

# Computer Security

Matteo Secco

May 26, 2021

## Contents

# 1 Introduction to Computer Security

## 1.1 Security requirements

### CIA Paradigm

**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 injection

**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 created for SQL injection

**Thread agent** whoever/whatever may cause an attack to occur

- 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 vulnerabilities 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 cryptography

**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:**  $\text{len}(m) = \text{len}(k)$

**Require:** keys not to be reused

**return**  $k \oplus m$

---

**Brute Force** perfect chypers are immune to brute force (as many "reasonable" messages will be produced). Real world chipers are not.

A real chipher is vulnerable if there is a way to break it that is faster then brute forcing

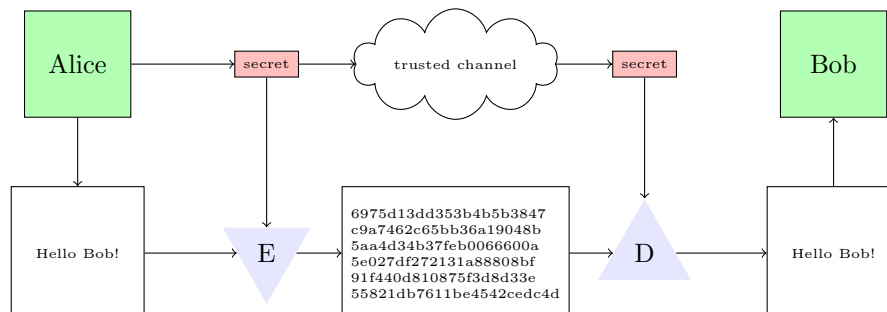
#### Types of attack

**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 decrypt the message

Scalability issue

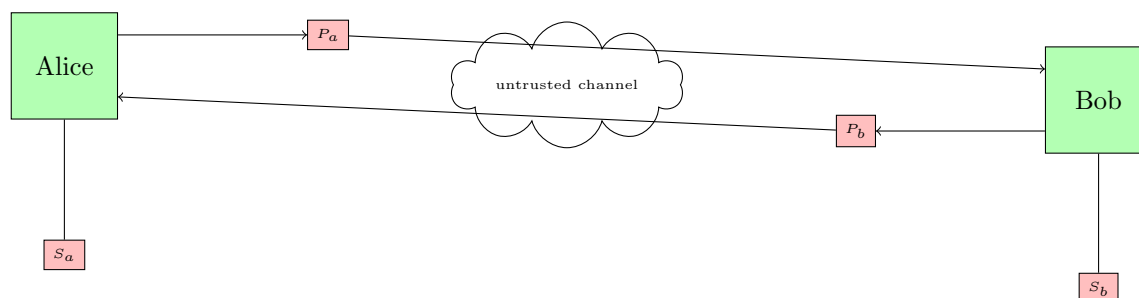
Key agreement issue

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

**Safety properties** are properties 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 enforces 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	...
...	...	...	...	...

#### 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  $\rightarrow$  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 columns: 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

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

**Lattice** Touple  $\langle \text{Level}, \text{Label} \rangle$ .

**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 than mine (to avoid leaking of information)

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

**Tranquility** Secrecy levels of objects cannot change dynamically

## 6 Software Security

Good software engineering  $\rightarrow$  meet requirements. Security is a non functional requirement. *The rest of the lesson is history and not particularly interesting*

## 7 Buffer Overflow

### 7.1 Memory stack

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

### 7.2 Registers

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

**ESP** Contains the address of the last stack operation: Top of the stack

**EBP** Contains the base of the current function frame

**Segment** 16-bit registers 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

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

```

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 }

```

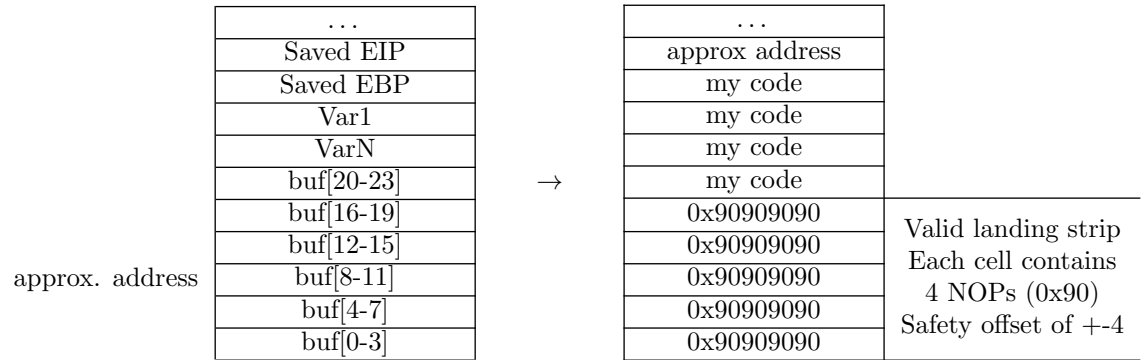


### Possible jumping destinations

- 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





**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

```

1 int main() {
2     char* hack[2];
3
4     hack[0]="/bin/sh";
5     hack[1]=NULL;
6
7     execve(hack[0], &hack, &hack[1]);
8 }

```

## 7.6 Defending against Buffer Overflow

### Shell code example

#### 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 chosen 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  $\wedge X = \text{something}$ , 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

```

1 #include <stdio.h>
2
3 void test(char* arg){
4     char buf[250];
5     snprintf(buf, 250, arg);
6     printf("buffer: %s\n", buf);
7 }
8
9 int main(int argc, char* argv[]) {
10     test(argv[1]);
11     return 0;
12 }

```

Two executions of the code above result in the following

```

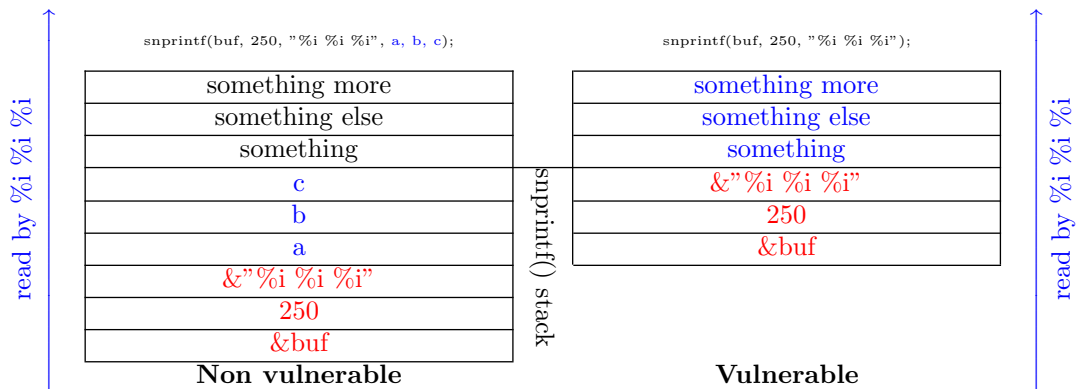
$ ./code "ciao"
buffer: ciao

```

```

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

```



```

1 #include <stdio.h>
2
3 void test(char* arg){
4     char buf[250];
5     snprintf(buf, 250, arg);
6     printf("buffer: %s\n", buf);
7 }
8

```

```
9 int main(int argc, char* argv[]) {  
10     test(argv[1]);  
11     return 0;  
12 }
```

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!
```

