William Hupp
Reversing III
CIS5930

Reversing II Write-Up

Install the Tools Part 3

10 points

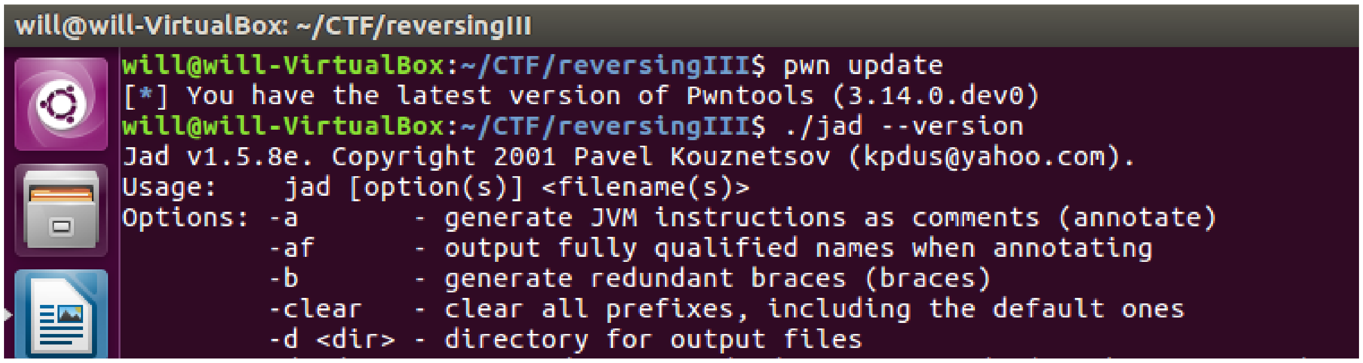
Problem:

Provide a screen shot that you have the latest versions of the following tools installed. Install instructions are located in the slides.

- pwntools
- jad

flag{f1ag_f1ag_f1ag_f1ag}

Solution:



```
will@will-VirtualBox:~/CTF/reversingIII$ pwn update
[*] You have the latest version of Pwntools (3.14.0.dev0)
will@will-VirtualBox:~/CTF/reversingIII$ ./jad --version
Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov (kpdus@yahoo.com).
Usage:    jad [option(s)] <filename(s)>
Options: -a       - generate JVM instructions as comments (annotate)
         -af      - output fully qualified names when annotating
         -b       - generate redundant braces (braces)
         -clear   - clear all prefixes, including the default ones
         -d <dir> - directory for output files
```

flag{f1ag_f1ag_f1ag_f1ag}

Journal III

10 points

Problem:

Continue your journal of x86 instructions. You must have a minimum of 10 new instructions, 30 total instructs, in your journal. If needed find 10 new instructions online that you don't know and add them to your journal.

Submit your journal as part of homework submission.

flag{r0und_THR33}

Solution:

AAA – ASCII adjust AL after addition - Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result. - **Usage: AAA**

ADC - Add with carry - Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format. - **Usage: ADC AL, *imm8*** *(*Add with carry imm8 to AL).

ADDSD – Add scalar double-precision floating-point values - Adds the low double-precision floating-point values from the second source operand and the first source operand and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. - **Usage: F2 0F 58 /r ADDSD xmm1, xmm2/m64** (Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.)

ADDSUBPD – Packed double-FP add/subtract - Adds odd-numbered double-precision floating-point values of the first source operand (second operand) with the corresponding double-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered double-precision floating-point values from the second source operand from the corresponding double-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand. - **Usage: 66 0F D0 /r ADDSUBPD *xmm1, xmm2/m128*** *(*Add/subtract double-precision floating-point values from xmm2/m128 to xmm1.)

ADOX – Unsigned integer addition of two operands with overflow flag - Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the overflow-flag (OF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of OF represents a carry from a previous addition. The instruction sets the OF flag with the carry generated by the unsigned addition of the operands. - **Usage: F3 0F 38 F6 /r ADOX r32, r/m32** (Unsigned addition of r32 with OF, r/m32 to r32, writes OF.)

AESENCLAST - Perform last round of an AES encryption flow - This instruction performs the last round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand. - **Usage: 66 0F 38 DD /r AESENCLAST xmm1, xmm2/m128** (Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.)

ANDN - Logical AND NOT - Performs a bitwise logical AND of inverted second operand (the first source operand) with the third operand (the second source operand). The result is stored in the first operand (destination operand). - **Usage: VEX.NDS.LZ.0F38.W0 F2 /r ANDN r32a, r32b, r/m32** (Bitwise AND of inverted r32b with r/m32, store result in r32a.)

BEXTR - Bit field extract - Extracts contiguous bits from the first source operand (the second operand) using an index value and length value specified in the second source operand (the third operand). Bit 7:0 of the second source operand specifies the starting bit position of bit extraction. A START value exceeding the operand size will not extract any bits from the second source operand. Bit 15:8 of the second source operand specifies the maximum number of bits (LENGTH) beginning at the START position to extract. Only bit positions up to (OperandSize -1) of the first source operand are extracted. The extracted bits are written to the destination register, starting from the least significant bit. All higher order bits in the destination operand (starting at bit position LENGTH) are zeroed. The destination register is cleared if no bits are extracted. - **Usage: VEX.NDS.LZ.0F38.W0 F7 /r BEXTR r32a, r/m32, r32b** (Contiguous bitwise extract from r/m32 using r32b as control; store result in r32a.)

BLSI - Extract lowest set isolated bit - Extracts the lowest set bit from the source operand and set the corresponding bit in the destination register. All other bits in the destination operand are zeroed. If no bits are set in the source operand, BLSI sets all the bits in the destination to 0 and sets ZF and CF. - **Usage: VEX.NDD.LZ.0F38.W0 F3 /3 BLSI r32, r/m32** (Extract lowest set bit from r/m32 and set that bit in r32.)

BSR – Bit scan reverse - Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is 0, the content of the destination operand is undefined – **Usage: 0F BD /r** (Bit scan reverse on *r/m16.)*

flag{r0und_THR33}

jUNGLES aRE dANGEROUS

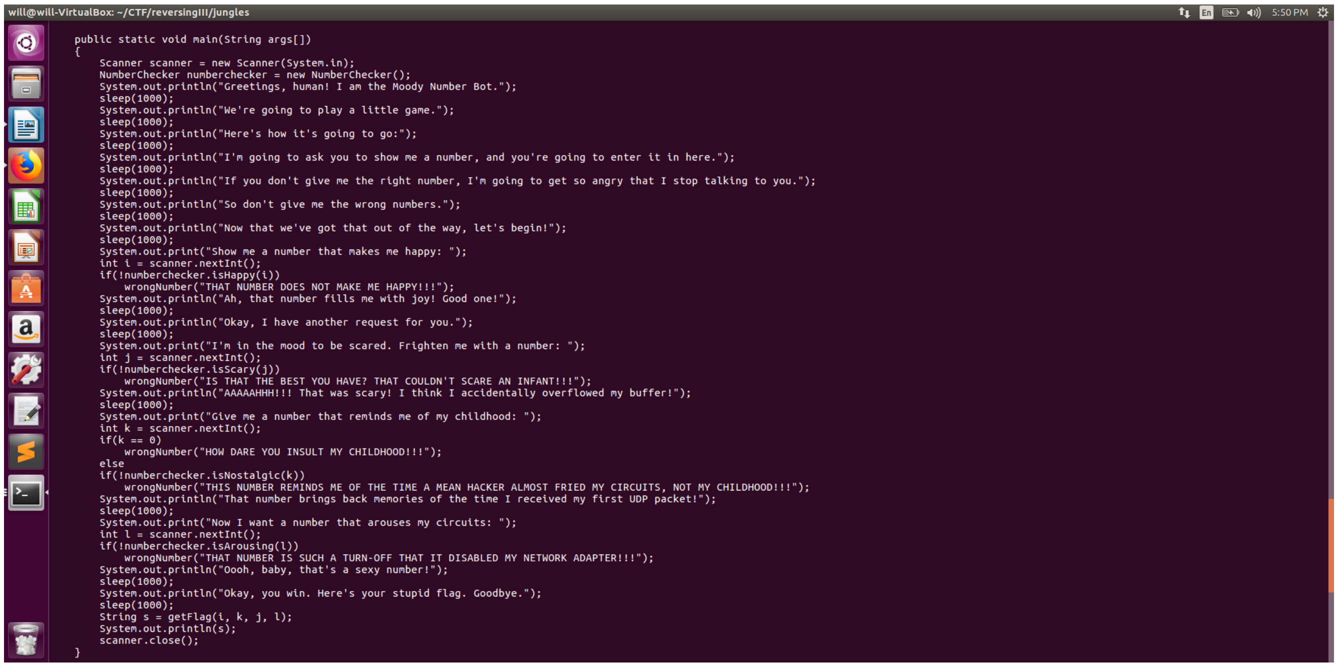
20 points

Problem:

Find the flag! Put all your steps in your write up.

Solution:

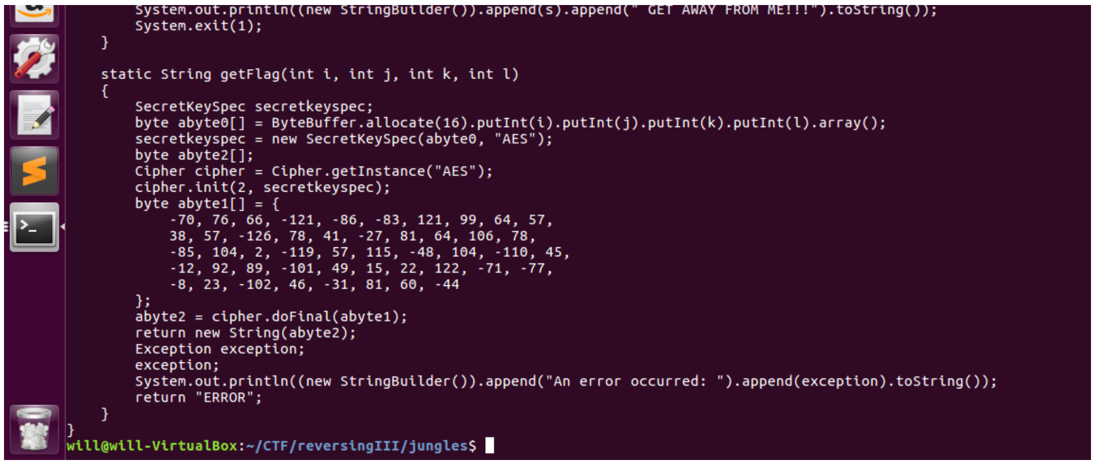
I downloaded the file ‘jUNGLES_aRE_dANGEROUS’, and I ran the ‘file’ command on it. I noticed it was a ‘JAR’ file. Using my knowledge of java, I know a JAR file is is just zipped java .class files. To get those class files, I used the command ‘unzip jUNGLES_aRE_dANGEROUS’. Here I see the class files ‘MoodyNumbers.class’ and ‘NumberChecker.class’. The JAR archive also contained an unimportant manifest file. I ran the program with ‘java -jar *S.java’ and realized it was prompting for a number. To analyze the java source code and figure out the number I need to enter I used the tool ‘jad’ with command ‘./jad *.class’ which created ‘MoodyNumbers.jad’ and ‘NumberChecker.jad’. Analyzing MoodyNumbers.jad with ‘cat *Numbers.jad’ I see the following



```
will@will-VirtualBox: ~/CTF/reversingIII/jungles

    public static void main(String args[])
    {
        Scanner scanner = new Scanner(System.in);
        NumberChecker numberchecker = new NumberChecker();
        System.out.println("Greetings, human! I am the Moody Number Bot.");
        sleep(1000);
        System.out.println("We're going to play a little game.");
        sleep(1000);
        System.out.println("Here's how it's going to go:");
        sleep(1000);
        System.out.println("I'm going to ask you to show me a number, and you're going to enter it in here.");
        sleep(1000);
        System.out.println("If you don't give me the right number, I'm going to get so angry that I stop talking to you.");
        sleep(1000);
        System.out.println("So don't give me the wrong numbers.");
        sleep(1000);
        System.out.println("Now that we've got that out of the way, let's begin!");
        sleep(1000);
        System.out.print("Show me a number that makes me happy: ");
        int i = scanner.nextInt();
        if(!numberchecker.isHappy(i))
            wrongNumber("THAT NUMBER DOES NOT MAKE ME HAPPY!!!");
        System.out.println("Ah, that number fills me with joy! Good one!");
        sleep(1000);
        System.out.println("Okay, I have another request for you.");
        sleep(1000);
        System.out.print("I'm in the mood to be scared. Frighten me with a number: ");
        int j = scanner.nextInt();
        if(!numberchecker.isScary(j))
            wrongNumber("IS THAT THE BEST YOU HAVE? THAT COULDN'T SCARE AN INFANT!!!");
        System.out.println("AAAAAHHH!!! That was scary! I think I accidentally overflowed my buffer!");
        sleep(1000);
        System.out.print("Give me a number that reminds me of my childhood: ");
        int k = scanner.nextInt();
        if(k == 0)
            wrongNumber("HOW DARE YOU INSULT MY CHILDHOOD!!!");
        else
        if(!numberchecker.isNostalgic(k))
            wrongNumber("THIS NUMBER REMINDS ME OF THE TIME A MEAN HACKER ALMOST FRIED MY CIRCUITS, NOT MY CHILDHOOD!!!");
        System.out.println("That number brings back memories of the time I received my first UDP packet!");
        sleep(1000);
        System.out.print("Now I want a number that arouses my circuits: ");
        int l = scanner.nextInt();
        if(!numberchecker.isArousing(l))
            wrongNumber("THAT NUMBER IS SUCH A TURN-OFF THAT IT DISABLED MY NETWORK ADAPTER!!!");
        System.out.println("Oooh, baby, that's a sexy number!");
        sleep(1000);
        System.out.println("Okay, you win. Here's your stupid flag. Goodbye.");
        sleep(1000);
        String s = getFlag(i, k, j, l);
        System.out.println(s);
        scanner.close();
    }
```

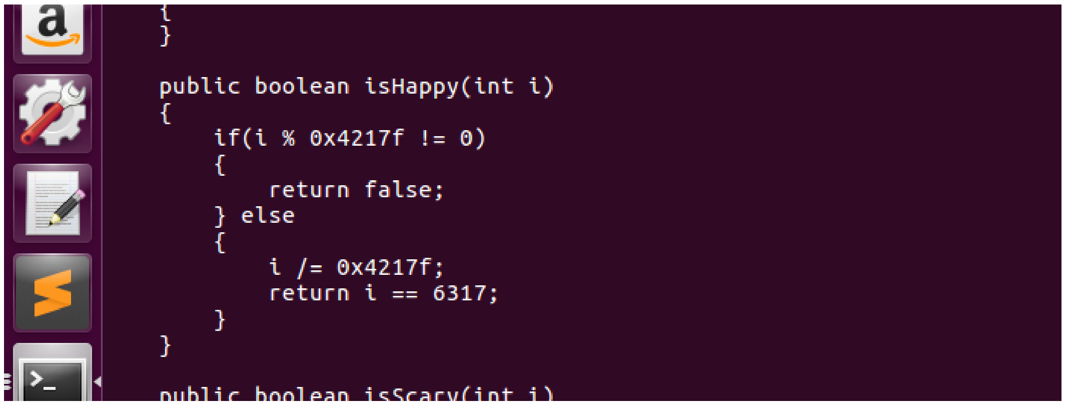
Here we see the main execution of the program. It is prompting for 4 numbers to be entered and those numbers are checked in 4 functions. You have to enter 4 numbers correctly and after the program accepts the 4 numbers, the getFlag() function is called which decrypts the flag using AES decrypting and the numbers you entered for the functions as decrypting keys.



```
        System.out.println((new StringBuilder()).append(s).append(" GET AWAY FROM ME!!!").toString());
        System.exit(1);
    }

    static String getFlag(int i, int j, int k, int l)
    {
        SecretKeySpec secretkeyspec;
        byte abyte0[] = ByteBuffer.allocate(16).putInt(i).putInt(j).putInt(k).putInt(l).array();
        secretkeyspec = new SecretKeySpec(abyte0, "AES");
        byte abyte2[];
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(2, secretkeyspec);
        byte abyte1[] = {
            -70, 76, 66, -121, -86, -83, 121, 99, 64, 57,
            38, 57, -126, 78, 41, -27, 81, 64, 106, 78,
            -85, 104, 2, -119, 57, 115, -48, 104, -110, 45,
            -12, 92, 89, -101, 49, 15, 22, 122, -71, -77,
            -8, 23, -102, 46, -31, 81, 60, -44
        };
        abyte2 = cipher.doFinal(abyte1);
        return new String(abyte2);
        Exception exception;
        exception;
        System.out.println((new StringBuilder()).append("An error occurred: ").append(exception).toString());
        return "ERROR";
    }
}
will@will-VirtualBox:~/CTF/reversingIII/jungles$ 
```

When the user enters an incorrect answer, that is not accepted by the checking functions, the program exits. So, to get the flag, we must find the numbers accepted by the 4 check functions. The functions can be found in ‘NumberChecker.jad’. The first number check is done with the the isHappy() function.



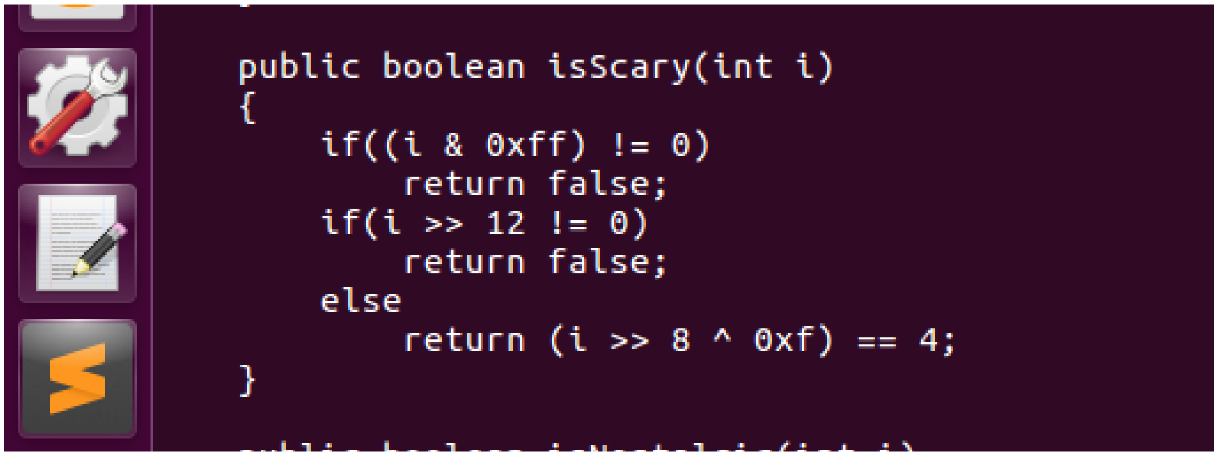
```
    {
    }

    public boolean isHappy(int i)
    {
        if(i % 0x4217f != 0)
        {
            return false;
        } else
        {
            i /= 0x4217f;
            return i == 6317;
        }
    }

    public boolean isScary(int i)
```

Here the parameter ‘i’ is the user entered number. We need the function to return true. For that to occur, we we need our user entered number to be divisible by 0x4217f (indicated by the if statement). This enters the else statement. 0x42197 is the decimal value 270719 (found using an online hex converter found at https://www.rapidtables.com/convert/number/hex-to-decimal.html). The user entered number is then divided by 0x4217f and the result of that division must be equal to 6317. For this to occur the user entered number must be 0x4217f * 6317. Which is 270719 * 6317 = **1710131923**. This is the first number to enter to the program. The program accepts it and prompts for a second number.

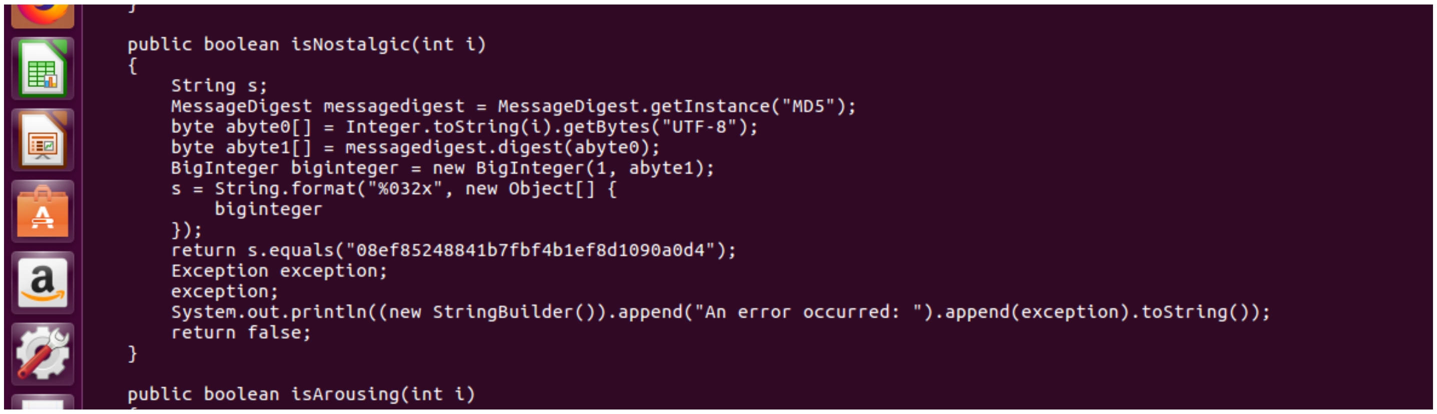
The second number is checked using the isScary() function.



```
public boolean isScary(int i)
{
    if((i & 0xff) != 0)
        return false;
    if(i >> 12 != 0)
        return false;
    else
        return (i >> 8 ^ 0xf) == 4;
}
```

The main program passes in the user entered number as ‘j’, and that is ‘i’ in this function. In the first if statement, it makes sure the lower byte is not equal to 255. This is because 255 & 0xff = 0. I used the number 256. The second if statement makes sure the number is less than 4096. That is because 2^12 = 4096 and shifting 12 bits divides by 2 every shift. To get a value of 0 the number must be less than 12 bits and therefore under the value of 4096. If the first 2 conditions are satisfied then the user entered value >> 8 ^0xf must equal 4 for the function to return true. I opened python and found out that 256 >> 8 = 1. I then multiplied 256 by 4 and found out at 1024 >> 8 = 4. However, 4 ^0xf =11. To reverse this xor I typed 11 0xf into python and confirmed that 11 0xf = 4. Therefore, I have to get the user input >> 8 to equal 11. To accomplish this I multiplied 256 * 11 which is 2816. I did this because 256 >> 8 = 1 so 258 * 11 must equal 11. 2816 satisfies the first 2 if statements and the final return condition. I entered **2816** into the program, and the program accepts the number and prompts for the a third input.

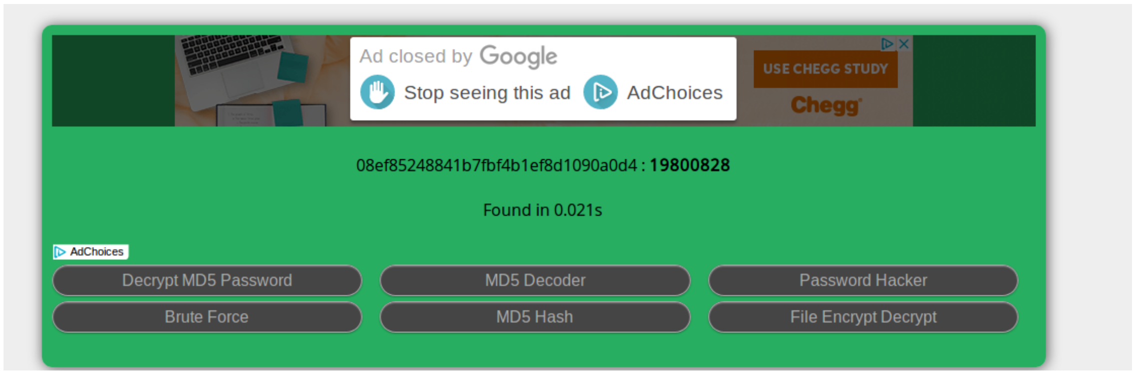
The third number is checked in the isNostalgic() function. The ‘i’ parameter is the third number the user entered.



```
public boolean isNostalgic(int i)
{
    String s;
    MessageDigest messagedigest = MessageDigest.getInstance("MD5");
    byte abyte0[] = Integer.toString(i).getBytes("UTF-8");
    byte abyte1[] = messagedigest.digest(abyte0);
    BigInteger biginteger = new BigInteger(1, abyte1);
    s = String.format("%032x", new Object[] {
        biginteger
    });
    return s.equals("08ef85248841b7fbf4b1ef8d1090a0d4");
    Exception exception;
    exception;
    System.out.println((new StringBuilder()).append("An error occurred: ").append(exception).toString());
    return false;
}

public boolean isArousing(int i)
```

Immediately I notice that some message is being MD5 encoded with ‘MessageDigest.getInstance(“MD5”)’. I also noticed a string "08ef85248841b7fbf4b1ef8d1090a0d4" is being returned. There is also a lot of other complicated stuff going on but that is unimportant. On a whim, I ran that string being returned through an MD5 decoder found at http://md5decrypt.net/en .

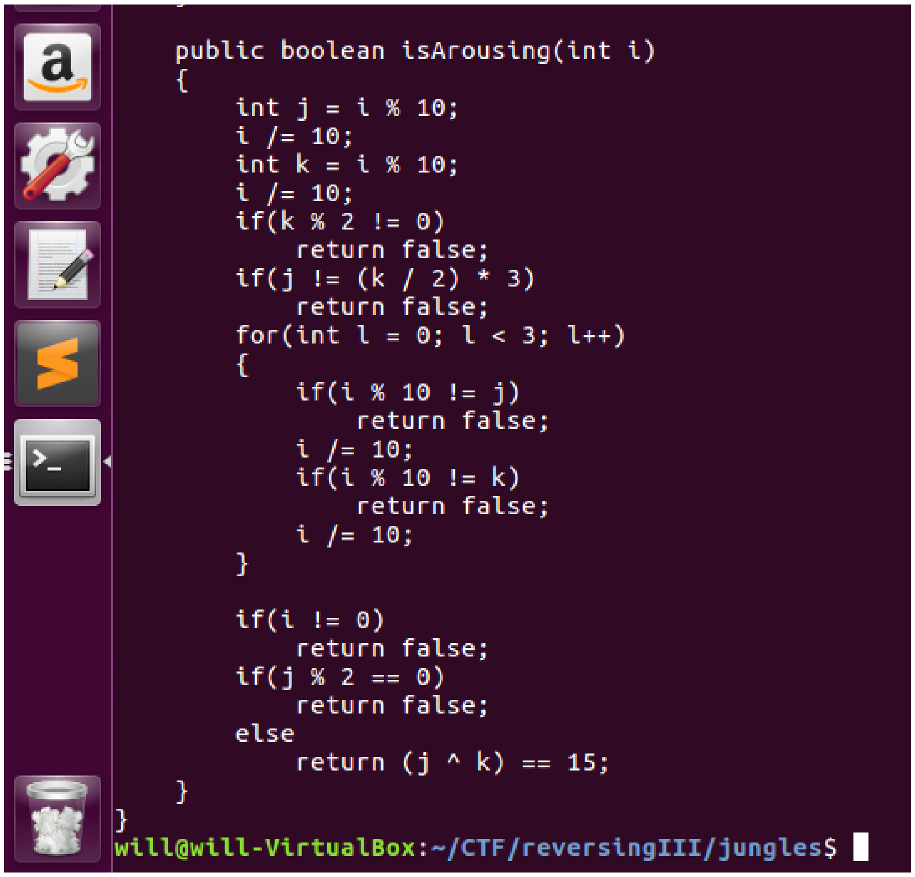


```
08ef85248841b7fbf4b1ef8d1090a0d4 : 19800828

Found in 0.021s
```

I entered **19800828** (the MD% decoded string) into the program and the program accepted it and prompted for a fourth number.

The fourth number is checked in the isArousing() function.



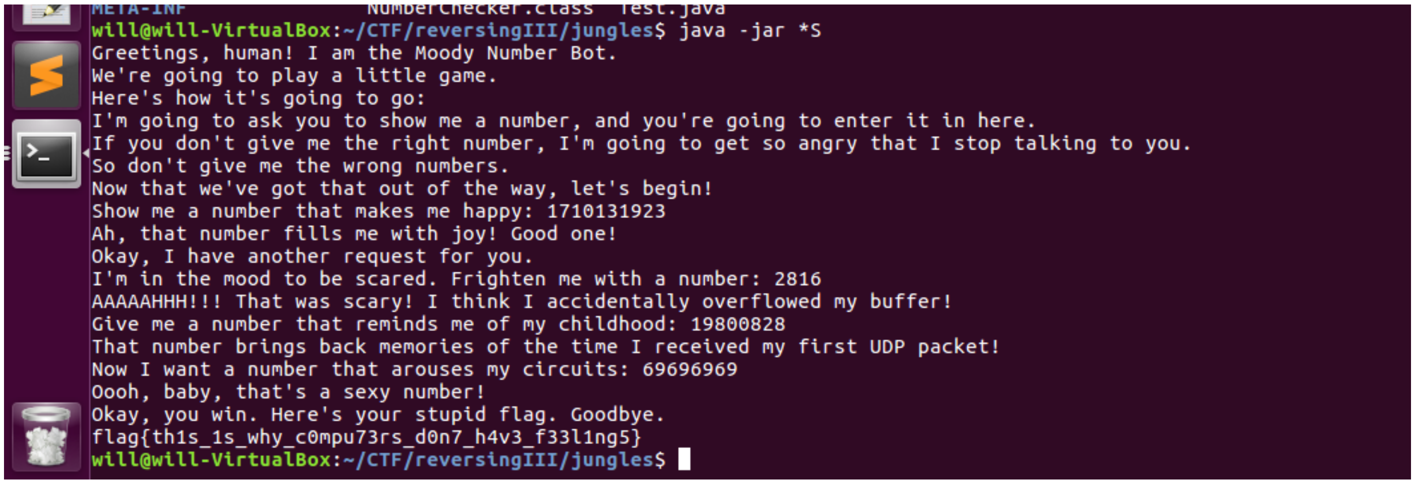
```
    public boolean isArousing(int i)
    {
        int j = i % 10;
        i /= 10;
        int k = i % 10;
        i /= 10;
        if(k % 2 != 0)
            return false;
        if(j != (k / 2) * 3)
            return false;
        for(int l = 0; l < 3; l++)
        {
            if(i % 10 != j)
                return false;
            i /= 10;
            if(i % 10 != k)
                return false;
            i /= 10;
        }

        if(i != 0)
            return false;
        if(j % 2 == 0)
            return false;
        else
            return (j ^ k) == 15;
    }
}
will@will-VirtualBox:~/CTF/reversingIII/jungles$
```

The first 4 lines, before the if statement,

```
int j = i % 10;
i /= 10;
int k = i % 10;
i /= 10;
```

Ensure that j and k are single digits 0 – 9. Also, the two I/=10 shave off two digits from the user entered number. if(k%2 != 0) return false ensures that k is even. If (j%2 ==0) return false ensure j is odd. If(j!= (k/2)*3) return false ensures that j=(3/2)k. The final return ensures j^k = 15. Using python, I see that 9^6 = 15. j=9 and k=6 and 9=(3/2)6. After the first 69 is entered on user input, there is a for loop that runs 3 times, checking these even and odd values and shaving off 2 digits from user input. Since the loop runs three times and we have 69 already. The final value must be 69696969. After entering **69696969** into the program, the flag printed and the program exited. After entering in all 4 correct numbers I get the flag.



```
META-INF                NumberChecker.class   Test.java
will@will-VirtualBox:~/CTF/reversingIII/jungles$ java -jar *S
Greetings, human! I am the Moody Number Bot.
We're going to play a little game.
Here's how it's going to go:
I'm going to ask you to show me a number, and you're going to enter it in here.
If you don't give me the right number, I'm going to get so angry that I stop talking to you.
So don't give me the wrong numbers.
Now that we've got that out of the way, let's begin!
Show me a number that makes me happy: 1710131923
Ah, that number fills me with joy! Good one!
Okay, I have another request for you.
I'm in the mood to be scared. Frighten me with a number: 2816
AAAAAHHH!!! That was scary! I think I accidentally overflowed my buffer!
Give me a number that reminds me of my childhood: 19800828
That number brings back memories of the time I received my first UDP packet!
Now I want a number that arouses my circuits: 69696969
Oooh, baby, that's a sexy number!
Okay, you win. Here's your stupid flag. Goodbye.
flag{th1s_1s_why_c0mpu73rs_d0n7_h4v3_f33l1ng5}
will@will-VirtualBox:~/CTF/reversingIII/jungles$
```

flag{th1s_1s_why_c0mpu73rs_d0n7_h4v3_f33l1ng5}

Solve Them All

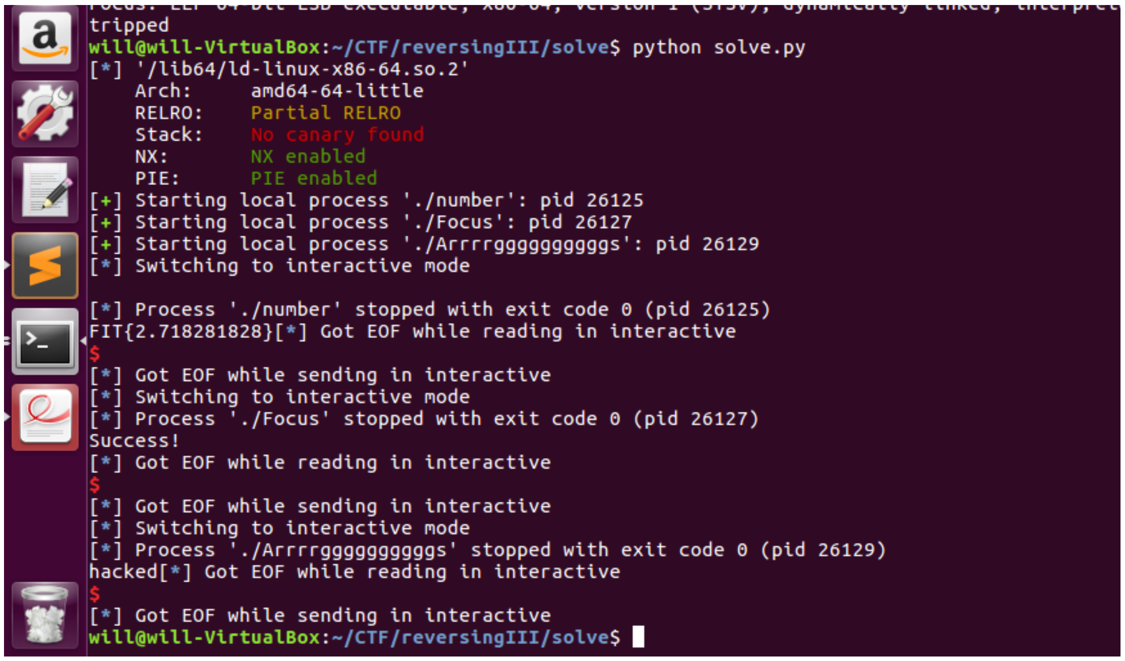
20 points

Problem:

Write a python script which uses the pwntools library to interact with Number, Arrrrgggggggggs, and Focus. Your script should print the flags from each challenge in one run.

flag{I_cAn_Pyth0n}

Solution:



```
Focus: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpret
tripped
will@will-VirtualBox:~/CTF/reversingIII/solve$ python solve.py
[*] '/lib64/ld-linux-x86-64.so.2'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process './number': pid 26125
[+] Starting local process './Focus': pid 26127
[+] Starting local process './Arrrrgggggggggs': pid 26129
[*] Switching to interactive mode

[*] Process './number' stopped with exit code 0 (pid 26125)
FIT{2.718281828}[*] Got EOF while reading in interactive
$
[*] Got EOF while sending in interactive
[*] Switching to interactive mode
[*] Process './Focus' stopped with exit code 0 (pid 26127)
Success!
[*] Got EOF while reading in interactive
$
[*] Got EOF while sending in interactive
[*] Switching to interactive mode
[*] Process './Arrrrgggggggggs' stopped with exit code 0 (pid 26129)
hacked[*] Got EOF while reading in interactive
$
[*] Got EOF while sending in interactive
will@will-VirtualBox:~/CTF/reversingIII/solve$
```

My code is attached as ‘solve.py’

I ran ‘file’ on all the commands and realized the executable format interpreter was ‘/lib64/ld-linux-x86-64.so.2’ and is 64 bit ‘amd64’ architecture. Since all the processes are local, we simply use the process command to run programs in pwntools. Using the examples from the slides, solving number was rather easy. First I opened the pwnlibrary to use its tools. I simply opened the ‘./number’ file with the ‘process’ command, read the program input until being prompted for the number with ‘recvuntil’ and sent the correct input with ‘send(“1152\n”)’. Finally, I used the ‘interactive’ function to interact with the program at run time. For focus, I repeated the same process without the ‘recvuntil’ since the program does not prompt for input with text. Args was a little more complicated. Since Args took the flag as a command line argument separated by spaces I had to figure out how to send arguments to pwntools. Looking at documentation, I realized you can run a program with command lines by passing in a list to ‘process’. The arguments follow the first list entry, which is the program to be run.

flag{I_cAn_Pyth0n}

Research

20 points

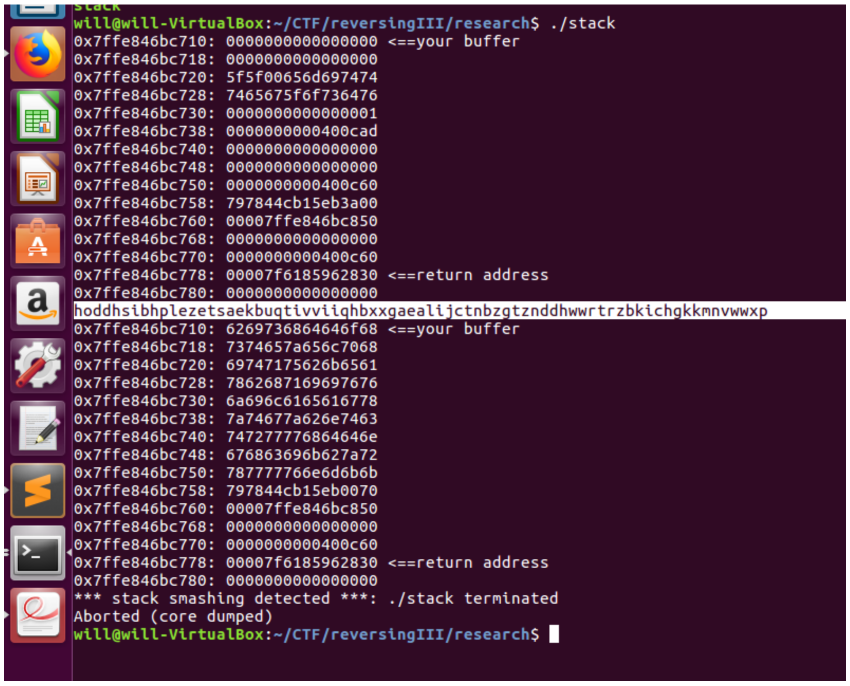
Problem:

Analyze the provided binary and explain with specific detail how and why you can make this program crash (or print an error).

flag{I_pr0mis3_I_Did_th3_r3seaRch}

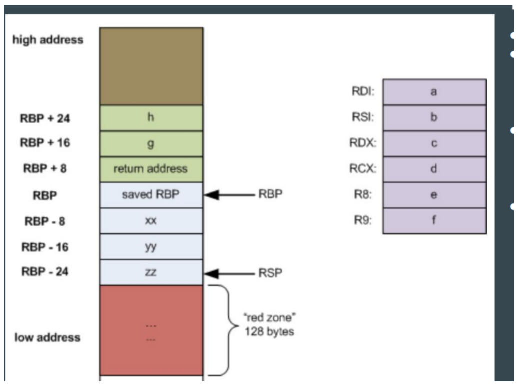
Solution:

The maximum number of characters the program can handle being entered, during execution, before crashing is 72. On the 73rd character, the program aborts on a core dump as follows.



```
stack
will@will-VirtualBox:~/CTF/reversingIII/research$ ./stack
0x7ffe846bc710: 0000000000000000 <==your buffer
0x7ffe846bc718: 0000000000000000
0x7ffe846bc720: 5f5f00656d697474
0x7ffe846bc728: 7465675f6f736476
0x7ffe846bc730: 0000000000000001
0x7ffe846bc738: 0000000000400cad
0x7ffe846bc740: 0000000000000000
0x7ffe846bc748: 0000000000000000
0x7ffe846bc750: 0000000000400c60
0x7ffe846bc758: 797844cb15eb3a00
0x7ffe846bc760: 00007ffe846bc850
0x7ffe846bc768: 0000000000000000
0x7ffe846bc770: 0000000000400c60
0x7ffe846bc778: 00007f6185962830 <==return address
0x7ffe846bc780: 0000000000000000
hoddhsibhplezetsaekbuqtivviiqhbxxgaealijctnbzgtznddhwwrtrzbkichgkkmnvwwxp
0x7ffe846bc710: 6269736864646f68 <==your buffer
0x7ffe846bc718: 7374657a656c7068
0x7ffe846bc720: 69747175626b6561
0x7ffe846bc728: 7862687169697676
0x7ffe846bc730: 6a696c6165616778
0x7ffe846bc738: 7a74677a626e7463
0x7ffe846bc740: 747277776864646e
0x7ffe846bc748: 676863696b627a72
0x7ffe846bc750: 787777766e6d6b6b
0x7ffe846bc758: 797844cb15eb0070
0x7ffe846bc760: 00007ffe846bc850
0x7ffe846bc768: 0000000000000000
0x7ffe846bc770: 0000000000400c60
0x7ffe846bc778: 00007f6185962830 <==return address
0x7ffe846bc780: 0000000000000000
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
will@will-VirtualBox:~/CTF/reversingIII/research$
```

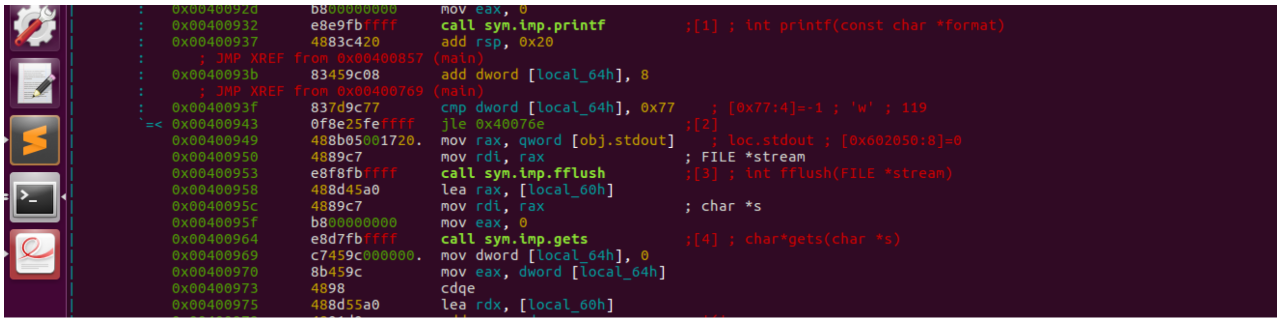
I found the answer to my solution in rather convoluted way. Firstly I ran ‘file ./stack’ and noticed it is a 64 bit ELF file. Therefore the stack used is 64 bit and each stack entry is 8 bytes. An example stack is as follows.



The saved RBP contains the return address for a function. To cause stack smashing we must overwrite part of this 8 byte return address for a vulnerable buffer (buffer overflow)

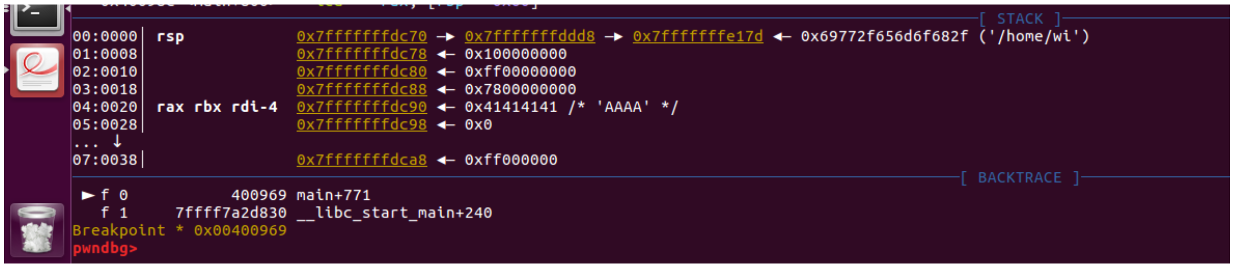
To see the size of a character I ran the program, entered a single ‘A’ and see the stack fills 1 of the 8 bytes of the first entry on the stdout stack with 0x41 (hex for A). Therefore each character is 1 byte. Analyzing the stack on the layout of the program I see that the return address is stored 0x68 bytes (108 decimal) away from the buffer. (0x778-0x710). This made me think 109 characters are needed to overwrite the return address. This is not the case.

Next I opened the program in radare. I opened up with radare with ‘r2 ./stack’ then proceeded to type ‘aaaa’ to analyze the binary. I then typed ‘s main’ to seek to main and ‘V’ to open visual mode. Inside radare I noticed the following.



```
      :   0x00400932      e8e9fbffff     call sym.imp.printf         ;[1] ; int printf(const char *format)
      :   0x00400937      4883c420       add rsp, 0x20
      :      ; JMP XREF from 0x00400857 (main)
      :   0x0040093b      83459c08       add dword [local_64h], 8
      :      ; JMP XREF from 0x00400769 (main)
      :   0x0040093f      837d9c77       cmp dword [local_64h], 0x77    ; [0x77:4]=-1 ; 'w' ; 119
      `=< 0x00400943      0f8e25feffff   jle 0x40076e                ;[2]
          0x00400949      488b05001720.  mov rax, qword [obj.stdout]    ; loc.stdout ; [0x602050:8]=0
          0x00400950      4889c7         mov rdi, rax                ; FILE *stream
          0x00400953      e8f8fbffff     call sym.imp.fflush         ;[3] ; int fflush(FILE *stream)
          0x00400958      488d45a0       lea rax, [local_60h]
          0x0040095c      4889c7         mov rdi, rax                ; char *s
          0x0040095f      b800000000     mov eax, 0
          0x00400964      e8d7fbffff     call sym.imp.gets           ;[4] ; char*gets(char *s)
          0x00400969      c7459c000000.  mov dword [local_64h], 0
          0x00400970      8b459c         mov eax, dword [local_64h]
          0x00400973      4898           cdqe
          0x00400975      488d55a0       lea rdx, [local_60h]
```

After the initial printing of the stack, there is a call to sym.imp.gets. This must be the vulnerable buffer. I decided to use pwndbg to analyze the program behavior. I opened pwndbg with ‘gdb .stack’, typed ‘start’ to start execution and set a breakpoint at the instruction right after sym.imp.gets with ‘b *0x00400969’ After typing ‘c’ to continue program execution, I typed ‘AAAA’ into the buffer. The output of pwndbg showed me the following.



```
───────────────────────────────────────────────────────────────────[ STACK ]──────────
00:0000│ rsp          0x7fffffffdc70 —▸ 0x7fffffffddd8 —▸ 0x7fffffffe17d ◂— 0x69772f656d6f682f ('/home/wi')
01:0008│              0x7fffffffdc78 ◂— 0x100000000
02:0010│              0x7fffffffdc80 ◂— 0xff00000000
03:0018│              0x7fffffffdc88 ◂— 0x7800000000
04:0020│ rax rbx rdi-4 0x7fffffffdc90 ◂— 0x41414141 /* 'AAAA' */
05:0028│              0x7fffffffdc98 ◂— 0x0
... ↓
07:0038│              0x7fffffffdca8 ◂— 0xff000000
─────────────────────────────────────────────────────────────────[ BACKTRACE ]──────────
 ► f 0           400969 main+771
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint * 0x00400969
pwndbg>
```

The 4 inputs bytes are stored on the stack initially at 0x20 away from the stack pointer. Now, since the input buffer stores its entries starting 0x20 away from the original buffer (esp), and the return address is stored 0x68 away from the start of the stack (the original shown buffer), we can find out the needed characters to overflow the buffer by doing 0x68-0x20=0x48 = 72 decimal. This brings us to the base of the return address. Therefore, entering a 73rd character (byte) into the buffer writes over the return address and causes stack smashing. This stack smashing can be dangerous. After 72 characters, we could write our own assembly into the return address by writing assembly instruction in x86. This way we could execute our own assembly on a vulnerable program through overwriting the return address of a vulnerable buffer.

flag{I_pr0mis3_I_Did_th3_r3seaRch}