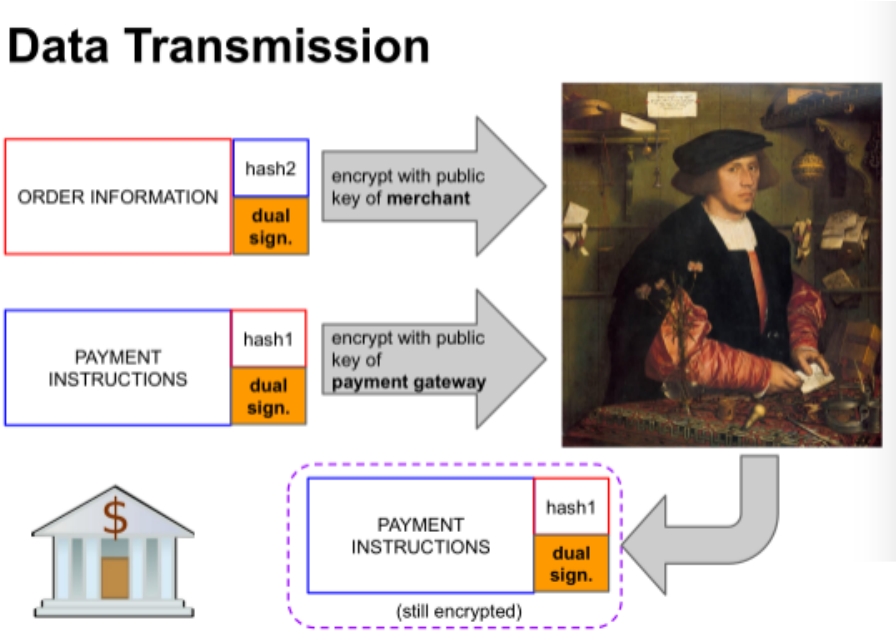
DHCP is the classic example of an unauthenticated protocol. An attacker can "easily" impersonate the DHCP server (by sniffing a DHCP request packet) and modify the IP address, the DNS and the default gateway of the victim.

17 Transaction Security: SSL, TLS and SET

Transactions are a mess. Securing a transaction is a badder mess. There's a trust issue (maybe multiple), a sensitive data issue, a timing issue, an atomicity issue, an authentication issue...
Two solutions exists, one more adopted than the other:

17.0.1 SET

Secure Electronic Transaction is a protocol enforced by VISA and MasterCard to make the online transactions secure. The underlying idea is the "dual signature": a signature that combines two pieces of a message directed to two different recipients. The architecture can be described as:



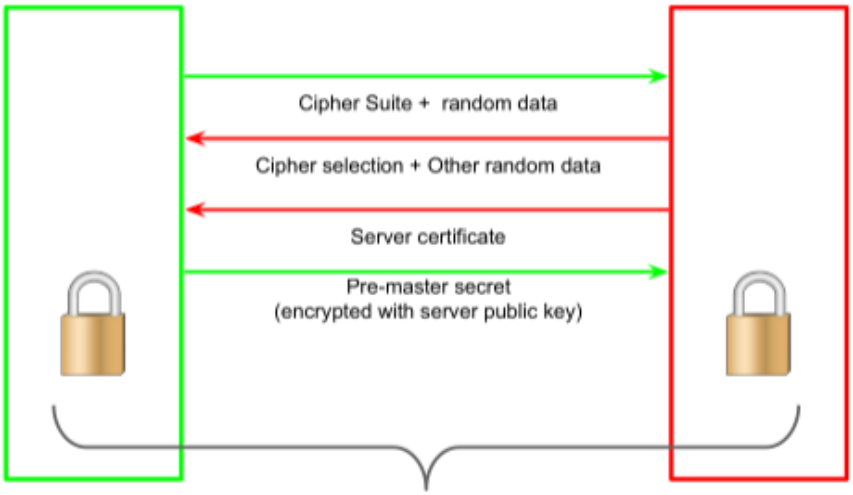
SET failed because the implementation was very difficult to put in place, and also not fully transaprent. The main problem was the requirement of a digital certificate *also for the client*: let's say it's not the most scalable solution.

17.0.2 HTTPS

HTTP over Secure Software Level is the de facto standard for the secured internet communication. SSL is the name for the previous versions of TLS (all TLS version behind 1.2 are deprecated at this time).
TLS (the actual securing protocol) redesigns the classic HTTP handshake, adding the "secure suit selection" part: this first part is issued by the client, that at first connections *tells the server which are his preferences for*

- key exchange algorithm
- bulk encryption algorithm
- message authetication code algorithm
- random number generation function used

At every voice in the cipher suit is associated a preference level, and the server sends back the configuration of algorithms to use for the communication basing his decision on this preference level. In the server response it also sends his certificates. *TLS also supports client authentication*, so it allows the client to reply with his own certificate set. The handshake can be summarized as so:



Compute shared secret from pre-master secret, client random data and server random data

Man in the Middle Attacks TLS is designed to be resistant to MitM attacks: what an attacker can do to impersonate the server is cut out from the handshake

- He cannot let the original server certificate go through, because he won't be able to be trusted
- He cannot send a fake certificate, the client can easily notice

Part VII
Secure Network Architecture

18 Firewalls

Firewalls are network access control systems that verify all the packets flowing through them. Easy. This verification can be don at IP level (like Access Control Lists) or at transport layer (NAT level, to be precise).

Firewall Cons Firewalls are perfect to filter traffic, but firewall positioning is a problem *by itself*: a FW is totally useless if an attacker can go around it. Also, the firewall is usually a computer by itself: they're vulnerable by itself and can be cracked, put offline etc. Moreover, the configurability of the firewall renders it a victim of bad configuration issues: if we're adding the wrong rules we're restricting too much OR not restricting enough the network we're rying to protect.

18.1 Firewall Taxonomy

Network Layer Firewall We can divide network layer firewalls in

- ACL. They just filter packets basing on static rules on the address or port, for example. ACL are very limited when it comes to detect attack that spans over multiple packets: they're old. They can perform only a static analysis that spans in detail up to the IP options enabled, and not fully even to TCP or UDP.
- Stateful packet filter firewalls. This kind of packet filters are an evolution of ACL, with added state: a stateful packet filter knows if a packet is in some way *related* with the previous or the next one. These firewalls are much more powerful wrt ACL: the ability to detect exploits *across packets* enables them to track connections and validate them. Also, *there's almost no need for reflexive rules*[5], because the state encapsulates it. Firewalls of this kind would alterthe ISN of TCP sessions in order to make more difficult to exploit a TCP packet for an attack, an will set "tolerance timers" on top of stateless protols (UDP) that simulates the "acceptance window" for a safe packet.

Application Layer Firewall At application level we can encounter

- Circuit level firewalls (legacy stuff). They act as a TCP level proxy.
- Application proxies: circuit level firewalls, but on top of the stack. They're not transparent to the client: the connection should pass through the proxy every time, to inspect/verify/sanitize it.

[5] response rules

19 Architectures for Secure Networks

19.1 Multi Zone Architectures

"Multi zone architectures" is the boring name for the *castle approach* to network design; this relies on the assumption that all that's inside is good, all that's outside is harmful. However, the limit of this approach are evident: how can I safely allow external access to my network? Well, we can simply *sectorize our internal citadel* into partitions that differs in the *level of clearance needed to access*.
The classic multi zone architecture is divided in:

1. the internet
2. a tunnel of DMZs (demilitarized zone) that's an intermediate portion of the network: this partition is *reachable from the public* and so all the machines in this zone *should not be trusted*. We can put several of them "in series" between th einside and the outside
3. an internal private network (that can be furtherly divided)

ALL THESE LEVEL ARE SEPARATED BY DIFFERENT FIREWALLS that are configured in different ways.

19.2 Virtual Private Network

VPNs are a threath to pure castle-approach structured networks. They in fact open a tunnel through them.

Full Tunnelling ALL traffic generated by a VPN connected host pass through the local area network of the organization, allowing him to benefit of the internal resources (among the others: the *firewall*).

Split Tunnelling "Local" traffic is redirected inside the organization's LAN, while the Internet traffic is left to the ISP to carry out. This actually open a door on a LAN. This modality is the most dangerous to a network that relies on a castled approach.

Part VIII
Malicious Software

20 Malwares

What are malwares? Mal(icious soft)wares are *programs that are explicitly written to violate a security policy* or more in general, *harm a system.*

20.1 Categories of Malwares

Malwares can have several features and objectives. We can classify them basing on these traits:

- Viruses: self propagating piece of code
- Worms: self propagating *program*
- Trojans: virus ”in disguise”

Computer viruses were first created as a proof of concept for programs that auto replicate or modifies on their own. The security problem was understood later. An interesting theoretica result is the formal impossibility to build a perfect virus detector: this problem can be seen as a formulation of the halting problem (still undecible while I'm writing).

21 Lifecycle of a Malware

The life of a malware is divided in 4 main stages:

Reproduce and Infect In these first two phases the malware is reproducing the logic under it and must not be detected. The tradeoff here is infection speed (or broad) vs detection probability. Nowadays, the spreading vector is something not necessarily hidden in the code, but more a known vulnerability or a social engineering attack. Several infection techniques are used: some viruses hides in the master boot record of computers or in the boot partitions. Or simply modify neighbouring files (like multicavity viruses, that inject code in unused region of program code).
Another method is hiding in a macro: an innocent like document can transform in a weapon if a macro is inserted (and blindly executed).

Hide More than a phase, a *condition*: as already mentioned, being hidden equals not being detected, that's fundamental for a malware. There are plenty of techniques to stay hidden: hiding in macros, being polyglot etc.

Deliver Payload Well: deliver payload. Take a bunch of harmful code and execute it where the virus had reached.

22 Defending from a Malware

Malwares are a problem. They can harm everything related to software, from the hardware itself, to communications, to assets, to public images. Defending against them is necessary: we see three main approaches.

- Software maintainance: often viruses exploits known software vulnerabilities. Patching them (and updating the softwares in general) makes the environment less exploitable from an attacker
- Malware signatures: so far one of the most utilized techniques to recognize malicious software. A program can be "recognized" from his signature (usually an hash derived from his bytecode). This enables antivirus to detect malicious softwares and block them.
- Behavioral and static executable analysis: viruses can worms can be recognized also by "known bad software practices" that are usually exploited, like the utilization of some known vulnerable library or the exhibition of a suspicious behaviour at runtime

22.1 Known Malicious Behaviours

Not all (known) patterns of execution of malwares are easily detectable or static:

- A program can be polymorphic or metamorphic, so be able to modify itself at runtime (for example inserting random dead code or on the fly code reordering). More specifically, we tlak about "software polymorphism" when the syntax and the overall code of the malware change during execution, but the semantic remains the same. If encryption is used to do so, then the technique called *packing* is being used. Instead, metamorphism alters the behaviour of the program *in an ineffective way*, so that the final result is achieved nonetheless. Techniques to metamorph a piece of code are inserting no-operation or useless always verified checks, or to scramble the execution flow without altering the algorithm.
- A malware can be dormant, and waiting for particular conditions to be met: a particular time, a trigger...
- Anti virtualization technique can be deployed: a program could *detect* when it's being debugged and change his behaviour
- Rootkits: rootkits defines an approach to hacking a machine. A rootkit is a set of tool that "implants a root user" on a machine. This root user can then use other pieces of the rootkit itself to harm the system itself. I like more the Wikipedia definition though[6].

[6] *A rootkit is a collection of computer software, typically malicious, designed to enable access to a computer or an area of its software that is not otherwise allowed (for example, to an unauthorized user) and often masks its existence or the existence of other software.*

Rootkits Philosophy at the base of the rootkit: "become root and use your toolkit to remain so". Rootkits are collections of softwares, that each helps in a task (like becoming root, hiding processes and srevices). The "hiding services and processes" part is crucial: to avoid being detected, hiding malicious processes (and the tracks left behind in general) is fundamental.
Rootkits are divided in two macro categories:

- Userland rootkits, that exploits userspace tools (like normal OS utilities) to remain hidden and gain accesses and priviledges.
- Kernel space rootkits, which instead are executed at kernel level. They're much more difficult to build and implant (as they require several high level authentication in order to do so) but they're also much more powerful. Kernel space rootkits can be detected only after *post mortem analysis*. The kernel space rootkits, operating at (and under) OS level can completely hide software artifacts, but also *compromise the system* in a significative way. They can even intercept and redirect syscalls.

At a higher level of technicality, Bootkits (rootkits hidden in the master boot record or in the boot artition) and HyperVisor level rootkits exists.

Part IX
Mobile Security

23 Mobile Devices

Why should an attacker target mobile devices? Here's a bunch of pros:

1. Always online
2. Nowadays, high computing capability
3. Handles sensitive data

So harming a person mobile device means that the attacker gains access to private emails and messages, maybe private accounts or personal transactions, and can be sure that a "way out" on the network is always present. This makes mobile devices extremely attractive for hackers.

24 Security Models for Mobile Operative Systems

24.1 Apple Model

Apple mobile operative system iOS implements code signing to enforce security. This method asks developers to digitally sign (usually through a checksum verifier method) the code and the whole app is verified *every time it's executed*. The certificate is signed by Apple Store, that is the only trusted element of the Apple ecosystem. *No unsigned code can be executed in an Apple device.*[7]
Another measure deployed is sandboxing with predefined permissions: the apps developed targeting apple iOS have no access to *any* file or hardware in the device; they can only access their "personal" directory. The kernel uses Mandatory Access Control to manage accesses.

24.2 Android Model

Android checks the *integrity* of the cryptographic sign of the app at installing time, but not during runtime. This allows the app to put in place dynamic code generation, or OTA silent updates. Also, every app can be *self signed* and there's no CA to regulate that.
Also Android sandboxes apps, but the permissions are handled by the underlying linux kernel.

[7] Unless jailbreak.

Part X
Exercises tips and notes

25 Binary Exploits

25.1 Buffer Overflows

For buffer overflows, we need to keep in mind how the stack is organized when a function is called. When function F calls functions D, in order, we have[8]:

1. Arguments in reverse order
2. The instruction pointer of F
3. The base pointer of F (this address will be the new base pointer)
4. The local variables of D:
    - if standard types, in order
    - if structs, the field are allocated in reverse order
    - if vectors, allocated from the last element

Remember: *stack grows towards smaller addresses.*
Also, watch out for the dimension of the exploit: if the shellcode/malicious code is bigger than the available space on the stack, it means that the only technique we can use is return to libc (or at least exploiting some sort of enviromental variable).

Building the exploit To exploit a buffer overflow vulnerability, we need to

1. Identify which variables we can control
2. Find the one initialized without check
3. Calculate how many bytes we have to overwrite in order to modify the return address of the function
4. Using this information to substitute the saved return address with one "known", possibly insert a nop sled[9] to be sure
5. The resulting exploit will 99% look like a sequence of nop followed by the actual exploit and an address targeting somewhere in the nop sled

Something to watch out: sometimes this string must contain a certain type of parameter, so watch out for other controls!

[8] in the reference architecture, so a GCC compiled C code with target a 32 bit machine, with all the options for debug and security disabled

[9] A series of null operations (nop) to "nullify" a portion of memory.

Constants In the referenced architecture, these are the usual values for the system variable types / general dimensions

Type	Size in bits	Size in bytes
Word (memory location length)	32	4
Pointer (and registers)	32	4
Integer	32	4
Char	8	1
Structs	(sum of bits of fields)	(sum of bytes of fields)

25.2 Format String

25.2.1 Remainder: 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 overwritten*;
- 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$n writing mechanism putting two consecutive addresses on the memory, as parameters for N$.* What's 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 "littler" (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

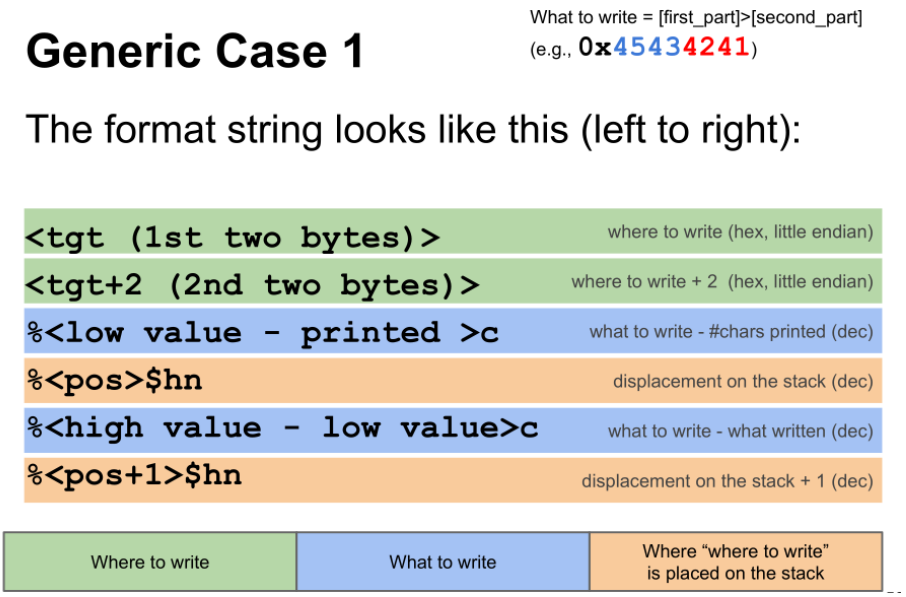
$$\begin{cases} \text{target addresses } TA,\ TA+2 \\ \text{malicious data } MD \\ \text{offset of the target address } OFF,\ OFF+1 \end{cases} \tag{3}$$

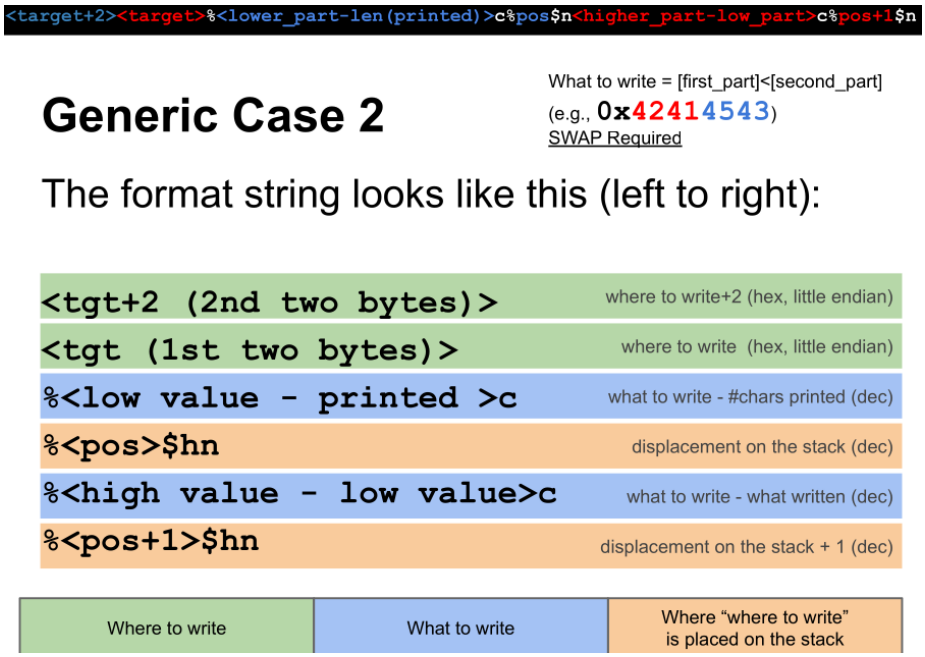
TA and TA+2 are written in the hexadecimal format with the "\x" every byte AND remember that the bytes are to be written in reverse order (usually little endian). REMINDER: if we are inverting the string (so the higher absolute value is the second part of the malicious data) also the two addresses are swapped in the final exploit.

$$(TA)(TA{+}2)\,\%(low|\,MD\,|{-}chars\,printed)\,c\%(OFF)\$hn\%(high{-}low)|\,MD|\,c\%(OFF{+}1)\$hn \tag{4}$$

A valid exploit to write "0x44674234" at address "0x42414515" is
"\x15\x45\x41\x42\x17\x45\x41\x42%16940c%7$hn%563c%8$hn"

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.





26 Web Vulnerabilities

26.1 Cross Site Scripting

It's important to recognize the XSS vulenrrability we're working with. Here's a list of signals you can look after for each XSS vulns we've seen

Type of error	Stored XSS	Reflected XSS	DOM-Based XSS
Unchecked insertion of data in the database	X		
Unchecked write of content to the client *from DB*	X		
Unchecked write of content to the client *from client itself*		X	X
Unchecked write of content to the client *using a client side technology*			X

26.2 Cross Site Request Forgery

In order to have a CSRF, some conditions are necessary:

1. cookies used somewhere
2. state changing actions
3. "automation" of user interaction through ambient variables (usually cookies, or sessionstorage)

What to remember for a valid CSRF attack: it *99% of the cases* involves 3 parties: these are the attacker, a *target website* and the client.

26.3 SQL Injection

To spot an SQL injection type of vulnerability is enough to see if there's some unchecked insertion in the database, some unsanitized input. Usually the countermeasures used are

- SQL prepared statements, that converts all inputs to string
- Client side input sanitizing (easily disabled)
- Server side input sanitizing

This kind of attacks usually require to extract some information from a DB; an SQLi injection exploit follows a rigid structure

1. a string terminator (apice or double apices) that is used to close the parameter that can be exploited
2. either a modified or additional condition to manipulate a `WHERE` clause, where usually the parameters are inserted
3. either a
    - query terminator plus comment symbol
    - UNION clause
    - apices (in the case we'd like to still use the part of the original query we've cut out)

If the injected code exploits a vulnerability in a `INSERT INTO` clause, the additional condition of the list is replaced with an additional `VALUES` clause.
REMEMBER: the malicious input should not undergo transformations that alter the payload (like base64 encoding or hashing of some kind). So WATCH OUT for this kind of encodings. Also, while building the exploit, remember that the data we've to verify that the data were correctly inserted OR we must be able to exfiltrate properly the information we're trying to steal. *This is not obvious* and can require several logical steps to get to the right infos.

27 Secure Network Architectures

27.1 ACLs and Firewall Rules

Classic approach to firewall rules in general:

1. deny all
2. allow selected
3. (additional) add reflection rules to allow symmetric communication

Typical exercise has this table to fill out (here with some rule example):

Firewall ID	SRC IP	SRC port	direction	DST IP	DST port	policy	description
FW1	any	any	net1 → net2	any	any	deny	deny all incoming
FW1	any	any	net1 → net2	any	any	deny	deny all outgoing
FW1	any	any	net1 → net2	any	80	allow	allow web traffic
FW1	any	80	net2 → net1	any	any	allow	allow web traffic, response rule to previous *only for stateless fws*

REMEMBER: the specified port for a protocol is one you *receive* on.

28 Malwares

Typical exercise on this part: recognize which techniques is being used by a program to hide. Quick list of easy-to-prove properties:

Property	Polymorphism	Packing	Metamorphism	Trigger based
Same instructions, different order (jumps inserted)			X	
Different instructions	X	X		
Encrypt decrypt primitives	X	X		
Compression primitives		X		
Same syscalls	X	X		
Lots of ”sleep” functions accessed through conditional instructions on some hardware property				X