MIECT: Security	2015-16

Practical Exercise:
Input Handling Attacks: Buffer Overflows

September 7, 2015	Due date: no date

Changelog

- v1.0 - Initial Version.

Introduction

The goal of this laboratory project is to assess some of the problems arising from improper handling of input data, in particular *buffer overflows*.
Also we expect you to gain experience with *buffer overflows*, to analyse the ways to detect them, and to understand their consequences. This project should be developed in a 32bit Linux system and using the C language.
It is not expected the inclusion of code inside the vulnerable test programs. While, programs can be modified as found appropriate in order to demonstrate the attack, data exploiting *buffer overflows* should be provided to the programs developed by means of packets through the UDP socket created.
For this laboratory you will need to use a VirtualBox image and code that is available at `http://www.joaobarraca.com/page/teaching/security`

Tips

- You can use the `netcat` program to sent data through a socket. In alternative, you can create a simple UDP client in any language you find appropriate. As an example you may send a given file named `data` to the server using `nc -v -u 127.0.0.1 12345 < data`.

- Please consider the GDB reference card available at the course web page.

- GCC v4.4 doesn't output CFI and LFE directives which GCC v4.6 produces. Therefore, the assembler code produced using GCC 4.4 is easier to understand. Install GCC 4.4 if possible.

1 Observation of ASLR

Address Space Layout Randomization is a mechanism employed to reduce the risk of *buffer overflows* by randomly arranging the positions of key data areas. The Stack address space provided to each application is also random.

Modify the file `server.c` so that you can print the address of a variable `addr`.

Verify the address of an internal variable (e.g., `addr`) when enabling and disabling the ASLR mechanism. In Linux this can be achieved by writing 0 or 1 to `/proc/sys/kernel/randomize_va_space`

2 Observation of a *buffer overflow*

Observe the assembly of the `sendEcho` function by compiling the program using the `gcc` compiler with the following options:

```
-S -masm=intel -fno-stack-protector
```

Note: The LEAVE instruction is equivalent to:

```
MOV esp, ebp
POP ebp
```

Check how the stack is managed before the call to function `sendEcho` and at the beginning and end of this function. Compute the length that the buffer `inbuffer` must have to provoke an *overflow* when `inbuffer` is written to `outbuffer`.

Check experimentally the length that the string `inbuffer` must have to effectively provoke a damaging *overflow*.

Create a *buffer overflow* scenario and observe its occurrence with the C debugger, `gdb`.

Note: To properly run a program in the debugger, with useful symbolic information (names of variables and functions, line information, etc.) the program must be compiled with the `-g` flag.

The flags `-masm=intel -fno-stack-protector` should also be used when compiling so that the assembly code produced is similar to the format used by `gdb`.

3 Memory allocation in the stack

Compile the program with the options `-z execstack -masm=intel -fno-stack-protector`.
Create a *buffer overflow* and observe the addresses of variables `outBuffer` and `sentTime`. Swap the order of declaration of variables and repeat the tests.

Compare the results observed.

Create a *buffer overflow* that sets the `sendTime` to 1. Verify that you received the current time through the socket.

4 Control of *buffer overflows* with canaries

The `gcc` version for the actual Linux distributions is shipped with a default option to protect stacks from overflows using canaries (options `-fstack-protector` and `-fstack-protector-all`).
These canaries protect critical stack elements using the StackGuard and SSP/Propolice strategies. Compile the same program with these stack protections and observe the generated assembly. Afterwards check what happens with the *buffer overflows* tested in the previous experiences.

5 Controlled jump into existing functions

Create a new function in the file `server.c` which sends the current user id (`man 2 getuid`) through the socket. Deliberately provoke a *buffer overflow* that creates a jump to the function when `sendEcho` returns.

Verify that the current user id is sent through the socket.

6 Execution of code injected into the stack

Control the *buffer overflow* of the previous program in order to provoke a jump **into the stack** and execute code inserted dynamically. As a recommendation, make a call to a system function (`_exit` or `exec`, for instance).

To find the assembly instructions required to make such a call, include a call to the function in a C program, execute it with the debugger and perform a step-by-step execution from the point of interest. See below:

```
void foo()
{
    exit(0);
}

int main (int argc, char **argv)
{
   foo();
}
```

Alternatively, disassemble the desired function using the commands `ar` and `objdump` (see example below for function `_exit`).

```
ar -x /usr/lib/i386-linux-gnu/libc.a _exit.o
objdump -Mintel -d _exit.o
```

In Linux, execution protection (NX) can be manipulated by using the `gcc` flag `-z execstack` or by using the `execstack` command line tool.

Because function `sprint` doesn't allow you to inject the 0x00 character (EOL), it may be necessary to create custom assembly code. This code will do the same as observed with `objdump` but without having the 0x00 character. Please consider that there are several instructions producing the same result. As an example, `mov eax, 0x00000001`, can be replaced by `xor eax,eax` plus `inc eax`.
In order to achieve this, you can create a text file with your assembly and then observe the assembly created. To compile the assembly file `exit.s` use: `nasm exit.s`. This will produce a file named `exit`. To observe the byte code produced, you can use `x86dis` or `hexdump`.

The following example describes the process of assembling and inspecting the result:

```
$> cat test.s
mov     eax,0x01
int     0x80
hlt

$> nasm test.s
```

```
$>x86dis -s intel  -e 0 -L < test
00000000 66 B8 01 00 00 00                    mov       eax, 0x00000001
00000006 CD 80                                int       0x80
00000008 F4                                   hlt
```

```
$>hexdump -e '/4 "%04X "' test;echo
1B866 80CD0000 00F4
```

You can also inject a call to bash, which is very common method for gaining access to a system. The shell would execute with the permissions of the user that launched the vulnerable application. For an example, check the support documentation or the instructions at `http://badishi.com/basic-shellcode-example/`