Software security lab01

Davide Caria
Tee Wei Teoh
Sara Lehne Engesvik

11 October 2023

1 Code analysis of exercise n2: WinLoose

1.1 Point 1

The program has been compiled with the following command:

```
gcc -fno-stack-protector -o winloose winloose.c
```

and tested with different inputs for the *./winloose* file. As an example we tried with several inputs and wrote down the outputs.

- ./winloose 3 3
- ./winloose 10 10
- ./winloose 3 18
- ./winloose 30 7
- ./winloose 5 8
- ./winloose 0 11
- ./winloose 1 8

1.2 Point 2

According to the previous list of possible inputs we deduced the following:

- $argv[2] > 8$ the program crashes into a segmentation fault error
- $argv[2] < 8$ the output is always $Youloose...$
- $argv[2] = 8$ and $argv[1] = 0$ the output is $Youloose...$
- $argv[2] = 8$ and $argv[1] != 0$ the output is always $YouWin!$

The guessing of the numbers suggest that something makes the program crash when the second argument is above 8, looking at the C code we can see that this is due to a buffer overflow in line 15. If the program does not crash or loop endlessly it can produce two outputs which depend on the value of the variable X. Specifically, there are no boundary checks of the input given by the user. Therefore, if we use a value that exceeds the array length we step into the X variable location and overwrite it. This means that if the first argument is different from zero, in the last iteration it will become the new value of X making us win the game.

1.3 Point 3

To confirm our guesses we disassembled the code using the *objdump* tool. The specific disassembled code on the lab machine was the following:

```
0000000000001169 <main>:
    1169:       f3 0f 1e fa             endbr64
    116d:       55                      push   %rbp
    116e:       48 89 e5                mov    %rsp,%rbp
    1171:       48 83 ec 20             sub    $0x20,%rsp
    1175:       89 7d ec                mov    %edi,-0x14(%rbp)
    1178:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
    117c:       c6 45 fb 00             movb   $0x0,-0x5(%rbp)
    1180:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
    1187:       eb 22                   jmp    11ab <main+0x42>
    1189:       48 8b 45 e0             mov    -0x20(%rbp),%rax
    118d:       48 83 c0 08             add    $0x8,%rax
    1191:       48 8b 00                mov    (%rax),%rax
    1194:       48 89 c7                mov    %rax,%rdi
    1197:       e8 d4 fe ff ff          call   1070 <atoi@plt>
    119c:       89 c2                   mov    %eax,%edx
    119e:       8b 45 fc                mov    -0x4(%rbp),%eax
    11a1:       48 98                   cltq
    11a3:       88 54 05 f3             mov    %dl,-0xd(%rbp,%rax,1)
    11a7:       83 45 fc 01             addl   $0x1,-0x4(%rbp)
    11ab:       48 8b 45 e0             mov    -0x20(%rbp),%rax
    11af:       48 83 c0 10             add    $0x10,%rax
    11b3:       48 8b 00                mov    (%rax),%rax
    11b6:       48 89 c7                mov    %rax,%rdi
    11b9:       e8 b2 fe ff ff          call   1070 <atoi@plt>
    11be:       39 45 fc                cmp    %eax,-0x4(%rbp)
    11c1:       7e c6                   jle    1189 <main+0x20>
    11c3:       80 7d fb 00             cmpb   $0x0,-0x5(%rbp)
    11c7:       74 11                   je     11da <main+0x71>
    11c9:       48 8d 05 34 0e 00 00    lea    0xe34(%rip),%rax        # 2004 <_IO_stdin_used+0x4>
    11d0:       48 89 c7                mov    %rax,%rdi
    11d3:       e8 88 fe ff ff          call   1060 <puts@plt>
    11d8:       eb 0f                   jmp    11e9 <main+0x80>
    11da:       48 8d 05 2d 0e 00 00    lea    0xe2d(%rip),%rax        # 200e <_IO_stdin_used+0xe>
    11e1:       48 89 c7                mov    %rax,%rdi
    11e4:       e8 77 fe ff ff          call   1060 <puts@plt>
    11e9:       b8 00 00 00 00          mov    $0x0,%eax
    11ee:       c9                      leave
    11ef:       c3                      ret
```

Figure 1: Assembly code for *winloose* using *objdump*

Then, after analyzing the output of *objdump* we worked out a possible memory stack structure, including the location of the variables and arrays. We assumed the following configuration:

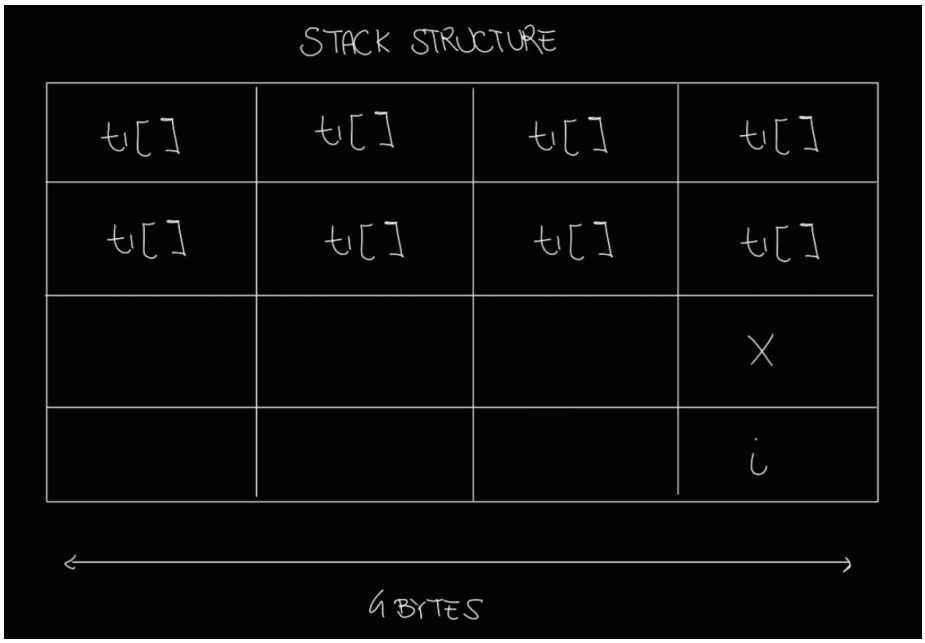


Figure 2: Possible stack structure for *winloose*

According to the behaviour of the code, and the output of *objdump* we can imagine that the X variable is just under the array allocation space so when the correct input is provided, there is a buffer overflow. The configuration of the stack could be different depending on the system that we run the code on, the X variable can be placed all the way to the left and the behaviour would still be the same.

1.4 Point 5

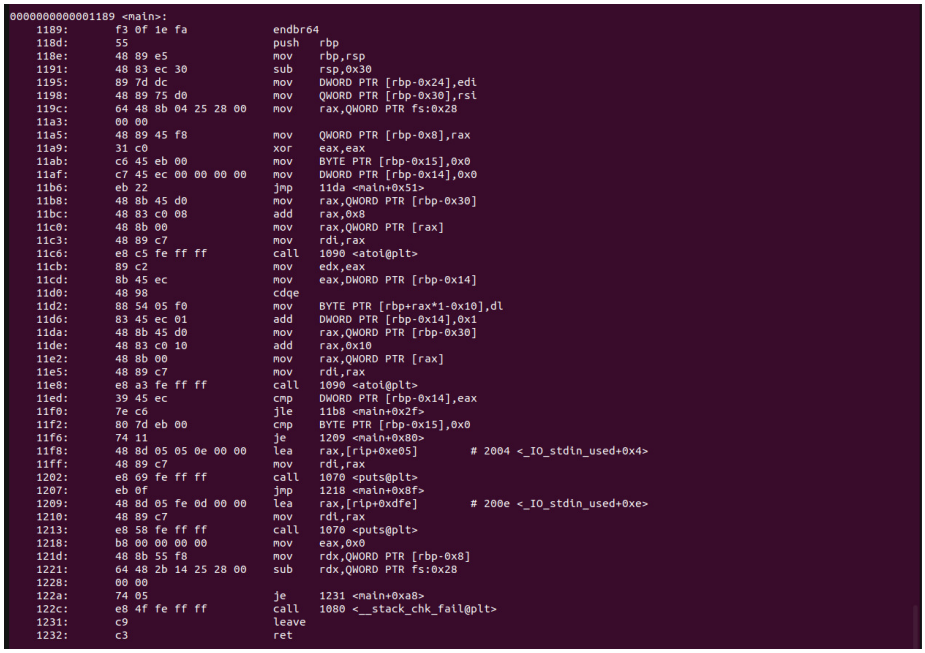
After running the compilation with the stack protection option as follows:

```
gcc -fstack-protector -o winloose winloose.c
```

we tested the list of previous inputs to discover a different behaviour. Using a valid input from Point 1, such as (5,8), we end up with a *stack smashing error*. This suggests that we are not allowed to overflow the buffer *t1* defined in the program. This results in a crush of the executable. This behaviour was expected since the $-fstack-protector$ is enabled and it is supposed to stop the execution as soon as a buffer overflow is detected.

1.5 Point 6

After disassembling the protected code with *objdump* tool we obtained the following assembly instructions:



```
0000000000001189 <main>:
    1189:       f3 0f 1e fa             endbr64
    118d:       55                      push   rbp
    118e:       48 89 e5                mov    rbp,rsp
    1191:       48 83 ec 30             sub    rsp,0x30
    1195:       89 7d dc                mov    DWORD PTR [rbp-0x24],edi
    1198:       48 89 75 d0             mov    QWORD PTR [rbp-0x30],rsi
    119c:       64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
    11a3:       00 00
    11a5:       48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
    11a9:       31 c0                   xor    eax,eax
    11ab:       c6 45 eb 00             mov    BYTE PTR [rbp-0x15],0x0
    11af:       c7 45 ec 00 00 00 00    mov    DWORD PTR [rbp-0x14],0x0
    11b6:       eb 22                   jmp    11da <main+0x51>
    11b8:       48 8b 45 d0             mov    rax,QWORD PTR [rbp-0x30]
    11bc:       48 83 c0 08             add    rax,0x8
    11c0:       48 8b 00                mov    rax,QWORD PTR [rax]
    11c3:       48 89 c7                mov    rdi,rax
    11c6:       e8 c5 fe ff ff          call   1090 <atoi@plt>
    11cb:       89 c2                   mov    edx,eax
    11cd:       8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    11d0:       48 98                   cdqe
    11d2:       88 54 05 f0             mov    BYTE PTR [rbp+rax*1-0x10],dl
    11d6:       83 45 ec 01             add    DWORD PTR [rbp-0x14],0x1
    11da:       48 8b 45 d0             mov    rax,QWORD PTR [rbp-0x30]
    11de:       48 83 c0 10             add    rax,0x10
    11e2:       48 8b 00                mov    rax,QWORD PTR [rax]
    11e5:       48 89 c7                mov    rdi,rax
    11e8:       e8 a3 fe ff ff          call   1090 <atoi@plt>
    11ed:       39 45 ec                cmp    DWORD PTR [rbp-0x14],eax
    11f0:       7e c6                   jle    11b8 <main+0x2f>
    11f2:       80 7d eb 00             cmp    BYTE PTR [rbp-0x15],0x0
    11f6:       74 11                   je     1209 <main+0x80>
    11f8:       48 8d 05 05 0e 00 00    lea    rax,[rip+0xe05]        # 2004 <_IO_stdin_used+0x4>
    11ff:       48 89 c7                mov    rdi,rax
    1202:       e8 69 fe ff ff          call   1070 <puts@plt>
    1207:       eb 0f                   jmp    1218 <main+0x8f>
    1209:       48 8d 05 fe 0d 00 00    lea    rax,[rip+0xdfe]        # 200e <_IO_stdin_used+0xe>
    1210:       48 89 c7                mov    rdi,rax
    1213:       e8 58 fe ff ff          call   1070 <puts@plt>
    1218:       b8 00 00 00 00          mov    eax,0x0
    121d:       48 8b 55 f8             mov    rdx,QWORD PTR [rbp-0x8]
    1221:       64 48 2b 14 25 28 00    sub    rdx,QWORD PTR fs:0x28
    1228:       00 00
    122a:       74 05                   je     1231 <main+0xa8>
    122c:       e8 4f fe ff ff          call   1080 <__stack_chk_fail@plt>
    1231:       c9                      leave
    1232:       c3                      ret
```

Figure 3: Assembly code for *winloose* with stack protection

We can see that there are some differences in how the stack is protected in this new version of the assembly code: First there has been given more space to the stack and the buffers that are allocated. Then the compiler inserted more *cmp* statements to check if we are exceeding the boundaries. Specifically, a canary value is initialized by the compiler to a random value and placed where the buffer overflow can happen. At the end of the loop the canary value is checked again and if it is different with respect to the original value the program jumps to a routine. The routine is $stack\_chk\_fail$ and it handles the buffer overflow that just occurred.

2 Code analysis of exercise n4: ExecShell

2.1 Point 1

After analyzing the the provided c code, we found three vulnerabilities:

- No input sanitization
- Buffer-overflow

- Use-after-free

2.1.1 No input sanitization

For what concerns input-sanitization, we can see that there is no protection or check that prevents the insertion of wrong values. Assuming that a correct user is trying to operate the tool, he has no boundaries neither on the size nor in the content. This means, as example, that a string that does not represent a directory is considered valid.

2.1.2 Buffer-overflow

There is an attempt to limit the possibility of buffer-overflow at line 22:

```
if (strlen(a) > 14) { ... }
```

Nevertheless, this does not solve the problem completely and leaves the chance of buffer-overflow later in the code. Specifically, at line 36 we can see that a new buffer is allocated with slightly more space compared to the first allocation.

```
p3 = (char *) malloc (24 * sizeof (char));  // allocation 2
```

This may help with some standard use cases, for example when there are small paths as inputs. However, any input longer than 24 characters will result in an overflow.

2.1.3 User-after-free

This last vulnerability presents itself when we have an input larger than 16 character. In line 24 the program frees the pointer to the buffer p:

```
free (p);
```

and later in the code the pointer p is used again as argument for the system call $system()$. This is the perfect example of use-after-free vulnerability. In this particular case, the vulnerability is combined with wrong code logic: the pointer p refers only to the original buffer so even if the user insert a valid string during the second trial, the program will still execute:

```
system(p)
```

ignoring the new allocated string $p3$.

2.2 Point 2

A possible way to execute an arbitrary shell command can leverage the character ; which concatenates multiple command. As an example we could provide a valid path for the ls tool but concatenate a malicious code.

```
input: /tmp ; rm -r /
```

This input will be concatenated with the predefined string in the program to form a valid shell command:

```
$ ls /tmp ; rm -r /
```

This command will list all the elements under */tmp* directory and subsequently it will try to wipe the disk from the root folder /. Most probably the command will not work thanks to the *admin* write restriction. Nevertheless, other subfolder could be deleted if the read-write policy is not properly configured.

3 Sub-rule explanation

The team choose *Rule* 08 and sub-rule $MEM30 - C$ from the CERT Secure Coding Standard.

RULE	SEVERITY	LIKELIHOOD	REMEDIATION COST
MEM30-C	High	Likely	Medium

The given rule, MEM30-C, focuses on avoiding access to freed memory, which can lead to undefined behavior and exploitable vulnerabilities. It outlines the risks associated with accessing or modifying memory that has been deallocated using functions like *free*() or *realloc*(). The main problems addressed by the rule include reading or writing to memory after it has been freed and the potential for double-free vulnerabilities

3.1 Problem 1: Accessing Freed Memory

Accessing a pointer to memory that has been deallocated (dangling pointer) is undefined behavior and can lead to exploitable vulnerabilities. Reading or writing to this memory can result in abnormal program termination and denial-of-service attacks

3.2 Problem 2: Double-Free Vulnerabilities

Freeing memory multiple times, either directly or through incorrect usage of *realloc*(), can cause double-free vulnerabilities. Double-free can lead to the execution of arbitrary code with the permissions of the vulnerable process.
An example of **non compliant code**:

```
#include <stdlib.h>

void f(char *c_str1, size_t size) {
    char *c_str2 = (char *)realloc(c_str1, size);
    if (c_str2 == NULL) {
        free(c_str1);
    }
}
```

An example of **compliant code**:

```
#include <stdlib.h>

void f(char *c_str1, size_t size) {
    if (size != 0) {
        char *c_str2 = (char *)realloc(c_str1, size);
        if (c_str2 == NULL) {
            free(c_str1);
        }
    }
    else {
        free(c_str1);
    }
}
```

In the non-compliant code, the *realloc*() function is called with a size argument of zero, which can lead to potential issues. According to the C standard, when *realloc*() is called with a size of zero, it may behave like $free()$ or return a unique pointer that can be used for future allocations. If the returned pointer is used and the original pointer (c_str1) is already de-allocated, this can lead to undefined behavior. In the compliant code, a check for $size! = 0$ is added before calling *realloc*(). This ensures that *realloc*() is only called when the size is non-zero, addressing the issue and preventing potential undefined behavior associated with calling *realloc*() with a size of zero.

3.3 Solution for problem 1 (Accessing Freed Memory)

Ensure that memory is not accessed (read or written) after it has been freed. Properly manage the order of operations to prevent such access. Free the memory after its final use to avoid any potential access to the freed memory

3.4 Solution for problem 2 (Double-Free Vulnerabilities)

Properly handle memory deallocation to prevent double-free vulnerabilities. Ensure that memory is freed only once and is not attempted to be freed again