Operating Systems Security – Assignment 4

Version 1.0.0 – 2014/2015

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

1 Heap buffer overflow

In this exercise we demonstrate that operating system and compiler mitigation measures are not always sufficient to counter all buffer overflow attacks. The **C++** program we use is vulnerable program to a string buffer overflow. However, in modern operating systems it can be protected by:

- Address Space Layout Randomization (ASLR)
- Data Execution Prevention (DEP)
- Stack Smashing Protector (SSP)

Prerequisites

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

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

class Command {
protected:
    const char* exec;
    const char* name;
    char password[256];
public:
    Command() { printf("New instance of Command() at address: %p\n",this); }
    void setPassword(const char* password) { strcpy(this->password,password); }
    bool checkpass() {
        const char *hash1, *hash2;
        hash1 = crypt(password,"$6$1122334455667788$");
        hash2 = "$6$1122334455667788$vDzpRFs0Pl/L0M4/WXWsmv5/eTYlh5xoA"
                "lMoPy512JiBLrAZTNzbL.uWv3ZI6XxFUYnFzRIX2kGXF9M133D4h1";
        if (strcmp(hash1,hash2) == 0) {
            return true;
        } else {
            printf("Error: incorrect password\n");
            return false;
        }
    }
    virtual int run() { return system(this->exec); }
    const char* getName() { return name; }
};

class Date: public Command {
public:
    Date() {
        this->name = "date";
        this->exec = "/bin/date";
    }
};

class Shell: public Command {
public:
    Shell() {
        this->name = "shell";
        this->exec = "/bin/sh";
    }
    int run() {
        if (this->checkpass()) {
```

```
            return Command::run();
        } else {
            return -1;
        }
    }
};

int main(int argc, char *argv[]) {

    const char* password;

    if (argc == 2) {
        password = "";
    } else if (argc == 3) {
        password = argv[2];
    } else {
        printf("syntax: %s <date|shell|...> [password]\n",basename(argv[0]));
        return 1;
    }

    Command* cmds[2] = { new Date(), new Shell() };

    for (size_t i=0; i<2; i++) {
        cmds[i]->setPassword(password);
        if (strcmp(argv[1],cmds[i]->getName()) == 0) {
           cmds[i]->run();
        }
    }

    return 0;
}
```

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

$ **g++ -Wall -g -O3 -o cmd cmd.cpp -lcrypt -fstack-protector-all -fpie**

$ **sudo chown root:root cmd**

$ **sudo chmod u+s cmd**

Open a text editor and create the file **bruteforce.sh** with the following contents:

```
#!/bin/bash
i=0
while [ $i -lt 65536 ]; do
#while [ $i -lt 10 ]; do
   let i=i+1
   echo "The counter is $i";
   echo "./cmd shell " $(python -c "print \"A\"*$i");
   ./cmd shell $(python -c "print \"A\"*$i")
done
```

Make the script executable with

$ **chmod a+x bruteforce.sh**

Objectives

a) What is a heap buffer overflow and how is it different from a stack overflow?

b) Explain briefly what the program is doing and what argument you need supply to execute a certain action.

c) Execute the script **bruteforce.sh** and figure out how many 'A' characters need to be concatenated and given as an argument to get a shell.
   $ **./bruteforce.sh**
   Explain why the value might be different than the string buffer size + 1.

d) Start the debugger and execute it with the same value in **###** as argument
   $ **gdb -q cmd --ex "b main" --ex "r shell $(python -c 'print "A"*###')"**
   Use the *next instruction* (**ni**), *step instruction* (**si**) and *examine memory* (**x**) to figure out what is happening. Explain in detail which address gets overwritten and why this influences the control flow and execution path of the program.

e) Explain for each mitigation technique (ASLR, DEP, SSP) what it does and why it does not mitigate this attack.

f) What mitigation techniques could operating systems and compilers offer to avoid such attacks. Evaluate the impact of your suggestion on performance, use of resources and flexibility. For instance, could you re-compile the linux kernel (and all libraries) with your solution without significant impact?

2 Covert channels

An operating system tries to avoid information leakage between processes which are executed by different users. However, it is not always capable of identifying suspicious behaviour, especially if the processes use generic information leakage channels such as:

- Existence of a file
- File attributes
- CPU usage
- Temperature sensor
- "Disk full" errors

Objectives

a) Write two (simple) programs that communicate messages to each other using a covert information leakage channel that is not (inherently) identified by the operating system as communication channel between processes. Hand in the source-code and explain how it should be installed/used.

b) Execute the programs under two different (**non**-root) users and let it communicate to one another. Then, open the log files of your (Kali) Linux system and see if the communication leaves any visible trace (such as unusual and suspicious errors). Explain why your method is undetectable or how it could be optimized to avoid detection by operating systems that perform more advanced monitoring.