Peter Amthor [eMail]
Department of Computer Science and Automation
Distributed Systems and Operating Systems Group

TECHNISCHE UNIVERSITÄT
ILMENAU

Exercises 2

Buffer Overflow Attacks

In the lectures, you saw in principle how a buffer overflow attack on a program that does not check the length of its input is done: By extending an input string beyond the intended size of the buffer, an attacker may overwrite important information in memory such as the return address of a function.

Now you will conduct a buffer overflow attack yourself. Your victim will be a legacy Linux program (x86-32) that is provided on moodle (`victim.tar.gz`).[1] The tools you will use, a disassembler (`objdump`), a debugger (`gdb`) and a compiler (`gcc`), are described in detail at the end of this document. As *an exception for this exercise*, all teams solve the same assignment in steps of advancing difficulty. During the seminar you should explain your solution and compare it with the other teams. Start with the first step and proceed to each next one as soon as you have a solution.

Please refer to the hints at the end of the document as well as to the example given in the lecture (chapter 1).

Basics:

Once extracted (command "`tar -zxvf victim.tar.gz`"), run the victim program via "`./victim`" on command line. You should now see a prompt for a password:

```
Password (max. 12 characters):
```

that you may just skip using `<Enter>` for now. You will see the message "`Could not authenticate user!`". Your goal is to bypass this authentication mechanism.

To this end, you will cause a buffer overflow in the program's stack, which is written "top-down" from the higher towards the lower memory addresses. Whenever a function is called, its parameters are pushed on the stack, followed by the current contents of the *instruction pointer* and *base pointer* registers (usually called `EIP` and `EBP`). Thus, these registers may be restored when leaving the function. After that, the function reserves some memory on the stack for its local variables.

Your attack will exploit the fact that, due to this stack buildup, the saved instruction pointer is placed in memory *behind* the buffer of a local variable. Its value may thus be overwritten by a buffer overflow. In this way, when the function eventually returns, the attacker can force the execution flow to jump to an arbitrary address instead of the original one.

[1] It runs on every modern 64 bit kernel. You may use VirtualBox etc. to set up an Ubuntu or Mint Linux if needed.

Step 1:

Your first step will aim at just overwriting the return address in order to "call" another function. What you are doing is, however, not a real call but merely a jump to the first instruction of the function. (What is the difference?)

The `victim` program reads your input via the function `password_prompt`, which was compiled from the following source code in C:

```
void password_prompt(void) {
    char buf[12];
    gets(buf);
    strncpy(password, buf, 12);
}
```

The called functions `gets` and `strncpy` are implemented in the standard C library (`libc`) under Linux and documented in chapter 3 of the command line manual. Now construct an attack that overwrites the return address of the `password_prompt` stack frame to jump to the `memtest` function, which is used internally for debugging. This will terminate the program after the execution of `memtest`.

For planning your attack, refer to the descriptions at the end of this document. Use the debugger to obtain all the information you need. Your actual attack should be as well conducted in the debugger, otherwise memory addresses may change. Further, pay attention to the correct (reverse) byte order: an address `0x89abcdef` is written as `0xefcdab89` in the input. Since `gets` quits reading from the keyboard as soon as there is a line break (`0x0A`) or end-of-file-byte (`0x04`) occurring, you should avoid these characters in your input!

This assignment is solved when `memtest` shows his output:

```
Return address: 0x...
Frame address:  0x...
```

Step 2:

Parameters of a function are passed by pushing them on the stack before calling the function. Thus, they can also be manipulated with a buffer overflow attack. Use a disassembler to find out the actual position of the parameter you want to overwrite.

Now try to gain access to the restricted part of the `victim` program! The function `grant_access` simulates the execution of such an access, however requiring one parameter: The user id (uid) of the user to grant access. To find out your own uid, use the command "`id -ur`". You may convert it into a hexadecimal number on the command line, using "`printf '%x\n' 500`".

The `grant_access` function was compiled from the following C code:

```
void grant_access(unsigned int uid) {
    unsigned int current_uid = get_user_id();
    if(uid != current_uid) {
        puts("Wrong user ID!");
        exit(1);
    }
    grant = 1;
    grant_uid = uid;
```

```
        puts("+++ Access granted +++");
    }
```

This assignment is solved when the message "`Access granted`" appears. A subsequent program crash is normal because of the corrupted stack your attack has left. (Why?)

Step 3:

As long as the program crashes once you have gained access, your attack is useless. Therefore, construct an attack that leaves the stack undamaged. To achieve this, execute your own code on the stack during a buffer overflow attack.

To inject executable code, you have to write a program in assembly language and then compile it into machine language. Within this program, you should place a return address that points to the beginning of your own code (which should be always the same address if executed in the debugger). Once you manage to make the program jump to your own code, you may use it to grant yourself access and repair the stack so a correct return into `password_prompt` is performed when the execution of `grant_access` is finished!

Because of some technical details, you have a maximum of 20 bytes available for your code – that should be enough to prepare the parameter to be passed, place a valid pointer to your intended return address and perform the jump into `grant_access`. The easiest way to jump to a specific address is pushing the target address on the stack and calling the `ret` assembler instruction.

Remember that a consistent stack layout requires the stored values of the `EIP` and `EBP` registers and a correct value of the stack pointer (`ESP` register).

This assignment is solved when the messages "`Access granted`" and "`Authentication complete.`" appear and the program exits normally.

Step 4:

In assignment 3 you have executed your attack code by manipulating the return address to point to the beginning of your code. This is possible because the `victim`-program keeps the position of the stack in memory stable. Under normal circumstances, however, the address of the stack depends on various factors and usually changes on every run of the program.

How could an attacker nevertheless execute his/her code on the stack?

Step 5:

You have now gained your first experiences with buffer overflow attacks. Discuss which requirements are necessary to perform such an attack!

During a buffer overflow attack functions may be called or even own code executed. Which results are possible for an attacker to achieve this way?

Tools and Hints:

To create your attack, write your input for the program in a hexadecimal editor (e.g. `hexedit` or `bless` in any major Linux distribution). Save it as a binary file and redirect it into `victim` as your input, e.g.:

```
[user@host]$ touch exploit.bin     (create file)
[user@host]$ hexedit exploit.bin     (edit file)
...
[user@host]$ ./victim < exploit.bin  (execute victim)
```

Keep in mind to use Little-Endian byte order!

To understand how `victim` works, you will need the source code of the program. You can obtain it by using the disassembler `objdump`:

```
[user@host]$ objdump -d victim > source.dump
```

and displaying the result in any text viewer/editor, such as `less`:

```
[user@host]$ less source.dump
```

To analyze the program at runtime, use the `gdb` debugger. Call it with your executable file as first argument and then use the `gdb` command line. Listed below is an exemplary session with explanatory comments:

```
[user@host]$ gdb victim     (run the debugger)
(gdb) break password_prompt (set a break point at password_prompt)
Breakpoint 1 at 0x80486de
(gdb) run < exploit.bin     (run the program with the specified input)
Starting program

Breakpoint 1, 0x080486de in password_prompt()
(gdb) print/x $esp          (show contents of the ESP-register
                             in hexadecimal)
$1 = 0xabcdef10
(gdb) x/24w $esp            (examine memory: 24 words starting from the
                             address in ESP)
0xabcdef10:    0x00000000  0x00000000
...
(gdb) x/i $eip              (examine instruction at the address contained
                             in EIP)
0x80403020:    movl   $0x0,%eax
(gdb) continue              (continue running)
...
```

You may use the built-in help function of `gdb`: just type `help`.

In order to inject your own code, write it in assembly language and compile it into machine code using `gcc`:

```
[user@host]$ vim exploit.s (or any other editor)
...
[user@host]$ gcc -m32 -c exploit.s
```

Then you may just copy the resulting byte string into your input – use `objdump` to get it:

```
[user@host]$ objdump -d exploit.o
```

Keep in mind that gcc expects AT&T assembler syntax (operand order: first source, second destination). Instructions such as mov or push are usually written with a suffix, indicating the size of data; usually an "l" is used for "long"-values (32 bit here). Here is a valid sample-program for gcc:

```
.text
movl $0xa,(%esp)  # replace topmost stack entry by 0xa
pop %eax          # pop top of stack into EAX
mov %eax,0xabcd   # write content of EAX to address 0xabcd
```

Some programs or libraries used here may have to be installed on your system first. However, the recommended tools should be present by default in most of the current Linux distributions. Further information may be found on the internet.

Please refer also to the manuals of gdb and gcc ("man 1 ..") as well as these online manuals:
http://sourceware.org/gdb/current/onlinedocs/gdb/
http://sourceware.org/binutils/docs-2.17/as/