Software Security
Assignment 3: Buffer Overflow

Deadline: Tuesday, 23/12/2024

1. **תאריך הגשה: 23.12.2024 בשעה 23:55 למטלה הקשורה ב-Moodle בלבד.**
2. **אין להגיש בשום פנים ואופן למייל של מרצה או מתרגל - אך ורק ב-Moodle.**
3. **דחיית העבודה ניתנת רק במקרה של מילואים או אישור מחלה. יש להגיש בקשת סטודנט בצירוף המסמכים. אין לפנות במייל למתרגל או למרצה בבקשת דחיית העבודה!**
4. **ניתן להגיש את העבודה בזוגות בלבד.**
5. **לא יתקבלו עבודות שהוגשו באיחור.**
6. **תאריך הגנה יתפרסם בהמשך.**
7. **במקרה של העתקה מלאה או חלקית של העבודה (מסטודנטים אחרים, מ-Internet או מכל מקום אחר), יינתן ציון 0 על העבודה של כלל הסטודנטים המעורבים והם יעלו לוועדת משמעת. במקרה שהתגלתה העתקה, אנו שומרים את הזכות לבדוק העתקות גם בעבודות קודמות.**

In this assignment, you will use controlled buffer overflows in order to exploit a program bug.

The programs below are intended to be compiled in Visual Studio 2015, as Win32 Console Application for **x64 (64-bit) platform**. *Release* build should be configured with *Minimize Size* optimization level in project settings. For *Debug* build, *Basic Runtime Checks* and *Security Check* must be disabled.[1] You must use these exact project settings in VS2015.

You may use a different architecture / compiler, but may need to similarly adjust security checks (e.g., -fno-stack-protector switch in *gcc* under Linux).

You should submit an archive with all relevant code and a *.pdf* with running examples for each task and detailed explanations of your approach to solving the tasks.

Task 1

A programmer wrote the following program (see Appendix 1), which checks a password in a corresponding routine, and if the check is successful, runs some sensitive code in a dedicated function.

During testing, the programmer discovered that a long (but not too long) password will cause the password check to succeed, even though the password is incorrect. For instance:

```
Address of pwd:    00000000001DFB30
Address of result: 00000000001DFB34

Enter password: donkey
Password check is successful.
Doing authenticated things...
Done.
```

You should:

1. Explain: why does the check succeed?

[1] You may find it preferable to put the main function into a separate main.c file.

2. Show and explain contents of relevant stack locations inside the check function (e.g., use *Debug* → *Windows* → *Memory* menu in Visual Studio, or *gdb* commands in Linux).
3. Explain the importance of relative order of pwd and result local variables on stack during execution.
4. Does swapping the pwd and result local variable source code rows help? Why, and what do you think is the reason?
5. For a much longer password, the program ultimately crashes, although the password check still succeeds. Explain.
6. The programmer decided to mitigate the bug by adding an else result = 0; line. Does the "exploit" still work? Explain.

Task 2

We now wish to exploit the modified program (with the added else line) — see Appendix 2. This task is best done using *Debug* build.

In order to easily specify any binary input, we support %XX sequences in the input (percent-encoding), where XX is a per-byte hexadecimal notation. The input is decoded in-place. For instance, NUL-terminated don%09key input becomes don*[Tab]*keyey, where *[Tab]* is the byte 0x09, and the remaining NUL-terminated ey suffix is unaffected after rewriting the string.

Also, for simplicity, we will assume to know the address of an **unrelated** code point in the program — e.g., the address of task2 function.

You should design a method for giving the program a specific input (without running the debugger every time), which, although not being the correct password, will allow the attacker to run the sensitive dostuff function. It is fine if the program crashes afterwards.

The attack should cause the check function to "return" directly to the beginning of dostuff function.

You should:

1. Explain: how is this attack supposed to work?
2. Explain how you intend to compute the injected address, given a known code point address above. You cannot simply print the address of dostuff function!
3. Show the code that you added to the program in order to assist in the computation above (e.g., printing results of some pointer arithmetic). The purpose of this code should be to produce some static information about the program — for instance, distance between addresses of functions (which is constant for a given compiled program).

   Alternatively, show the debugging process that you used to produce this information. Again, the information must be static — you cannot run the debugger every time you want to exploit the program.
4. Show two successful execution examples, along with the corresponding computations that you performed. The input must be deduced from the static information computed above, along with the known dynamic address of task2 function that is shown during

execution. I.e., you cannot just enter an address that you see in a debugger, because that would require running the debugger for each exploit.

Task 3

The private keyword in almost all programing languages does not actually provide any layer of protection, and is strictly a means for developers to keep track of encapsulation. By knowing the structure of a class (the layout of its member fields), a developer can maliciously access and modify any member field (private or public) by using pointers to directly access the memory.

Using the code below (Appendix 3), and without changing any function other than malicious, do the following:

1. Print the credit card number to the screen.
2. Change the credit card limit to 100,000.
3. Change the pin-code to 1,234.
4. Print all the updated values.

Do not change this function's signature!

If you use the debugger, add the information that you took advantage of (such as class memory layouts) to the submitted document.

Explain whether such an exploit is possible in other languages like Java, which uses a virtual machine to run code.

Good luck!

Appendix 1

File: task1.c

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>

static int check() {
	char pwd[4];
	int  result = 0;

	printf("Address of pwd:    %p\n"
	       "Address of result: %p\n\n",
	       pwd, &result);

	printf("Enter password: ");
	scanf("%s", pwd);

	if (strcmp(pwd, "ok") == 0) {
		printf("Password is correct!\n");
		result = 1;
	}

	return result;
}

static void dostuff() {
	printf("Doing authenticated things...\n");
}

int task1() {
	if (check()) {
		printf("Password check is successful.\n");
		dostuff();
	}

	printf("Done.\n");
	return 0;
}

int main() {
	return task1();
}
```

Appendix 2

File: task2.c

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// computes the value of a hexadecimal digit
static int hexvalue(char c) {
      int value = -1;

      if (isxdigit(c)) {
            if (isdigit(c))
                  value = c - '0';
            else if (isupper(c))
                  value = c - 'A' + 10;
            else
                  value = c - 'a' + 10;
      }

      return value;
}

// converts percent-encoded string to binary string in-place
static void decode(char* s) {
      for (char *t = s;  *s;  t++, s++) {
            if (*s == '%'  &&  isxdigit(s[1]) && isxdigit(s[2])) {
                  *t = hexvalue(s[1]) * 16 + hexvalue(s[2]);
                  s += 2;
            }
            else
                  *t = *s;
      }
}
```

```c
static int check() {
	char pwd[4];
	int  result = 0;

	printf("Enter password: ");
	scanf("%s", pwd);

	decode(pwd);
	printf("Decoded to: %s\n", pwd);

	if (strcmp(pwd, "ok") == 0) {
		printf("Password is correct!\n");
		result = 1;
	}
	else
		result = 0;

	return result;
}

static void dostuff() {
	printf("Doing authenticated things...\n");
}

int task2() {
	printf("Hello from task2() at %p!\n", task2);

	if (check()) {
		printf("Password check is successful.\n");
		dostuff();
	}

	printf("Done.\n");
	return 0;
}

int main() {
	return task2();
}
```

Appendix 3

File: task3.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>

namespace {
      class CreditCard {
            friend std::ostream& operator<<(std::ostream&, const CreditCard&);

      private:
            char number[17];
            char expiration[5];
            int  pin;
            int  creditLimit;

      public:
            CreditCard(char* number, char* expiration, int pin, int creditLimit)
                  : pin(pin), creditLimit(creditLimit) {
                std::strncpy(this->number,     number,     sizeof(this->number));
                std::strncpy(this->expiration, expiration, sizeof(this->expiration));
                this->number    [sizeof(this->number)     - 1] = '\0';
                this->expiration[sizeof(this->expiration) - 1] = '\0';
            }
      };

      // For debugging purposes only -- do not use in malicious()
      std::ostream& operator<<(std::ostream& out, const CreditCard &cc) {
            return out << "[DEBUG] " << cc.number << ' ' << cc.expiration
                       << ' ' << cc.pin << ' ' << cc.creditLimit;
      }

      void malicious(const CreditCard &cc) {
            // Print the 'private' credit card number

            // Change the credit limit to 100,000

            // Change the PIN to 1234

            // Print all the updated values
      }
}

extern "C"
int task3() {
  CreditCard cc("3717422122321221", "0805", 4321, 1000);
  std::cout << "Before: " << cc << std::endl;

  malicious(cc);
  std::cout << "After:  " << cc << std::endl;

  return 0;
}
```