Coursework Part 2: Portfolio

Secure Programming and Exploit Development

Index

1 Portfolio

This section will contain 4 summary technical reports on experimental findings and each report will contain the results obtained with a brief discussion based on the lecture materials and research.

1 Static analysis with clang

```
scan-build: Using '/usr/lib/llvm-9/bin/clang' for static analysis
hello.c:4:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
    4 | main()
      | ^~~~
hello.c: In function ‘main’:
hello.c:6:2: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]
    6 |  printf("Hello World!");
      |  ^~~~~~
hello.c:6:2: warning: incompatible implicit declaration of built-in function ‘printf’
hello.c:1:1: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
  +++ |+#include <stdio.h>
    1 | /* Hello world */
scan-build: Removing directory '/tmp/scan-build-2020-11-21-093545-1784-1' because it contains no reports.
scan-build: No bugs found.
```

Image 1 – *scan-build hello.c*

- The main function is not declaring the returning type. By default it assumes it is `int` (line 2), adding the type before the function name solves this warning.
- The function `printf()` is not implicit declared (line 6), advising to include `stdio.h` (line 10) so it can import the official prototype, it still compiles since in run time GCC by default links the object files with the standard library. Adding the `stio.h` include solves this.

A prototype "(…) tells the compiler about the number of parameters function takes, data-types of parameters and return type of function. (GeeksforGeeks, 2017)".

Accessing the HTML report created by scan-build bad some bugs and bad programming practices are outlined.

Reports

Bug Group	Bug Type ▾	File	Function/Method	Line	Path Length	
Dead store	Dead assignment	6.c	test	7	1	View Report
Logic error	Dereference of null pointer	6.c	test	7	5	View Report
Memory error	Memory leak	6.c	main	13	6	View Report

Image 2 – *6.c html report*

```
4   if (p)
5     return;
6
7   x = p[0]; // warn
      Value stored to 'x' is never read
8 }
```

Image 2.1 – *6.c Dead assignment*

Variable x is not being used, memory is being wasted.

```
1   #include <stdlib.h>
2   void test(int *p) {
3     int x;
4     if (p)
          3 ← Assuming 'p' is null →
        4 ← Taking false branch →
5       return;
6
7     x = p[0]; // warn
          5 ← Array access (from variable 'p') results in a null pointer dereference
8   }
9
10  int main(){
11    int * p = (int *)malloc(sizeof(int));
12    test(p);
          1 Passing value via 1st parameter 'p' →
        2 ← Calling 'test' →
13  }
```

Image 2.2 – *6.c Deference of null pointer*

The pointer is not referencing a valid object. In line 11, `malloc()` is allocating the size of an integer, 4 bytes, and casting the returned pointer to an integer. In line 7, we are trying to access the first value that `p` is pointing to but it was never initialized.

```
13  }
      6 ← Potential leak of memory pointed to by 'p'
```

Image 2.3 – *6.c Memory Leak*

`Malloc()` reserves memory in the heap, it is a good coding practice to free memory even having a garbage collector that does that automatically. (Baeldung, 2020)

A logic problem that I detected is that line 7 (Image 2.2) is going to read a value that will not exist because line 4 checked that the memory allocation was unsuccessful.

Clang can analyse C, C++, ObjC and their variants.

```
TranslationUnitDecl 0x252c988 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x252d220 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x252cf20 '__int128'
|-TypedefDecl 0x252d290 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x252cf40 'unsigned __int128'
|-TypedefDecl 0x252d578 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0x252d370 'struct __NSConstantString_tag'
|   `-Record 0x252d2e8 '__NSConstantString_tag'
|-TypedefDecl 0x252d610 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x252d5d0 'char *'
|   `-BuiltinType 0x252ca20 'char'
|-TypedefDecl 0x252d8e8 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0x252d890 'struct __va_list_tag [1]' 1
|   `-RecordType 0x252d6f0 'struct __va_list_tag'
|     `-Record 0x252d668 '__va_list_tag'
|-FunctionDecl 0x258b668 <5.c:1:1, line:5:1> line:1:6 used test 'void (int)'
| |-ParmVarDecl 0x258b5a0 <col:11, col:15> col:15 used z 'int'
| `-CompoundStmt 0x258b938 <col:18, line:5:1>
|   |-DeclStmt 0x258b7d8 <line:2:3, col:8>
|   | `-VarDecl 0x258b770 <col:3, col:7> col:7 used x 'int'
|   `-IfStmt 0x258b920 <line:3:3, line:4:13>
|     |-BinaryOperator 0x258b848 <line:3:7, col:12> 'int' '=='
|     | |-ImplicitCastExpr 0x258b830 <col:7> 'int' <LValueToRValue>
|     | | `-DeclRefExpr 0x258b7f0 <col:7> 'int' lvalue ParmVar 0x258b5a0 'z' 'int'
|     | `-IntegerLiteral 0x258b810 <col:12> 'int' 0
|     `-BinaryOperator 0x258b900 <line:4:5, col:13> 'int' '='
|       |-DeclRefExpr 0x258b868 <col:5> 'int' lvalue Var 0x258b770 'x' 'int'
|       `-BinaryOperator 0x258b8e0 <col:9, col:13> 'int' '/'
|         |-IntegerLiteral 0x258b888 <col:9> 'int' 1
|         `-ImplicitCastExpr 0x258b8c8 <col:13> 'int' <LValueToRValue>
|           `-DeclRefExpr 0x258b8a8 <col:13> 'int' lvalue ParmVar 0x258b5a0 'z' 'int'
`-FunctionDecl 0x258b9b0 <line:7:1, line:9:1> line:7:5 main 'int ()'
  `-CompoundStmt 0x258baf8 <col:11, line:9:1>
    `-CallExpr 0x258bad0 <line:8:3, col:9> 'void'
      |-ImplicitCastExpr 0x258bab8 <col:3> 'void (*)(int)' <FunctionToPointerDecay>
      | `-DeclRefExpr 0x258ba50 <col:3> 'void (int)' Function 0x258b668 'test' 'void (int)'
      `-IntegerLiteral 0x258ba70 <col:8> 'int' 0
```

Image 3 – *5.c AST Dump Clang*

Information such as memory addresses and position are outlined. The `TranslationUnitDecl` is the top-level structure, `TypededDecl` are internal declarations of clang. `FunctioDecl` are function declarations, ‘`test`’ and ‘`main`’ in this case. The AST has three isolated core nodes which are `Stmt` (statement), `Decl` (declaration) and `Expr` (expression). Each tree node can access trough traversal methods. For instance in this case we have an if-statement, `IfStmt`, has one conditional `Expr` that is the `BinaryOperator` and one if true `Stmt`. In this case, the `IsStmt` is transversing the tree calling the dedicated methods:

```
const Expr * getCond () const
const Stmt * getThen () const
```

(Devlieghere, 2015)

This exercise reflected exactly my understanding on static analysis based upon the lectures.

When using `#include` Clang will perform the analysis for the imported library recursively from every header and every header they include being this a poor performance mechanism to access the API of the libraries. Another problem is that compilation errors and breaks on library APIs can surge from collisions originated from same textual inclusion names. Clang can transform the textual inclusion to a semantic model, these models are only imported once per header. For example `#include <stdio.h>` will be mapped to `std.io` module.

Klee has the advantage of checking all possible values within the program using symbolic execution and constraint solving. It considers all feasible paths and input values, finding bugs while testing the paths and checks dangerous operations such as Pointer references, Array indexing, Division/module operations and Assert statements. (Cadar, 2016)
IKOS trough the usage of Abstract Interpretation offers: code parsing, model development, abstract domain management, results management, and analysis strategy. (Brat, Navas and Shi, 2020)
These programs have a heuristic nature so they give approximated answers and also need a huge amount of time and memory to perform the analysis and possible result in a lot of false positives.

2 AV evasion

Downloading the Amplia's Windows Credentials Editor WCE v1.42beta (64-bit) and testing it VirusToal I got 56 from 72 AV tools classifying it as malware:

Image 4 –*VirusTotal wce.exe*

Using Find-AVSignature to split the program into multiple binaries containing each 1000bytes, starting from the first byte and ending in the last one, we will use the divide and conquer approach to know in which bytes the AV is detecting malware patterns.

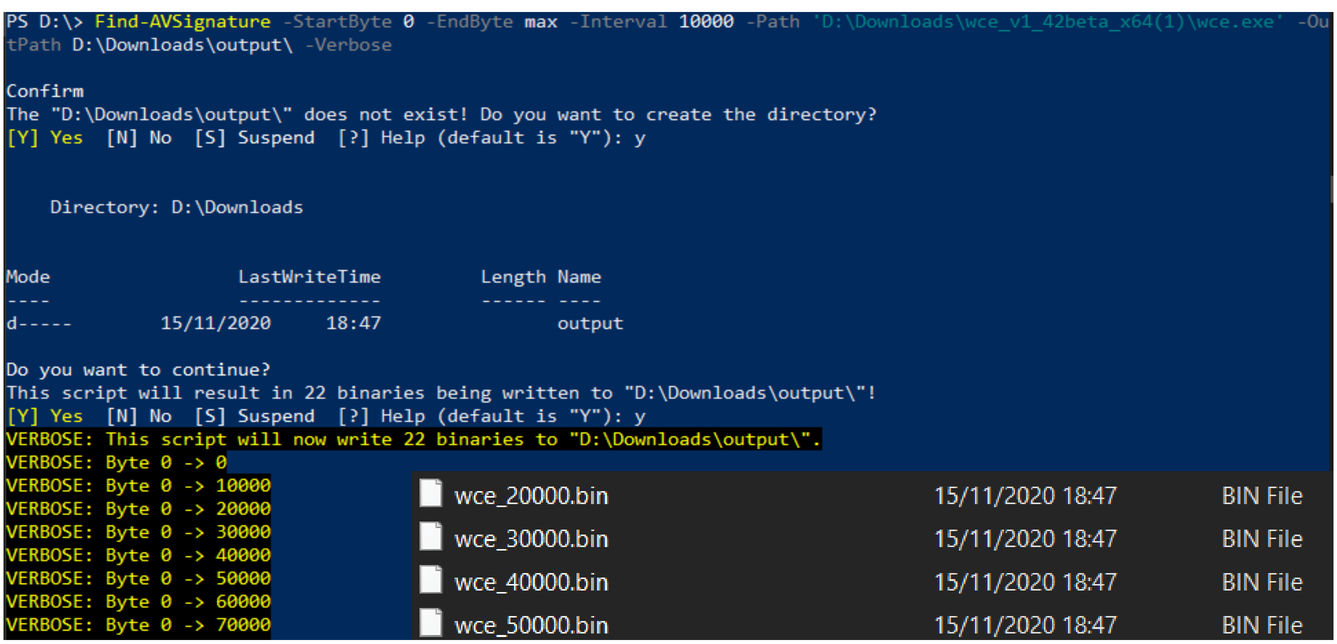


Image 5 – *Find-AVSignature first division* wce.exe was divided into 23 binary files. I scanned the output folder that contained all the binaries with WindowsDefender and got the following result:

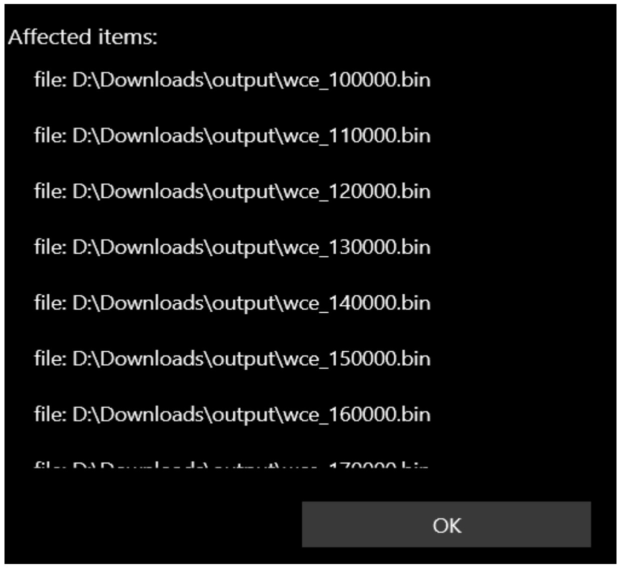
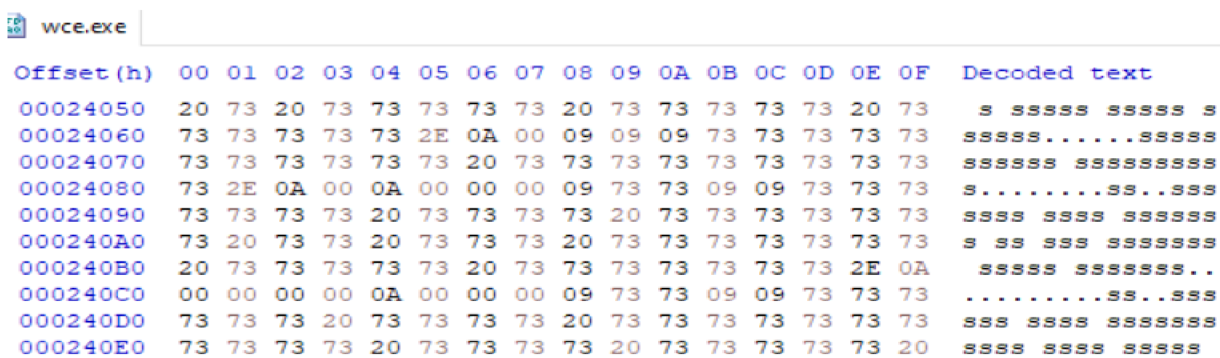


Image 6 – *Windows Defender Binaries*

Therefore I scanned them in VirusTotal and kept doing the same process between the binaries with a higher number of antivirus detecting its signatures and shrinking the data between divisions. I reached the StartByte 217080 and EndByte max, with an Interval of 1, and the last file had the highest score (wce_217087.bin) of 56/72. Opening the file I noticed that the piece of the data is utilized to give to the user information output such as how to use it and some error information. I changed this data since this is not essential for the program's functionality and added some random characters with a hex-editor.



wce.exe

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00024050	20	73	20	73	73	73	73	73	20	73	73	73	73	73	20	73	s sssss sssss s
00024060	73	73	73	73	73	2E	0A	00	09	09	09	73	73	73	73	73	sssss......sssss
00024070	73	73	73	73	73	73	20	73	73	73	73	73	73	73	73	73	ssssss sssssssss
00024080	73	2E	0A	00	0A	00	00	00	09	73	73	09	09	73	73	73	s........ss..sss
00024090	73	73	73	73	20	73	73	73	73	20	73	73	73	73	73	73	ssss ssss ssssss
000240A0	73	20	73	73	20	73	73	73	20	73	73	73	73	73	73	73	s ss sss sssssss
000240B0	20	73	73	73	73	73	20	73	73	73	73	73	73	73	2E	0A	sssss sssssss..
000240C0	00	00	00	00	0A	00	00	00	09	73	73	09	09	73	73	73	.........ss..sss
000240D0	73	73	73	20	73	73	73	73	20	73	73	73	73	73	73	73	sss ssss sssssss
000240E0	73	73	73	73	20	73	73	73	73	20	73	73	73	73	73	20	ssss ssss sssss

Image 7 – *HxD wce.exe string substituition*

After scanning the modified wce.exe I got a score of 36/72, bypassing 20 AVs.

Image 8 – modified wce.exe

The other AVs were more sophisticated and even trying to continue using a trial-and-error approach to change more characters, without corrupting the program, they were still detecting malicious patterns.

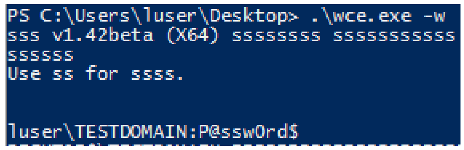


Image 9 – *Modified wce.exe running*

The Pecloack.py is a multi-pass encoder and heuristic sandbox, that does not work with 64-bit versions, so I downloaded the 32-bit one. This version had a 55/69 score. I will use this cloaking tool to check its efficiency.

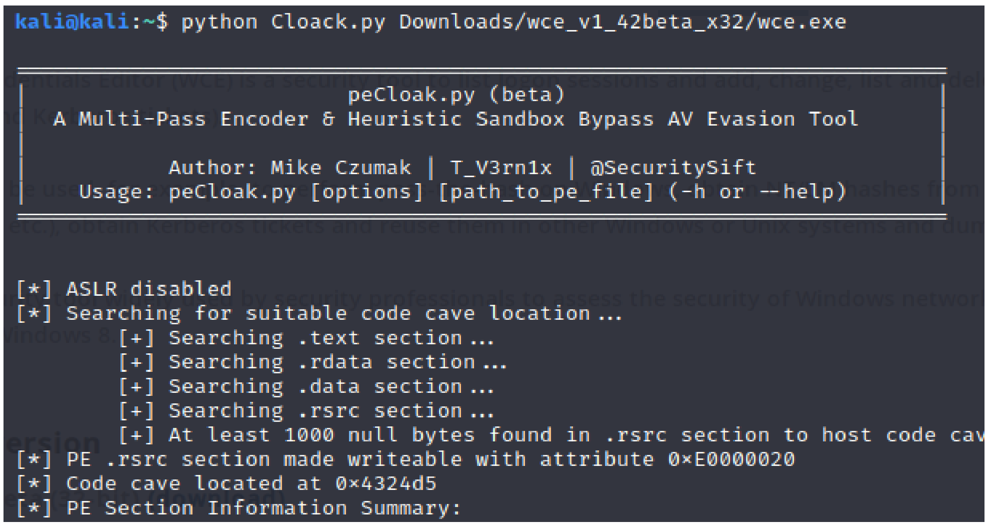


Image 10 – *pecloack.py wce.exe*

Pecloack.py was able to bypass 8 engines in the first time and 9 engines in the second time with H=100 (Heuristic iterations). Reading its source code it seems that the program tries to apply heuristic bypass methods, adding random instructions, it encrypts the source code, with ADD, SUB, XOR operations that will be decrypted at run time, it uses code cave, which are redirects added to existing executables (heuristic bypass and decoding). This seems a good tool to bypass simple AVs, but they are moving to behaviour analysis rather than digital signature detection which makes these techniques less effectve.

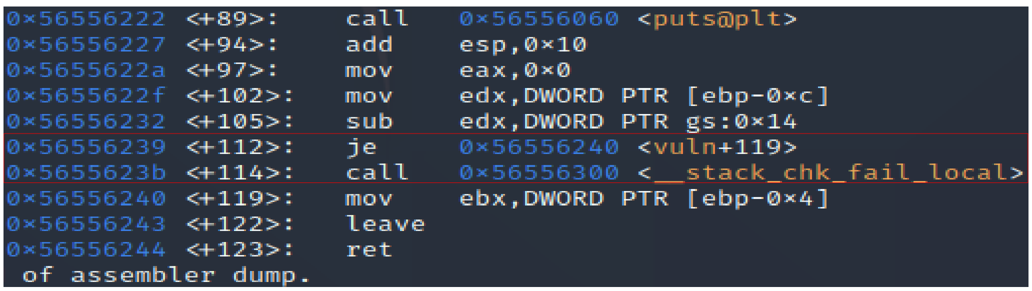
WCE is a tool which has the same behaviour as malware, it can execute Pass-the-Hash attack, dump cleartext passwords, steal NTLM credentials and Kerberos tickets accessing Windows and Unix machines trough them. This is destined for penetration testers check their systems' security. (AmpliaSecurity, n.d.)

In my case, I used Windows Defender, which instantly picked wce.exe has malware. Windows Defender has real-time behaviour and heuristic analysis, programs are closely monitored and if any flags are detected it prevents further actions, removing the program with the user authorization. This anti-malware does not necessarily block only malware, it marks all apps which their behaviour has malicious patterns has unsafe. (Microsoft, 2020)

Summing up, anti-malware is evolving to a behavioural analysis approach since malware is more sophisticate and complex. “Rather than track known threats (Cloonan, 2017)” they are tracking behaviours, this means that AVs will treat all unsafe apps the same.

3 32 bit stack smashing

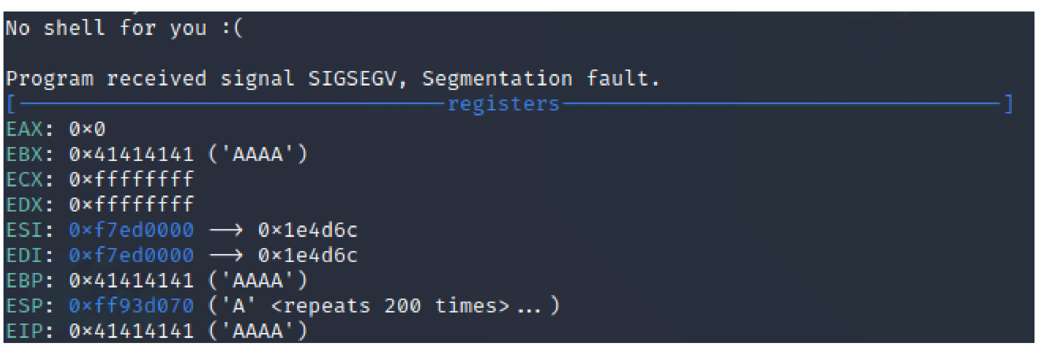
After setting up the needed configurations I compiled the code with the flag `-fstack-protector` (classic32prot) and without it (classic2). Kali Linux as default has this flag turned off that is why I did the inverse. This flag tells GCC to add a guard variable for every function containing vulnerable objects (canary).



```
0x56556222 <+89>:    call   0x56556060 <puts@plt>
0x56556227 <+94>:    add    esp,0x10
0x5655622a <+97>:    mov    eax,0x0
0x5655622f <+102>:   mov    edx,DWORD PTR [ebp-0xc]
0x56556232 <+105>:   sub    edx,DWORD PTR gs:0x14
0x56556239 <+112>:   je     0x56556240 <vuln+119>
0x5655623b <+114>:   call   0x56556300 <__stack_chk_fail_local>
0x56556240 <+119>:   mov    ebx,DWORD PTR [ebp-0x4]
0x56556243 <+122>:   leave
0x56556244 <+123>:   ret
 of assembler dump.
```

Image 11 – *GCC-peda classic32prot*

In image 11, the instruction in address `0x56556239` is where the canary verification is made and will call `__stack_chk_` fail if the canary was modified. This function will terminate the function that was called from while sending an overflow detection message and subsequently terminates the program. classic32 does not have these protection instructions. We can test this by passing 200 ‘A’ chars, from in.txt, for each program.



```
No shell for you :(

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0xffffffff
EDX: 0xffffffff
ESI: 0xf7ed0000 --> 0x1e4d6c
EDI: 0xf7ed0000 --> 0x1e4d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xff93d070 ('A' <repeats 200 times>...)
EIP: 0x41414141 ('AAAA')
```

Image 12 – *GCC-peda classic32 in.txt*

```
No shell for you :(
*** stack smashing detected ***: terminated

Program received signal SIGABRT, Aborted.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x2
ECX: 0xffffcdcc --> 0x0
EDX: 0x0
ESI: 0x8
EDI: 0x0
EBP: 0xffffcdcc --> 0x0
ESP: 0xffffcdb0 --> 0xffffcdcc --> 0x0
EIP: 0xf7fd2179 (<__kernel_vsyscall+9>: pop    ebp)
```

Image 13 – *GCC-peda classic32prot in.txt*

As we can see in Image 12 registers where overwritten while in Image 13 a stack smashing was detected terminating the program without overwriting registers. The next step is to find the offset of EIP feeding a huge random pattern into the program, it will overwrite all the registers, asking gdb to check their memory address and peda to give their offset.

```
gdb-peda$ pattern_create 400 in.txt
Writing pattern of 400 chars to filename "in.txt"
gdb-peda$ r < in.txt
gdb-peda$ x/wx $esp
0xfff7ba70:     0x41684141
gdb-peda$ pattern_offset 0x41684141
1097351489 found at offset: 100
```

Image 14 – *esp offset discover*

Doing the same process for the EIP we got 96 as the offset. Substituting x with 96 and y with 300 we get:

```
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd210 ('C' <repeats 200 times>...)
EIP: 0x41414141 ('AAAA')
```

Image 15 – *createpattern2.py check offset*

The next step is disabling overflow protections such as NX that marks memory as non-executable preventing shell code execution and ASLR to get a static address to pass into EIP.

```
kali@kali:~/Downloads/stack_smash32$ sudo execstack -s classic32
kali@kali:~/Downloads/stack_smash32$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
```

Image 16 – *disabling NX and ASLR*

Now we need a shellcode. I downloaded the recommended one and tried to run it but always ended with a segmentation fault error. At this point, I discovered that the shellcode was not compatible with my computer's architecture, I then searched for an appropriated one. After testing it I still got segmentation-fault errors. This happened because Linux ignores setuid for files containing shebang (!#) headers. To bypass this restriction the shellcode must do a system call to force setting the user id with `setuid(0)`.

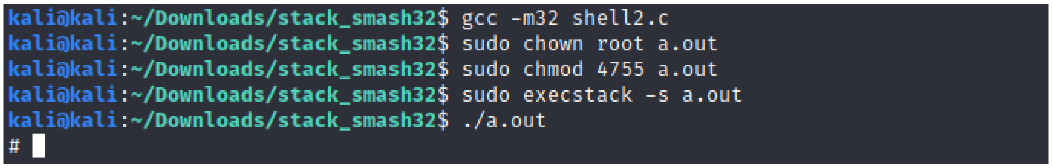


Image 17 – *shellcode root test*

Having a working shellcode we have to inject it in the stack and get its address to perform the actual exploit.

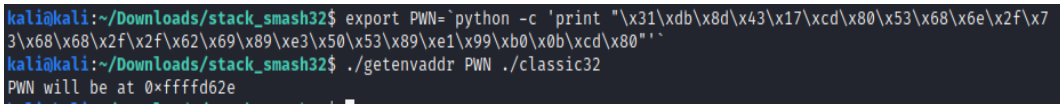


Image 18 – *shellcode payload*

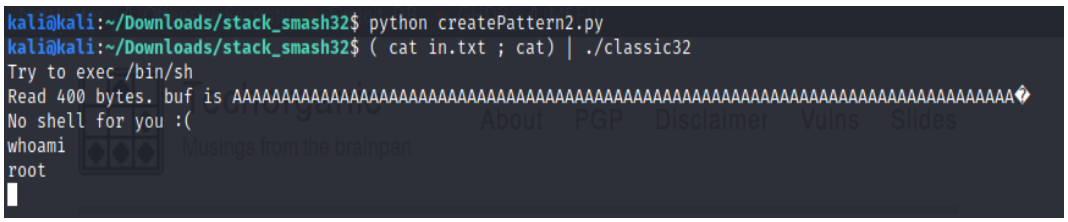


Image 19 – *exploiting*

To explain how the attack worked I will give a brief background. The stack is fragmented into contiguous small regions that are associated with each function call holding its respective argument data called frames. (QNX, n.d.)

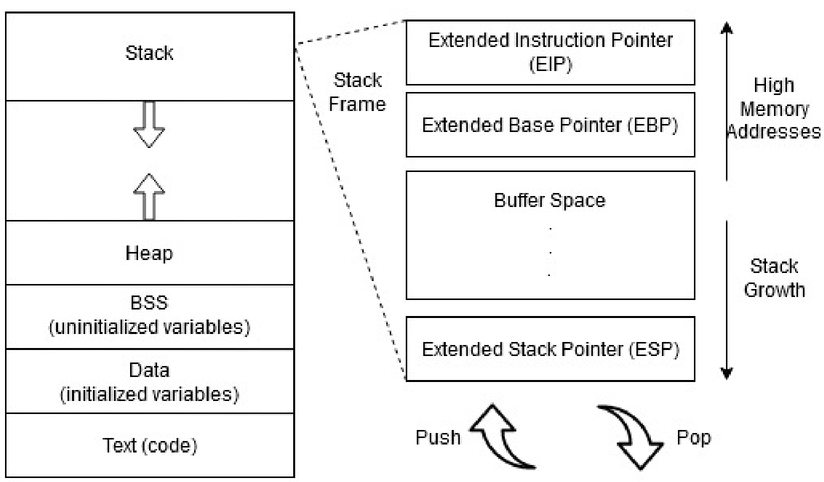


Diagram 1 – *Memory anatomy*

EIP has the address of the next instruction from where the `call` was made. When we send to the buffer a larger amount of data than it can handle it will start overwriting the top of the stack until reaching the EIP. When we overwrite reach the EIP and overwrite its data we can hardcode an address location.

When the ESP reaches the EIP it will then be moved to that address executing the instructions in that location.

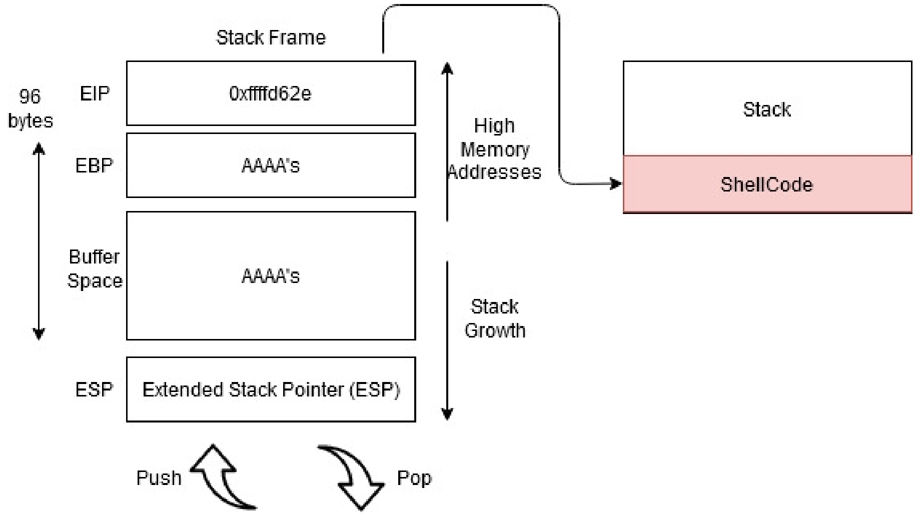


Diagram 2 – *Buffer Overflow*

A computer with an x64 architecture can run a 32-bit version software as I did in this exercise. The difference between 32-bit and 64-bit is mainly the size of register and buses. In general, to perform a buffer overflow we have to study the programs' behaviour and their flaws since architectures' differences are minor. For example, in 64-bit the instruction pointer is the RIP. This means that in this exercise if RIP is controlled the same attack will be performed in the same way. (Mattsson, 2010)

4 Binary Mangling

For task 1 I compiled the program with the -g flagged (simpleg) and without it (simple).

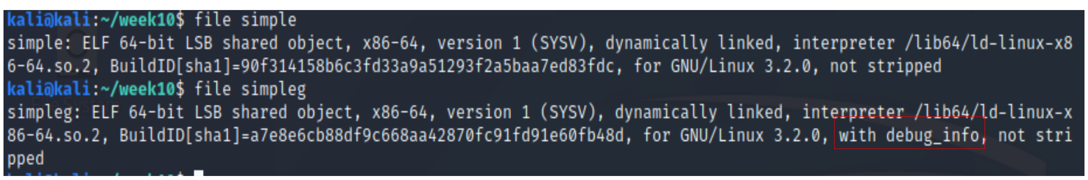


Image 20 – *file command*

Using file, to perform fileystem, magic and language tests, on both programs simpleg stated the additional information "w*ith debug_info*".

Using gdb to disassemble the main function as expected we see the debug information.

```
9               scanf("%s", &input);
   0x0000555555555189 <+36>:    lea    rax,[rbp-0x40]
   0x000055555555518d <+40>:    mov    rsi,rax
   0x0000555555555190 <+43>:    lea    rdi,[rip+0xea9]        # 0x555555556040
   0x0000555555555197 <+50>:    mov    eax,0x0
   0x000055555555519c <+55>:    call   0x555555555060 <__isoc99_scanf@plt>

10              printf("Checking input\n");
   0x00005555555551a1 <+60>:    lea    rdi,[rip+0xe9b]        # 0x555555556043
   0x00005555555551a8 <+67>:    call   0x555555555030 <puts@plt>
```

Image 21 – *gdb simpleg*

Doing now the Task2_GDB, the first command sets the display style of the disassembly code to Intel and the second one will disassemble the main function.

```
gdb-peda$ set disassembly-flavor intel
gdb-peda$ disassemble main
```

Image 22 – *Task2_GDB recon*

```
0x00005555555551e1 <+60>:    call   0x5555555550a0 <rand@plt>
0x00005555555551e6 <+65>:    pxor   xmm0,xmm0
0x00005555555551ea <+69>:    cvtsi2sd xmm0,eax
0x00005555555551ee <+73>:    movsd  xmm2,QWORD PTR [rip+0xf6a]        # 0x555555556160
0x00005555555551f6 <+81>:    movapd xmm1,xmm0
0x00005555555551fa <+85>:    divsd  xmm1,xmm2
0x00005555555551fe <+89>:    movsd  xmm0,QWORD PTR [rip+0xf62]        # 0x555555556168
0x0000555555555206 <+97>:    mulsd  xmm1,xmm0
0x000055555555520a <+101>:   movsd  xmm0,QWORD PTR [rip+0xf5e]        # 0x555555556170
0x0000555555555212 <+109>:   addsd  xmm0,xmm1
0x0000555555555216 <+113>:   cvttsd2si eax,xmm0
0x000055555555521a <+117>:   mov    DWORD PTR [rbp-0x8],eax
0x000055555555521d <+120>:   mov    eax,DWORD PTR [rbp-0x8]
0x0000555555555220 <+123>:   mov    edx,eax
0x0000555555555222 <+125>:   mov    eax,DWORD PTR [rbp-0x4]
0x0000555555555225 <+128>:   cdqe
0x0000555555555227 <+130>:   mov    BYTE PTR [rbp+rax*1-0x12],dl
0x000055555555522b <+134>:   add    DWORD PTR [rbp-0x4],0x1
0x000055555555522f <+138>:   cmp    DWORD PTR [rbp-0x4],0x9
0x0000555555555233 <+142>:   jle    0x5555555551e1 <main+60>
0x0000555555555235 <+144>:   mov    BYTE PTR [rbp-0x8],0x0
0x0000555555555239 <+148>:   lea    rax,[rbp-0x12]
0x000055555555523d <+152>:   mov    rdi,rax
0x0000555555555240 <+155>:   call   0x555555555040 <strlen@plt>
```

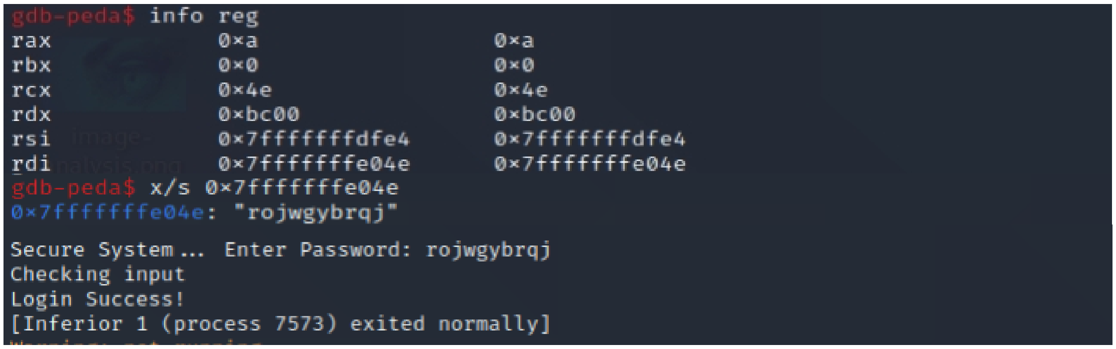
Image 23 – *Task2_GDB code analysis*

I identified fundamental function calls such as `rand` and `strlen`. I know that the code will generate 10 random chars and if it fails the program will end. To check if the `msg` has 10 chars the program is using `strlen`. This will be the best location to put a breakpoint because we can read the registers in time to get the generated password.

```
gdb-peda$ b* 0x0000555555555245
Breakpoint 2 at 0x555555555245: file Task2_GDB.c, line 25.
```

Image 24 – *Task2_GDB setting breakpoint*

After running the program, gdb hits the breaking point. I suspected that the password was stored in RDI so with the second command I will display the register memory contents in string format, giving the correct password.

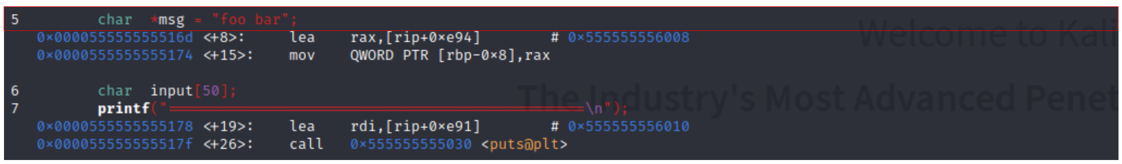


```
gdb-peda$ info reg
rax            0×a                 0×a
rbx            0×0                 0×0
rcx            0×4e                0×4e
rdx            0×bc00              0×bc00
rsi            0×7fffffffdfe4      0×7fffffffdfe4
rdi            0×7fffffffe04e      0×7fffffffe04e
gdb-peda$ x/s 0×7fffffffe04e
0×7fffffffe04e: "rojwgybrqj"

Secure System ... Enter Password: rojwgybrqj
Checking input
Login Success!
[Inferior 1 (process 7573) exited normally]
```

Image 25 – *Task2_GDB password discover*

Task4 asks to use radare2 to change the program behaviour. I started using GDB since it is more practical in debugging. The first thing that I want to know is the password which is “`foo bar`”.

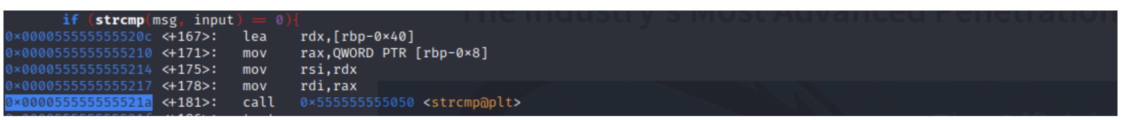


```
5         char  *msg = "foo bar";
   0×000055555555516d <+8>:     lea    rax,[rip+0×e94]        # 0×555555556008
   0×0000555555555174 <+15>:    mov    QWORD PTR [rbp-0×8],rax

6         char  input[50];
7         printf("==========================================\n");
   0×0000555555555178 <+19>:    lea    rdi,[rip+0×e91]        # 0×555555556010
   0×000055555555517f <+26>:    call   0×555555555030 <puts@plt>
```

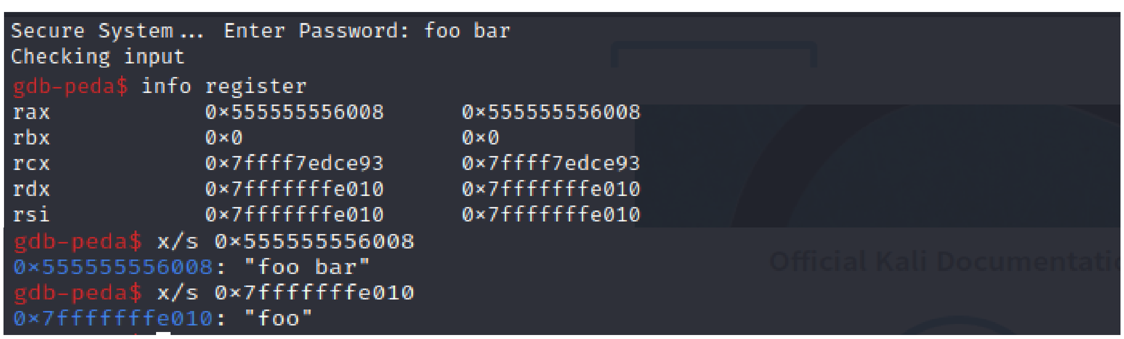
Image 26 – *Task4_R2Hacking password and input*

I inserted a breakpoint in the previous address of `strcmp`. At this point, I want to know how the input is handled.



```
        if (strcmp(msg, input) == 0){
0×000055555555520c <+167>:   lea    rdx,[rbp-0×40]
0×0000555555555210 <+171>:   mov    rax,QWORD PTR [rbp-0×8]
0×0000555555555214 <+175>:   mov    rsi,rdx
0×0000555555555217 <+178>:   mov    rdi,rax
0×000055555555521a <+181>:   call   0×555555555050 <strcmp@plt>
```

Image 27 – *Task4_R2Hacking address breakpoint*



```
Secure System ... Enter Password: foo bar
Checking input
gdb-peda$ info register
rax            0×555555556008      0×555555556008
rbx            0×0                 0×0
rcx            0×7ffff7edce93      0×7ffff7edce93
rdx            0×7fffffffe010      0×7fffffffe010
rsi            0×7fffffffe010      0×7fffffffe010
gdb-peda$ x/s 0×555555556008
0×555555556008: "foo bar"
gdb-peda$ x/s 0×7fffffffe010
0×7fffffffe010: "foo"
```

Image 28 – *Task4_R2Hacking reading registers*

With this, I concluded that there is a flaw in the input. `scanf` function reads input until it encounters a white-space, newline or End Of File(EOF). To fix this flaw we need to replace “%s” with “%[^\n]” in `scanf` parameters (Bollinger, 2018). This parameter will say that the data to be read will consist of a run of non-newline characters. We have to overwrite a constant string, they are stored in the `.rodata` (read-only data) segment and replace the `scanf` parameter with the new string address (Anon, 2019). First, we list the mentioned read-only strings which are contained in the section of ELF binaries and copy the address of a worthless string (i.e aesthetic strings).

```
kali@kali:~/week10/Tutorial_10_Codes$ rabin2 -z ./Task4_R2Hacking
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00002008 0x00002008 7   8    .rodata ascii foo bar
1   0x00002010 0x00002010 48  49   .rodata ascii ════════════════════════════════════════════════
2   0x00002048 0x00002048 48  49   .rodata ascii #                                              #
3   0x00002080 0x00002080 48  49   .rodata ascii #  PASSWORD ENTRY SYSTEM                       #
4   0x000020b8 0x000020b8 48  49   .rodata ascii #  ONLY AUTHORIZED USERS BEYOND THIS POINT     #
5   0x000020f0 0x000020f0 34  35   .rodata ascii \nSecure System... Enter Password:
6   0x00002116 0x00002116 14  15   .rodata ascii Checking input
7   0x00002125 0x00002125 14  15   .rodata ascii Login Success!
8   0x00002134 0x00002134 12  13   .rodata ascii Login Fails!
```

Image 29 – *Task4_R2Hacking address .rodata*

I chose the string from line 1. Now we will run radare2 and modify that string navigating to its address.

```
kali@kali:~/week10/Tutorial_10_Codes$ radare2 ./Task4_R2Hacking
[0x00001080]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00001080]> oo+
```

Image 30 – *Task4_R2Hacking prepare to overwrite*

```
[0x00001080]> 0x00002010
[0x00002010]> w %[^\n]
[0x00002010]> x
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00002010  255b 5e0a 5d3d 3d3d 3d3d 3d3d 3d3d 3d3d  %[^.]═══════════
0x00002020  3d3d 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d  ════════════════
0x00002030  3d3d 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d 3d3d  ════════════════
0x00002040  0000 0000 0000 0000 2320 2020 2020 2020  ........#
```

Image 31 – *Task4_R2Hacking replacing strings*

After modifying the string we need to replace the old parameter (%s address) with the new one (%[^\n] address). Since this program was compiled with debugging information it is easy to know where we need to do it, we have an indication saying that an effective address containing “%s” is being loaded to RDI.

```
0x000011ef      488d3d1d0f00.   lea rdi, qword [0x00002113] ; "%s" ; const char *format
```

Image 32 – *Task4_R2Hacking %s address*

```
[0x00002010]> 0x000011ef
[0x000011ef]> wa lea rdi, qword [0x00002010]
Written 7 byte(s) (lea rdi, qword [0x00002010]) = wx 488d3d1a0e0000
```

Image 33 – *Task4_R2Hacking operation overwrite*

As aspected the program flaw is fixed and that “%[^\n]” was also printed since we modified a string that was being printed.

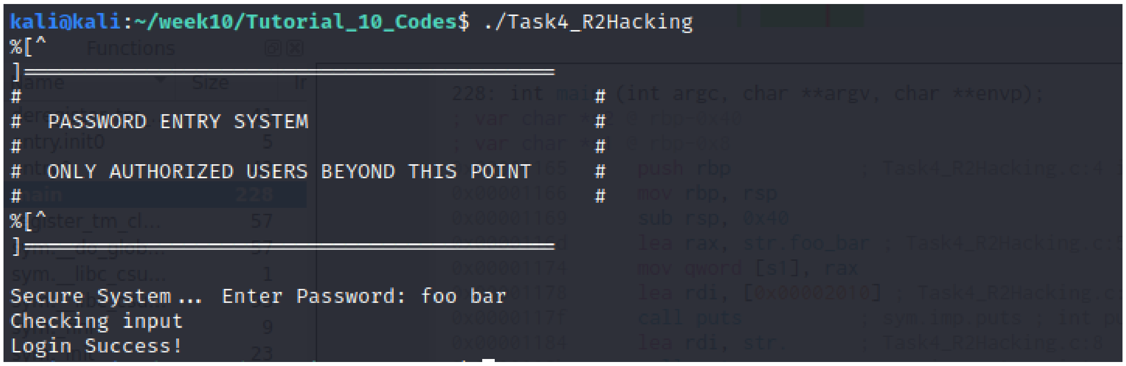


Image 34 – *Task4_R2Hacking testing*

In this portfolio we used reversed engineering to bypass a program’s authentication and to fix a program flaw. Reverse engineering can also be used to make products compatible with each other, “learn the principals that guided a competitors’ design”, to detect code theft and check if the program does what it is claimed to do. (Gomulkiewicz and Williamson, 1996)

2 Reflection

I will divide the learnt material into three topics: code analysis, malware and reverse engineering referencing the taught materials and technical reports outlining their importance in the investigation process. This reflection will be conducted based on the Borton’s development framework, starting with the explanation of what I learnt from the materials, how it was applied in the reports and their real-world applications.

1 Code analysis

The static analysis consists of testing a program without executing it trough the examination of the source to check type, style, flow, find bugs and perform security reviews. A static analysis program generates a token stream parsing the original code to build an Abstract Syntax Tree (AST). With the AST a control flow graph is built (Kundel, 2020).

Dynamic analysis is the process of analysing a program while it is on execution. It is used for example on the Verification phase of Microsoft SDLC. This is an easy and cheap way to debug, maintain and ensure the quality of a program in all scenarios for which it was designed for (Rouse, 2006).

In Portfolio’s section 1, I used Clang to analyse some simple programs that have flaws. This tool created a detailed report in multiple formats containing the programs’ potential bugs. The report also contained precise details about them such as the route which Clang took to originate them. I could see that the tool tries to list all the things that the program can do. It enabled a fast check and fix on all programs, is easy to understand why companies use it. The Clangs' AST was analysed meeting the expected predictions, from the source code a symbolc logic was representing the program. It was capable of analysing all the different programs because is using abstract interpretation. This enables the tools to answer undecidable problems because it tries to do an approximation of similar problems for which it has an answer (Hearn, 2020) .

Code analysis is an important feature that many companies use and some even have their own tools such as fbinfer from Facebook. It is crucial for the companies not only to find as many bugs as they can to prevent them to go into production but also to ensure that security standards or functional standards are validated (Slansky, 2020). This means that the chances that I am going to work with this type of tools are very high.

2 Malware

Malware is defined by Microsoft as "(…) a catch-all term to refer to any software designed to cause damage to a single computer, server, or computer network". It can be detected through Static and Dynamic analysis. One of the ways trough Static is Signature-based detection and one from Dynamic is the detection of abnormal behaviour. Some techniques were created to trick Static detection such as obfuscation, opaque predicates, etc. These techniques were enough to evade AV from some years ago, however more advanced anti-malware is performing more dynamic detection and even have AI implemented to increase its efficiency.

In report 2, I performed code obfuscation to try evading some AVs which partially worked since some stopped categorizing it has malware. Thenceforth, I used some automated cloacking tools that performed more AV evasions to that program. It did a good job for an automated tool, it modified more parts of the code therefore it contained fewer malware fingerprints. AVs are very efficient in protecting hosts against massive malware campaigns and known threats such as Trojans, rootkits, backdoors, etc (Rijnetu, 2017). With this knowledge I can understand why the AV classified the program even not being a malware, it's fingerprints and behaviour are very similar to one so it cause a false positive. Returning to the previous idea, the AVs' detection and defence fail when the malware is more sophisticated and complex such as zero-day attacks and APTs, such as Stuxnet.

It is important to understand how malware works how it tries to evade AVs and how they try to keep up with the malware evolution. This is and will continue to be a competition between attackers and security experts. Being in the cybersecurity field I will work with malware definitely either from the blue or red team.

3 Reverse Engineering

This is a very vast topic in which I explored a buffer overflow exploitation in section 3 and bypassed security measures/cracked and patched programs. Buffer overflow exploits cause corruption of data, crashes and code execution (OWASP, n.d.) and software cracking enables to remove or disable security features. Both are mainly used especially by attackers for obvious reasons while patching a program serves to update, fix and improve them.

In report 3, I disabled protection features to exploit a dangerous C function enabling to elevate privileges. The exploit had two steps, revere engineering the program to find the offsets and privilege escalation pointing the EIP to a shellcode that is written in opcode. There are a lot of opcodes that starts a shell however we have to be in account some details. The first one is having an opcode that is compatible with the computer architecture and the second one is if it is blocked by other security factors (defence in depth) such as the suid protection on scripts.

In report 4, I started to bypass login security reverse-engineering the program with GDB, a Unix debugger. Task 1 had the password in plain text which is a very bad practice, however, Task 2 generated a password adding some obfuscation to the program but adding some breakpoints solved the problem. Task 4 had a logic problem that was also fixed modifying the executable that was translated to assembly with radare2.

These exercises proved very clearly the importance of knowing Assembly which requires a lot of knowledge in how a computer works and its architecture. Revere engineering is used a lot in the industry to patch programs (i.e reverse engineering software to be compatible with new ARM CPUs), discover vulnerabilities and in software optimization. (Auerbach, n.d.)

Summing up, this portfolio enabled the consolidation of the lectured material applying the knowledge on practical examples. To accomplish the tasks research was made which expanded and deepened my knowledge about the subjects.

3 References

GeeksforGeeks. 2017. *Importance Of Function Prototype In C - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/importance-of-function-prototype-in-c/> [Accessed 15 November 2020].

Baeldung.com. 2020. [online] Available at: <https://www.baeldung.com/java-memory-leaks> [Accessed 15 November 2020].

Devlieghere, J., 2015. *Understanding The Clang AST*. [online] Jonas Devlieghere. Available at: <https://jonasdevlieghere.com/understanding-the-clang-ast/> [Accessed 21 November 2020].

Cadar, C., 2016. [online] Multicore.doc.ic.ac.uk. Available at: <http://multicore.doc.ic.ac.uk/SoftwareReliability/2016-2017/notes/11-KLEE-intro-and-demo.pdf> [Accessed 15 November 2020].

Brat, G., Navas, J. and Shi, N., 2020. *IKOS: A Framework For Static Analysis Based On Abstract Interpretation*. [online] ResearchGate. Available at:

<https://www.researchgate.net/publication/300368523_IKOS_A_Framework_for_Static_Analysis_Based_on_Abstract_Interpretation> [Accessed 15 November 2020].

Ampliasecurity.com. n.d. *Amplia Security - Research - WCE FAQ*. [online] Available at: <https://www.ampliasecurity.com/research/wcefaq.html> [Accessed 16 November 2020].

Docs.microsoft.com. 2020. *Next-Generation Protection In Windows 10, Windows Server 2016, And Windows Server 2019 - Windows Security*. [online] Available at: <https://docs.microsoft.com/en-gb/windows/security/threat-protection/microsoft-defender-antivirus/microsoft-defender-antivirus-in-windows-10> [Accessed 16 November 2020].

Cloonan, J., 2017. *Advanced Malware Detection - Signatures Vs. Behavior Analysis*. [online] Infosecurity Magazine. Available at: <https://www.infosecurity-magazine.com/opinions/malware-detection-signatures/> [Accessed 16 November 2020].

Mattson, J., 2010. *What Is The Difference Between A 32-Bit And 64-Bit Processor?*. [online] Stack Overflow Available at: <https://stackoverflow.com/questions/4552905/what-is-the-difference-between-a-32-bit-and-64-bit-processor> [Accessed 22 November 2020].

Qnx. n.d. *Help - QNX CAR 2 Documentation*. [online] Available at: <http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing_gdb_StackFrames.html> [Accessed 22 November 2020].

Anon. 2019. *Edit An .So File*. [online] Reverse Engineering Stack Exchange. Available at: <https://reverseengineering.stackexchange.com/questions/21061/edit-an-so-file> [Accessed 25 November 2020].

Gomulkiewicz, R. and Williamson, M., 1996. *The Problem Of Reverse Engineering*. [online] Kaner.com. Available at: <http://www.kaner.com/pdfs/reveng.pdf> [Accessed 25 November 2020].

Kundel, D., 2020. *Introduction To Abstract Syntax Trees*. [online] Twilio Blog. Available at: <https://www.twilio.com/blog/abstract-syntax-trees> [Accessed 26 November 2020].

Rouse, M., 2006. *What Is Dynamic Analysis? - Definition From Whatis.Com*. [online] SearchSoftwareQuality. Available at: <https://searchsoftwarequality.techtarget.com/definition/dynamic-analysis> [Accessed 26 November 2020].

Hearn, P., 2020. *Facebook's Code Checker - Computerphile*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=tKR2UZdRpV0> [Accessed 26 November 2020].

Slansky, D., 2020. *Understanding And Deploying Static Analysis*. [online] ARC Advisory Group. Available at: <https://www.arcweb.com/blog/understanding-deploying-static-analysis> [Accessed 26 November 2020].

Rijnetu, I., 2017. *These Campaigns Explain Why AV Detection For New Malware Remains Low*. [online] Heimdal Security Blog. Available at: <https://heimdalsecurity.com/blog/campaigns-av-detection-new-malware-low/> [Accessed 26 November 2020].

Owasp.org. n.d. *Buffer Overflow | OWASP*. [online] Available at: <https://owasp.org/www-community/vulnerabilities/Buffer_Overflow> [Accessed 26 November 2020].

Auerbach, J., n.d. *6 Benefits Of Reverse Engineering - CEDARVILLE Engineering Group, LLC – Sustaining Communities By Design*. [online] Cedarvilleeng.com. Available at: <http://www.cedarvilleeng.com/news-and-insights/blog/6-benefits-of-reverse-engineering/> [Accessed 26 November 2020].

Bollinger, J., 2018. *C - Ignore Spaces In Scanf()*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/51556986/c-ignore-spaces-in-scanf> [Accessed 26 November 2020].