MIECT: Security 2017-18

Practical Exercise: Input Handling Attacks: Buffer Overflows

September 20, 2017 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 analyze the ways to detect them, and to understand their consequences. This project should be developed in a 32-bit Linux system and using the `C` language. While the vulnerability exists in all architectures, using a 32-bit system will facilitate analysis.

It is not expected the inclusion of code inside the vulnerable test programs. 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 messages over UDP.

For this laboratory you will need to use a VirtualBox image and code that is available at the course page.

Please download the following items:

- Examples: `http://atnog.av.it.pt/~jpbarraca/classes/security/bo-examples.zip`
- Resources: `http://atnog.av.it.pt/~jpbarraca/classes/security/bo-resources.zip`

Tips

- You can use the `netcat` program to send data through a socket. As an example you may send a file named `data` to the server at port 12345 with the following command:

  ```
  nc -v -u 127.0.0.1 12345 < data
  ```

  In alternative, you can create a simple `UDP` client in any language you find appropriate, to send custom messages to the vulnerable server. In `python` this would be:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto('Attack message: \xDE\xAD\xBE\xEF', (127.0.0.1 12345))
s.close()
```

- Please consider the GDB reference card available at the course Web page.
- GCC uses an extension named Call Frame Information which adds extra `.cfi` lines to ASM files. To disable this and improve readability, compile with `-fno-asynchronous-unwind-tables`.
- Different compilers or compiler versions will produce different output! When comparing your results with the ones presented in the slides or by the professor, take this in consideration.
- In order to compile 32-bit code, use the `-m32` flag when compiling with GCC.

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 critical data areas. The Stack address space provided to each application is also randomly placed.

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

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

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

```
echo 1 > /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 -fno-asynchronous-unwind-tables -fno-pic -m32
```

Their meaning is the following:

- `-S`: Produce assembly code, and not a binary;
- `-fno-stack-protector`: Control stack protection mechanisms (the objective of analysis of this exercise);
- `-fno-asynchcronous-unwind-tables` and `-fno-pic`: Disable GCC features for readability purposes;
- `-m32`: Produce 32-bit code, also for readability purposes.

Note: The `LEAVE` instruction is equivalent to:

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

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

Experimentally check the length that the `inbuffer` string must have to effectively provoke a damaging *overflow* (e.g., one that crashes the application).

Create a *buffer overflow* scenario and observe it with the C debugger, `gdb`. `gdb` in the Virtual machine is tweaked to provide a better user interface. In alternative you can use `gdbgui` or `nemiver`, which should be already pre-installed, and may provide even nicer interfaces.

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 `-m32 -fno-pic -masm=intel -fno-stack-protector` should also be used when compiling so that the assembly code produced is similar to the format presented by `gdb`.

3 Memory allocation in the stack

Compile the program using the following options:

`-m32 -fno-pic -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[1].

Compile the same program with these stack protections and observe the resulting assembly code. 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.

Hint: change also `server.c` to print the address of the new function, and use it in the data used in the *buffer overflow*.

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).

[1] http://wiki.osdev.org/Stack_Smashing_Protector

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 the `x86dis` or `hexdump` tools.

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 a shell command (e.g. `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 shell codes available at `http://shell-storm.org/shellcode/`