

# Reverse Engineering I Write-Up

## Install the tools!

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.

- Radare2
- Ipython

If you wish to you wish to use different tool/s please provide a brief description of the tool/s and which tool/s from above are being replaced. NOTE: You are unsupported if you have technical problems with a tool not listed above. Also do not use non-free, non-open-source software.

flag{R3\_w33k\_0n3}

### Solution:



```
will@will-VirtualBox:~/CTF/reversing$ r2 --version
r2: invalid option -- '-'
radare2 2.4.0-git 17355 @ linux-x86-64 git.2.2.0-544-g53df70f
commit: 53df70fbd337e83b567f0eff8c551ecf450e7a3d build: 2018-02-21__12:46:56
will@will-VirtualBox:~/CTF/reversing$ ipython --version
5.5.0
will@will-VirtualBox:~/CTF/reversing$
```

flag{R3\_w33k\_0n3}

## Journal

10 Points

### Problem:

Keep a journal of all x86 instructions you did not already know. You must have a minimum of 10 instructions in your journal. If needed find 10 instructions online that you don't know and add them to your journal.

Submit your journal as part of homework submission.

flag{3asy\_p0int5}

## Solution:

**imul** –The imul instruction has two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above).

The two-operand form multiplies its two operands together and stores the result in the first operand. The result (i.e. `bx`) operand must be a register. – **imul `bx`, 12h**

The three operand form multiplies its second and third operands together and stores the result in its first operand. Again, the result operand must be a register. Furthermore, the third operand is restricted to being a constant value.

**mov** - The mov instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address. - **mov `eax`, `ebx` (copy the value in `ebx` into `eax`)**

**lea** - Load effective address - The lea instruction places the *address* specified by its second operand (`[val]`) into the register specified by its first operand. Note, the *contents* of the memory location are not loaded, only the effective address is computed and placed into the register. This is useful for obtaining a pointer into a memory region. - **lea `eax`, `[val]` (the address of `val` is placed in `eax`)**

**inc** - Increment - The inc instruction increments the contents of its operand (`eax`) by one. - **inc `eax`**

**dec** – Decrement - The dec instruction decrements the contents of its operand (`eax`) by one – **dec `eax`**

**not** - Bitwise logical not - Logically negates the operand contents (that is, flips all bit values in the operand `eax`) – **not `eax`**

**neg** - Negate - Performs the two's complement negation of the operand contents (`eax`) – **neg `eax`**

**cmp** - Compare - Compare the values of the two specified operands (`eax`, `ebx`), setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand. - **cmp `eax`, `ebx`**

**push** – Push onto the stack - The push instruction places its operand (`eax`) onto the top of the hardware supported stack in memory. Specifically, push first decrements ESP by 4, then places its operand into

the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses. **push eax**

**pop** – Pop the stack - The pop instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register (eax) or memory location ). It first moves the 4 bytes located at memory location [SP] into the specified register or memory location, and then increments SP by 4. - **pop eax**

**flag{3asy\_p0int5}**

## .bss

10 Points

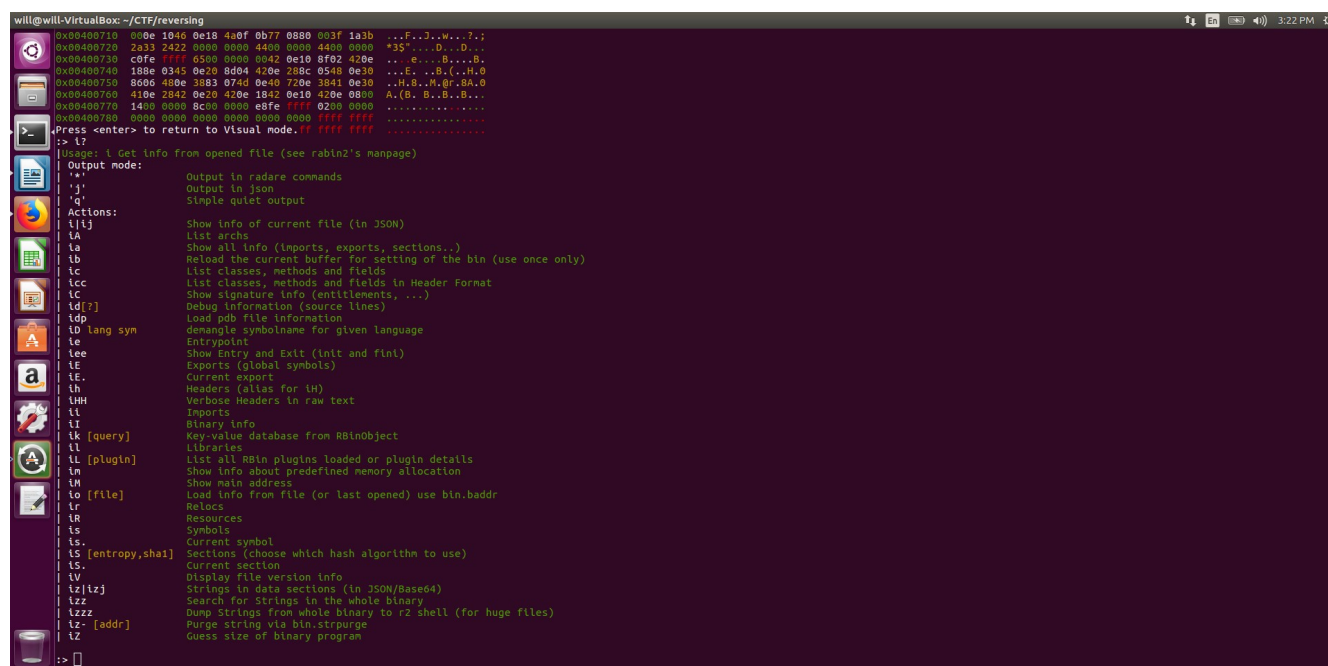
## Problem:

Analyze the challenge binary and find the flag in the .bss section. Add flag{} to your found flag.

NOTE: 1pt deduction per flag submitted that is from a different section.

## Solution:

This is a reversing challenge on a file called 'challenge'. Since it is a reversing problem I must use Radare. I started Radare with 'r2 challenge'. I immediately typed 'aaaa' to analyze the binary. I then hit 'V' to enter visual mode. I then used the ':i?' command.



```
will@will-VirtualBox: ~/CTF/reversing
0x00400710  0000 1045 0e18 4a0f 0b77 0880 003f 1a3b  ...F..J..W...?..
0x00400720  2a33 2422 0800 0000 4400 0000 4400 0000  *33"...D...D...
0x00400730  c0fe ffff 6500 0000 0042 0e10 8f02 420e  ...e....B....B..
0x00400740  188e 0345 0e20 8d04 420e 208c 0540 0e30  ...E...B.(..H.O
0x00400750  0600 408e 0033 0740 0e40 720e 3041 0e30  ...H.B..H.B..B..
0x00400760  410e 2042 0e20 420e 1842 0e10 420e 0800  A.(B..B..B...
0x00400770  1400 0000 8c00 0000 e8fe ffff 0200 0000  .....
0x00400780  0000 0000 0000 0000 0000 0000 ffff ffff  .....
Press <enter> to return to Visual mode. // ffff ffff .....
> i?
Usage: i Get info from opened file (see rabin2's manpage)
Output mode:
**
'j'      Output in radare commands
'j'      Output in json
'q'      Simple quiet output
Actions:
i[i?]    Show info of current file (in JSON)
ia       List archs
ia       Show all info (Imports, exports, sections..)
ib       Reload the current buffer for setting of the bin (use once only)
ic       List classes, methods and fields
icc      List classes, methods and fields in Header Format
ic       Show signature info (entitlements, ...)
id[?]    Debug information (source lines)
idp      Load pdb file information
id lang sym  demangle symbolname for given language
ie       Entrypoint
iee      Show Entry and Exit (Init and final)
ie       Exports (Global symbols)
ie       Current export
ih       Headers (alias for ih)
iHH      Verbose Headers in raw text
ii       Imports
ii       Binary info
ik [query] Key-value database from RabinObject
il       Libraries
il [plugin] List all Rabin plugins loaded or plugin details
im       Show info about predefined memory allocation
in       Show main address
io [file] Load info from file (or last opened) use bin.baddr
ir       Relocs
is       Resources
is       Symbols
is       Current symbol
is [entropy,sha1] Sections (choose which hash algorithm to use)
is       Current section
iv       Display file version info
iz[iiz]  Strings in data sections (in JSON/Base64)
izz      Search for Strings in the whole binary
izzz     Dump Strings from whole binary to r2 shell (for huge files)
izzz [addr] Purge string via bin.strpurge
iz       Guess size of binary program
>
```

I tried 'is' under Brandon's suggestion.

```
will@will-VirtualBox: ~/CTF/reversing
Flags in flagspace 'sections'. Press '?' for help.
045 0x00600ff8 0 section_end.dynamic
046 0x00600ff8 0 section.got
047 0x00601000 0 section_end.got
048 0x00601000 48 section.got.plt
049 0x00601030 0 section_end.got.plt
050 0x00601030 64 section.data
051 0x0060106d 0 section_end.data
> 052 0x00601070 0 section.bss
053 0x00601088 0 section_end.bss
054 0x00000000 52 section.comment
055 0x00000034 0 section_end.comment
056 0x00000000 268 section.shstrtab
057 0x0000010c 0 section_end.shstrtab
058 0x00000000 1800 section.syntax
Selected: section.bss

;-- section.bss:
;-- completed.7585:
;-- __TMC_END__:
0x00601070 0000 add byte [rax], al ; [26] --rw- section size 0 named .bss
0x00601072 0000 add byte [rax], al
;-- number:
0x00601074 0000 add byte [rax], al
0x00601076 0000 add byte [rax], al
0x00601078 0000 add byte [rax], al
0x0060107a 0000 add byte [rax], al
;-- did_y0u_us3_strings:
0x0060107c 0000 add byte [rax], al
0x0060107e 0000 add byte [rax], al
0x00601080 0000 add byte [rax], al
0x00601082 0000 add byte [rax], al
0x00601084 0000 add byte [rax], al
0x00601086 0000 add byte [rax], al
;-- section_end.bss:
;-- section_end.LOAD1:
;-- _end:
0x00601088 ff invalid
0x00601089 ff invalid
0x0060108a ff invalid
0x0060108b ff invalid
0x0060108c ff invalid
0x0060108d ff invalid
0x0060108e ff invalid
0x0060108f ff invalid
0x00601090 ff invalid
0x00601091 ff invalid
0x00601092 ff invalid
0x00601093 ff invalid
0x00601094 ff invalid
0x00601095 ff invalid
0x00601096 ff invalid
```

I find the bss section, which is where the flag is. The second column contains the address of that section in Radare. The address for bss is 0x00601070. I navigate there with 's 0x00601070'. After hitting 'V' again I page through to the instructions with 'p' and see the flag after scrolling down.

```
will@will-VirtualBox: ~/CTF/reversing
[0x00601070 47% 270 challenge]> pd $r @ obj.completed.7585
;-- section.bss:
;-- completed.7585:
;-- __TMC_END__:
; DATA XREF from 0x004004c0 (sym.deregister_tm_clones)
; DATA XREF from 0x004004e0 (sym.register_tm_clones)
; DATA XREF from 0x0040050e (sym.register_tm_clones)
; DATA XREF from 0x00400533 (sym.__do_global_dtors_aux)
0x00601070 0000 add byte [rax], al ; [26] --rw- sect
0x00601072 0000 add byte [rax], al
;-- number:
; DATA XREF from 0x00400585 (main)
; DATA XREF from 0x004005c6 (main)
0x00601074 0000 add byte [rax], al
0x00601076 0000 add byte [rax], al
0x00601078 0000 add byte [rax], al
0x0060107a 0000 add byte [rax], al
;-- did_y0u_us3_strings:
0x0060107c 0000 add byte [rax], al
0x0060107e 0000 add byte [rax], al
0x00601080 0000 add byte [rax], al
0x00601082 0000 add byte [rax], al
0x00601084 0000 add byte [rax], al
0x00601086 0000 add byte [rax], al
;-- section_end.bss:
;-- section_end.LOAD1:
;-- _end:
```

flag{did\_y0u\_us3\_strings}

# .data

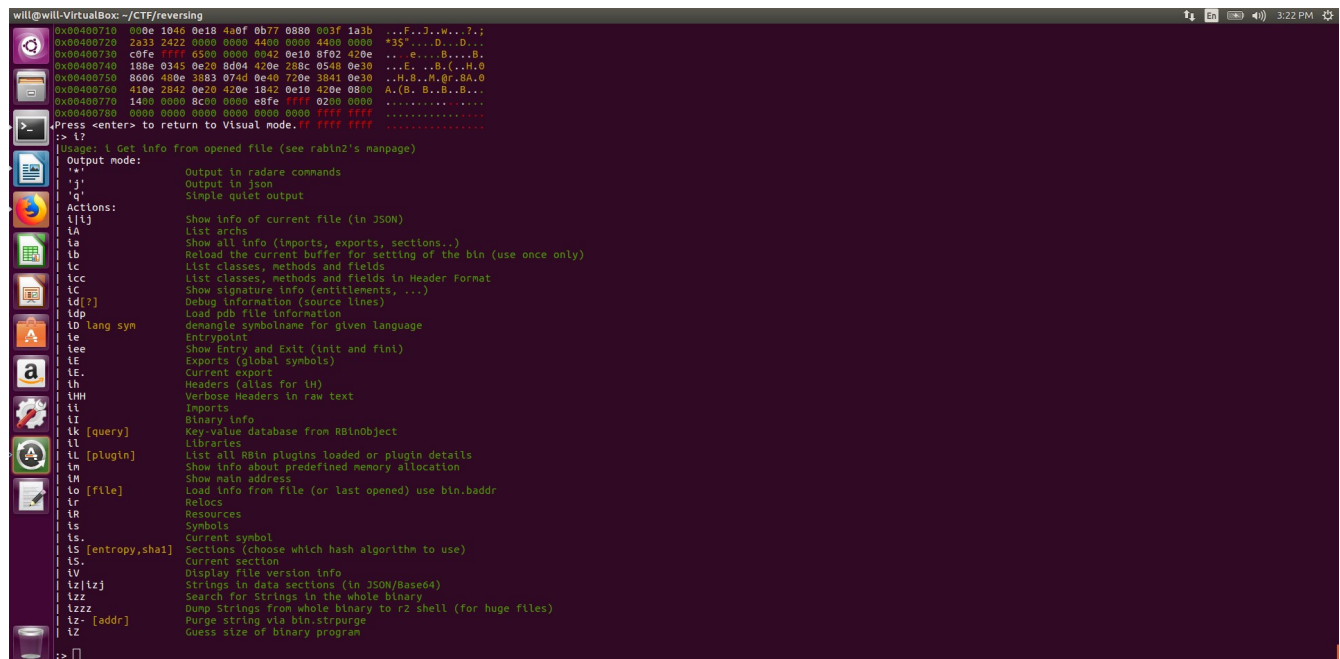
10 Points

## Problem:

Continue to analyze the challenge binary and find the flag in the .data section. Add flag{ } to your found flag. NOTE: 1pt deduction per flag submitted that is from a different section.

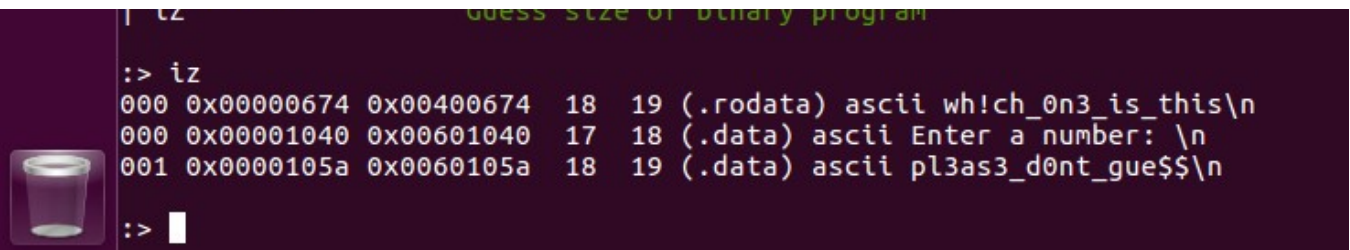
## Solution:

This is a reversing challenge on a file called 'challenge'. Since it is a reversing problem I must use Radare. I started Radare with 'r2 challenge'. I immediately typed 'aaaa' to analyze the binary. I then hit 'V' to enter visual mode. I then used the ':i?' command.



```
will@will-VirtualBox: ~/CTF/reversing
0x00400710 000e 1046 0e18 4a0f 0b77 0880 003f 1a3b ...F..J..w...7.;
0x00400720 2a33 2422 0000 0000 4400 0000 4400 0000 *35*...D...D...
0x00400730 c0fe 777f 6500 0000 0042 0e10 8f02 420e ...e...B...B...
0x00400740 180e 0345 0e20 8d04 420e 208c 0540 0e30 ...E...B...H...0
0x00400750 8606 480e 3883 074d 0e40 720e 3841 0e30 ..H..M..Qr..8A.0
0x00400760 410e 2842 0e20 420e 1842 0e10 420e 0880 A.(B..B..B...
0x00400770 1400 0000 0c00 0000 c8fe 777f 0200 0000 .....
0x00400780 0000 0000 0000 0000 0000 777f 777f .....
Press <enter> to return to Visual mode.77 777f 777f .....
> i?
Usage: i Get info from opened file (see rabin2's manpage)
Output mode:
+*+ Output in radare commands
+j+ Output in json
+q+ Simple quiet output
Actions:
|i|i Show info of current file (in JSON)
|A| List archs
|a| Show all info (imports, exports, sections...)
|b| Reload the current buffer for setting of the bin (use once only)
|c| List classes, methods and fields
|cc| List classes, methods and fields in Header Format
|c| Show signature info (entitlements, ...)
|d[?]| Debug information (source lines)
|dp| Load pdb file information
|D| demangle symbolname for given language
|E| Entrypoint
|ee| Show Entry and Exit (init and fini)
|E| Exports (global symbols)
|E| Current export
|th| Headers (alias for th)
|HH| Verbose Headers in raw text
|I| Imports
|I| Binary info
|k[query]| Key-value database from RBinObject
|l| Libraries
|L[plugin]| List all RBin plugins loaded or plugin details
|n| Show info about predefined memory allocation
|n| Show main address
|o[.file]| Load info from file (or last opened) use bin.baddr
|r| Relocs
|r| Resources
|s| Symbols
|s| Current symbol
|S[entropy,sha1]| Sections (choose which hash algorithm to use)
|S| Current section
|v| Display file version info
|z| Strings in data sections (in JSON/Base64)
|zz| Search for strings in the whole binary
|zzz| Dump strings from whole binary to r2 shell (for huge files)
|z- [addr]| Purge string via bin.strpurge
|z| Guess size of binary program
> |
```

I notice the 'iz' command gives strings contained in the data section.



```
will@will-VirtualBox: ~/CTF/reversing
> iz
000 0x000000674 0x00400674 18 19 (.rodata) ascii wh!ch_on3_is_this\n
000 0x000001040 0x00601040 17 18 (.data) ascii Enter a number: \n
001 0x00000105a 0x0060105a 18 19 (.data) ascii pl3as3_d0nt_gue$$\n
> |
```

I used that command and see there are 2 strings in the .data section. One of them looks like a flag and sure enough it is.

flag{pl3as3\_d0nt\_gue\$\$}

## .rodata

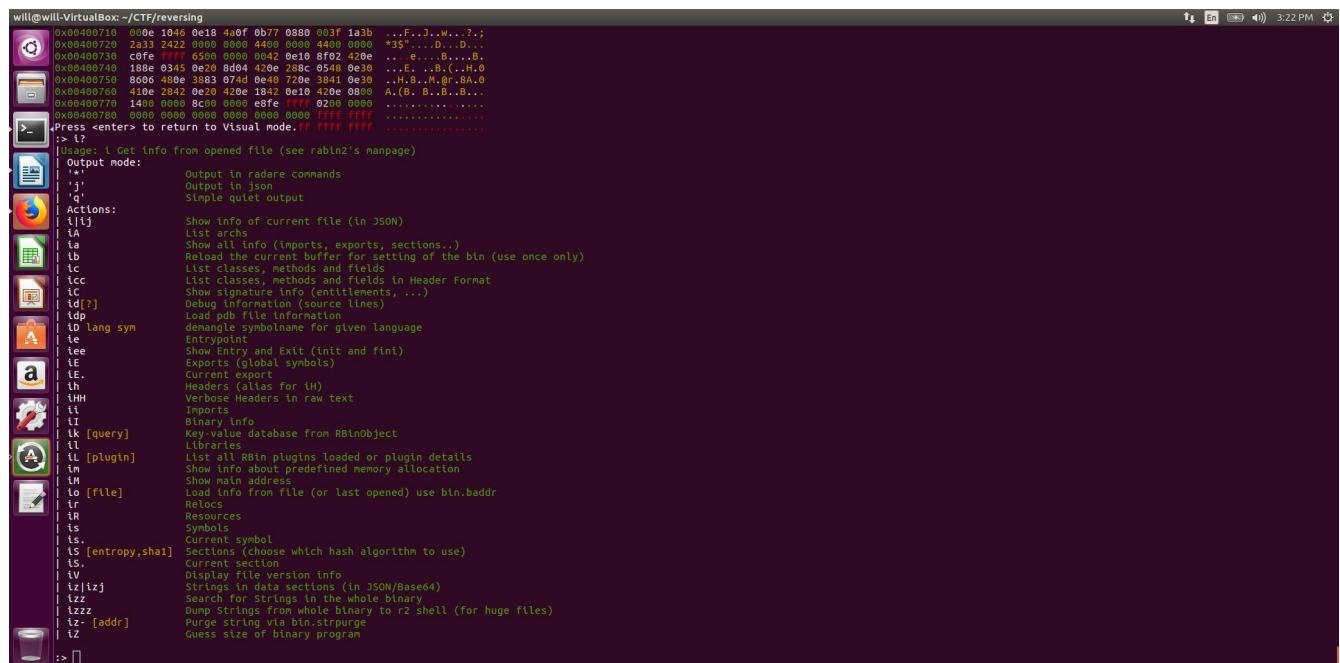
10 Points

## Problem:

Continue to analyze the challenge binary and find the flag in the .rodata section. Add flag{} to your found flag. NOTE: 1pt deduction per flag submitted that is from a different section.

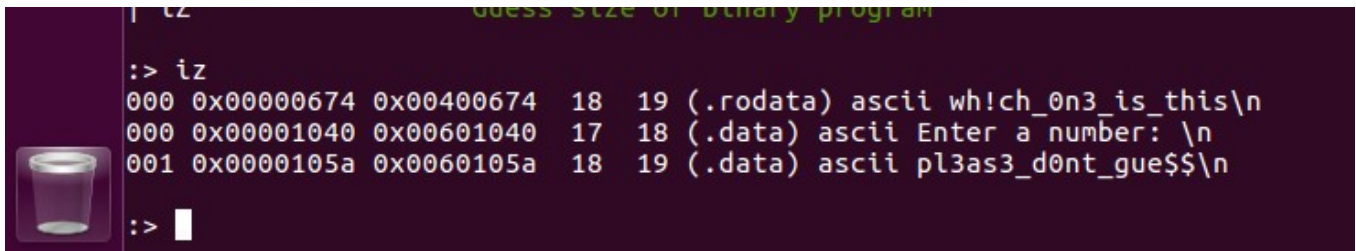
## Solution:

This is a reversing challenge on a file called ‘challenge’. Since it is a reversing problem I must use Radare. I started Radare with ‘r2 challenge’. I immediately typed ‘aaaa’ to analyze the binary. I then hit ‘V’ to enter visual mode. I then used the ‘:i?’ command.



```
will@will-VirtualBox: ~/CTF/reversing
0x00400710 000e 1046 0e18 4a0f 0b77 0880 003f 1a3b ...F..J..w...7.;
0x00400720 2a33 2422 0a00 0000 4400 0000 4400 0000 *33"...D...D...
0x00400730 c0fe ffff 6500 0000 0042 0e10 8f02 420e ...e...B...B...
0x00400740 189e 0345 0e20 8d04 420e 288c 0548 0e30 ...E...B...H..0
0x00400750 8000 480e 3883 074d 0e40 720e 3941 0e30 ..H..M..0r..BA.0
0x00400760 410e 2842 0e20 420e 1942 0e10 420e 0000 A.(B..B..B....
0x00400770 1430 0000 8c00 0000 e8fe ffff 0200 0000 .....
0x00400780 0000 0000 0000 0000 0000 0000 ffff ffff .....
Press <enter> to return to Visual mode. ff ffff ffff .....
:: V
Usage: L Get info from opened file (see rabin2's manpage)
Output mode:
**
'j'      Output in json
'q'      Simple quiet output
Actions:
U|L|J   Show info of current file (in JSON)
la      List archs
Ia      Show all info (imports, exports, sections..)
lb      Reload the current buffer for setting of the bin (use once only)
lc      List classes, methods and fields
lcc     List classes, methods and fields in Header Format
lc      Show signature info (entitlements, ...)
ld[?]   Debug information (source lines)
ldp     Load pdb file information
ld lang sym  demangle symbolname for given language
te      Entrypoint
tee     Show Entry and Exit (init and fini)
IE      Exports (global symbols)
IE.     Current export
lh      Headers (alias for lh)
lHH     Verbose Headers in raw text
li      Imports
li      Binary info
lk [query]  Key-value database from RBinObject
ll      Libraries
ll [plugin] List all RBin plugins loaded or plugin details
lm      Show info about predefined memory allocation
lm      Show main address
lo [file] Load info from file (or last opened) use bin.baddr
lr      Relocs
lr      Resources
ls      Symbols
ls.     Current symbol
ls [entropy,shai] Sections (choose which hash algorithm to use)
ts.     Current section
iv      Display file version info
rz|izj  Strings in data sections (in JSON/Base64)
lzz     Search for Strings in the whole binary
lzzzz   Dump Strings from whole binary to r2 shell (for huge files)
lz- [addr] Purge string via bin.striprurge
lZ      Guess size of binary program
::
```

I notice the ‘iz’ command gives strings contained in the data section.



```
000 0x00000674 0x00400674 18 19 (.rodata) ascii wh!ch_0n3_is_this\\n
000 0x00001040 0x00601040 17 18 (.data) ascii Enter a number: \\n
001 0x0000105a 0x0060105a 18 19 (.data) ascii pl3as3_d0nt_gue$$\\n
:> 
```

I used that command and see there is 1 string in the .rodata section. It looks like a flag and sure enough it is.

flag{wh!ch\_0n3\_is\_this}

## Warm up

10 Points

### Problem:

Continue to analyze the challenge binary and provide the following information in your write up:

- What constructs are used in this binary? Describe how you identified each construct.
- What common programming challenge is this binary implementing (calculating) ? The answer is one word. Flag format: flag{md5(to\_lower(answer))}

### Solution:

If else statement and while loop

As per usual, I opened Radare and typed 'aaaa' to analyze the binary

If statement: I started by analyzing main by typing 'V' and then 'p' twice to get to the instructions and scrolled to main().





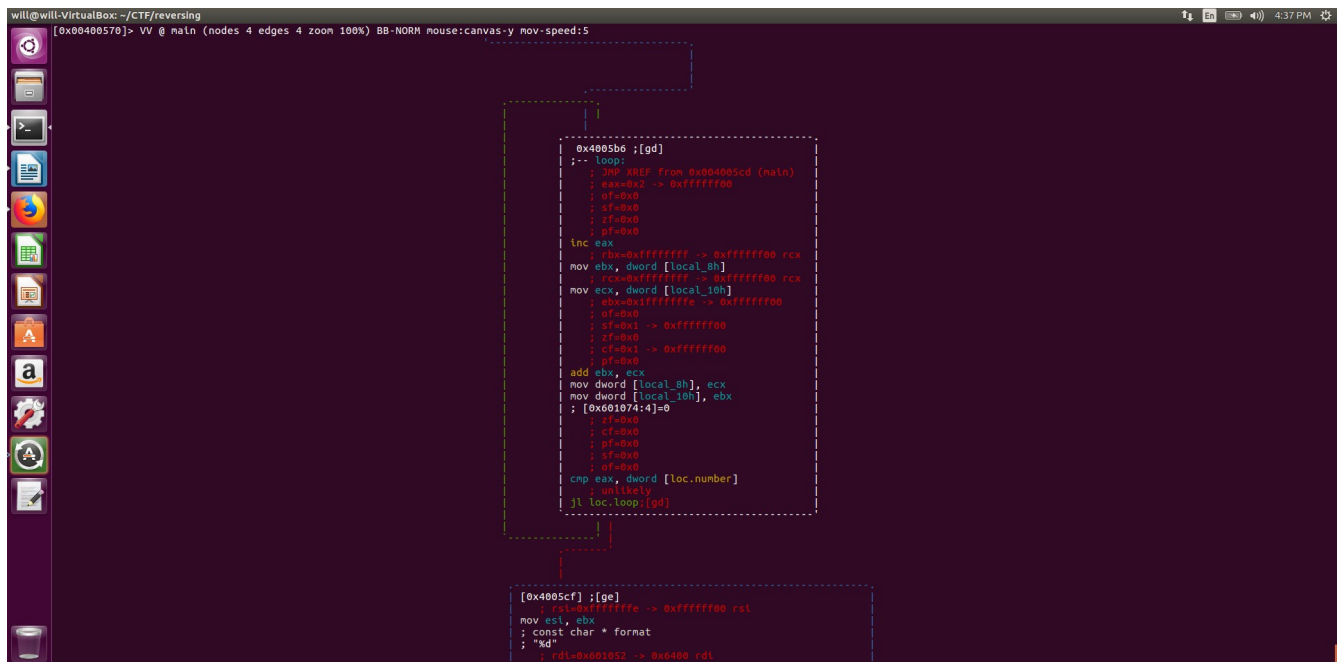


```

will@will-VirtualBox: ~/CTF/reversing
[0x004005c3] 10x 270 challenge> pd Sr @ main+83
: 0x004005c3 89d0f0 nov dword [local_10h], ebx
: 0x004005c6 3b0425741060. cmp eax, dword [loc.number]
: 0x004005c7 7c07 jl loc.loop
: 0x004005cf 89de mov esi, ebx
: 0x004005d1 48bfs2106000. movabs rdi, 0x001052
: 0x004005db b800000000 mov eax, 0
: 0x004005e0 e84bfeffff call sym.imp.printf
: 0x004005e5 c9 leave
: 0x004005e6 c3 ret
: 0x004005e7 60ef1f840000. nop word [rax + rax]
: 101
(fcn) sym.__libc_csu_init
sym.__libc_csu_init():
: DATA XREF from 0x00400480 (code?0)
: 0x004005f0 4157 push r15
: 0x004005f2 4156 push r14
: 0x004005f4 4189ff mov r15d, edi
: 0x004005f7 4155 push r15
: 0x004005f9 4154 push r12
: 0x004005fb 4c8d250e0820. lea r12, obj.__frame_dummy_init_array_entry
: 0x00400602 55 push rbp
: 0x00400603 488d2d0e0820. lea rbp, obj.__do_global_ctors_aux_fini_array_entry
: 0x00400608 53 push rbp
: 0x0040060b 4989f6 mov r14, rsi
: 0x0040060e 4989d5 mov r13, rdx
: 0x00400611 4c29e5 sub rbx, r12
: 0x00400614 48b3ce08 sub rsp, 8
: 0x00400618 48c1fd03 sar rbx, 3
: 0x0040061c e8dffdffff call sym.__init
: 0x00400621 48b5ed test rbp, rbp
: 0x00400624 7420 je 0x00400646
: 0x00400626 31db xor ebx, ebx
: 0x00400628 6f1f84000000. nop dword [rax + rax]
: JMP XREF from 0x00400643 (sym.__libc_csu_init)
: 0x00400630 4c89ea mov rdx, r13
: 0x00400633 4c89f6 mov rsi, r14
: 0x00400636 4489f6 mov edi, r15d
: 0x00400639 41114dc call qword [r12 + rbx*8]
: 0x0040063d 48b3c301 add rbx, 1
: 0x00400641 48b3eb cmp rbx, rbp
: 0x00400644 75ea jne 0x00400630
: JMP XREF from 0x00400624 (sym.__libc_csu_init)
: 0x00400646 48b3c408 add rsp, 8
: 0x0040064b 5b pop rbx
: 0x0040064d 5d pop rbp
: 0x0040064f 415c pop r12
: 0x00400651 415f pop r13
: 0x00400653 415e pop r14
: 0x00400655 c3 ret
: 0x00400656 60e0f1f84000. nop word cs:[rax + rax]
: 2
(fcn) sym.__libc_csu_fini
: 0x00400657 2

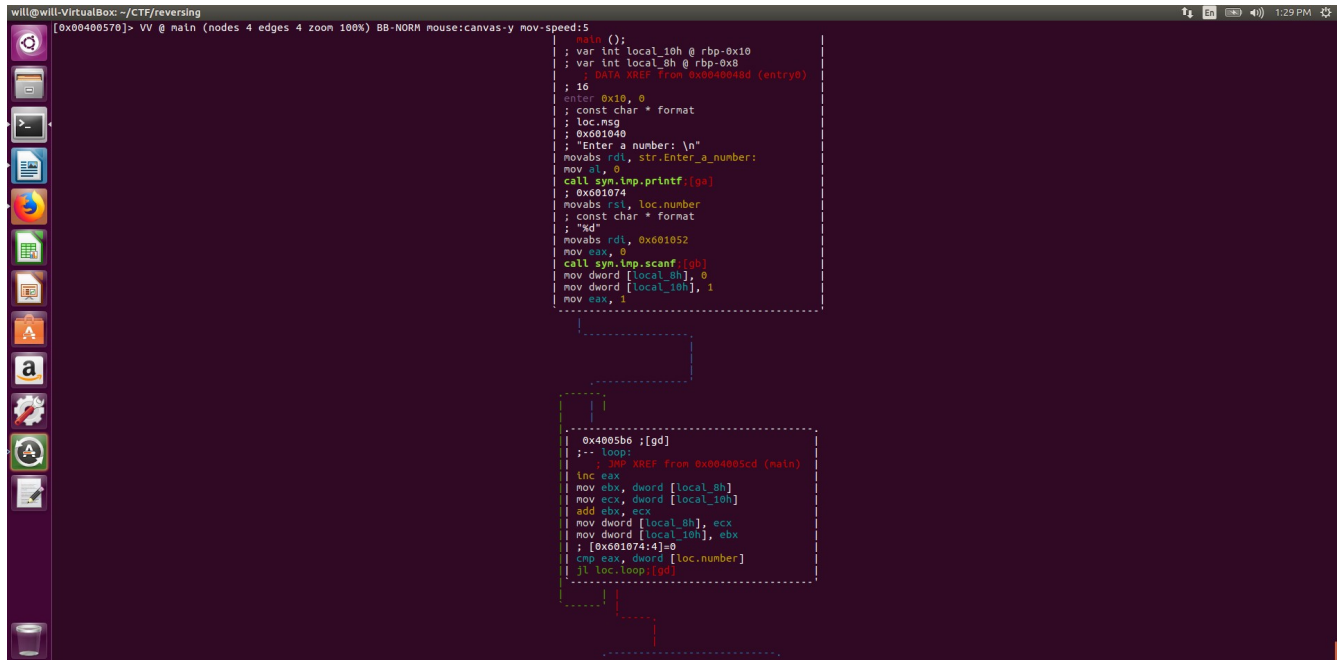
```

I opened that up in graph mode with ‘V’ and see the following:

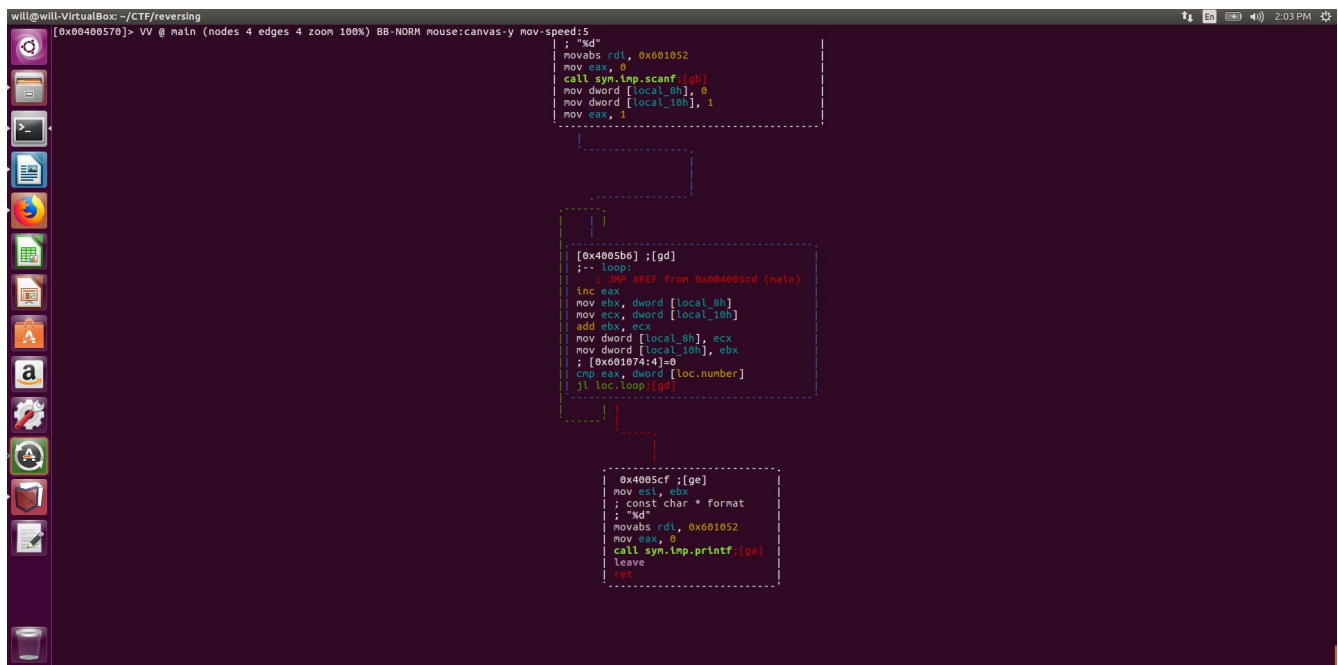


This is a while loop because at the end of a basic block there is a conditional jump that loops back to the start of a basic block. As indicated by the notes, this is the structure of a loop, since a block returns to itself. It is not a for loop because there is no incrementing block. Therefore, it must be a while loop. Also, Radare explicitly says that it is a loop.

The programming challenge this code is implementing is the Fibonacci sequence. I found out this information by analyzing main() and opening graph view.



In this portion of code, after the user enters a number, `local_8h` and `local_10h` are initialized to 0 and 1. These are the base cases for the Fibonacci sequence. `Eax` is initialized to 1 and is used as the counter for the loop. An addition is performed on 0 and 1, initially, and is stored in `ebx`. The result of the addition (`ebx`) and the second operand (`ecx`) are used as the operands for the next loop iteration. This is accomplished with the instructions `mov dword[LOCAL_8h], ecx` and `mov dword[LOCAL_10h], ebx`. Every iteration those local variables are used as the addition operands. The `cmp eax, dword[loc.number]` does the comparison for the loop to make sure the loop doesn't run more than `user_entered_number - 1` times. The loop runs as many times as the user entered number - 1 as indicated by `jl loc.loop;` After the loop the value of the final addition (`ebx`) is printed and then the program returns. This is all done in `main`.



I used the command 'chmod 777 challenge' followed by './challenge' to run the program and confirmed my results that the program is running a Fibonacci sequence. The Fibonacci value of 3 is 2 and the Fibonacci value of 10 is 55.

```
will@will-VirtualBox: ~/CTF/reversing
will@will-VirtualBox:~/CTF/reversing$ ./challenge
./challenge: command not found
will@will-VirtualBox:~/CTF/reversing$ challenge
challenge: command not found
will@will-VirtualBox:~/CTF/reversing$ ./challenge
Enter a number:
3
2will@will-VirtualBox:~/CTF/reversing$ ./challenge
Enter a number:
10
55will@will-VirtualBox:~/CTF/reversing$
```

After running Fibonacci through an md5 encoder found at <https://www.md5hashgenerator.com/> I get the flag.



Your Hash: **ef15d8edd00a6960c9c16937cbf14212**

Your String: fibonacci

Use this generator to create an MD5 hash of a string:

flag{ef15d8edd00a6960c9c16937cbf14212}

## Number

20 Points

### Problem:

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

Flag format: FIT{\*}

### Solution:

I started by downloading the file 'number' and changing the permissions so that I could run the program. For this problem I loaded the file 'number' into Radare2. I then proceeded to type 'aaaa' to analyze the binary and then 'V' to see the instructions. I paged through to get to the instructions list with 'p'. The first thing I did was search for main. I scrolled down and found this in main:

```
will@will-VirtualBox: ~/CTF/reversing
[0x0040064d 12x 270 number]> pd sr @ main
(fcn) nasm 282
main():
; var int local_20h @ rbp-0x20
; var int local_1ch @ rbp-0x1c
; var int local_1ah @ rbp-0x1a
; var int local_10h @ rbp-0x10
; var int local_bh @ rbp-0xb
; var int local_ah @ rbp-0xa
; var int local_9h @ rbp-0x9
; var int local_8h @ rbp-0x8
; var int local_7h @ rbp-0x7
; var int local_6h @ rbp-0x6
; var int local_5h @ rbp-0x5
; var int local_4h @ rbp-0x4
; var int local_3h @ rbp-0x3
; var int local_2h @ rbp-0x2
; var int local_1h @ rbp-0x1
; data xref from 0x0040057d
0x0040064d 55          push rbp
0x0040064e 4889e5       mov rbp, rsp
0x00400651 4883ec20     sub rsp, 0x20
0x00400655 c745f0000000 mov dword [local_10h], 0
0x0040065c c64546       mov byte [local_1h], 0x46 ; 'F' : 70
0x00400660 c645e4       mov byte [local_2h], 0x54 ; 'T' : 84
0x00400664 c645d03     mov byte [local_3h], 0x03 ; 'C' : 99
0x00400668 c645c49     mov byte [local_4h], 0x49 ; 'I' : 73
0x0040066c c645fb7b    mov byte [local_5h], 0x7b ; '!' : 123
0x00400670 c745e032e37. mov dword [local_20h], 0x31372e32
0x00400677 66c745e43832 mov word [local_1ch], 0x3238
0x0040067d c645e600     mov byte [local_1ah], 0
0x00400681 c645fa38     mov byte [local_6h], 0x38 ; '8' : 56
0x00400685 c645f931     mov byte [local_7h], 0x31 ; '1' : 49
0x00400689 c645f838     mov byte [local_8h], 0x38 ; '8' : 56
0x0040068d c645f732     mov byte [local_9h], 0x32 ; '2' : 50
0x00400691 c645f638     mov byte [local_ah], 0x38 ; '8' : 56
0x00400695 c645f57d     mov byte [local_bh], 0x7d ; '.' : 125
0x00400699 bff4074000   mov edi, str.hello__Please_collect_number ; 0x4007fa ; "hello! Please collect number" ; const char * s
0x0040069e e8dfeffff   call syn.imp.puts ; (1) ; int puts(const char *)
0x004006a3 48bd45f0     lea rax, [local_10h]
0x004006a7 4889c6       mov rsi, rax
0x004006aa bf12080000   mov edi, 0x400812
0x004006af b800000000   mov eax, 0
0x004006b4 e887feffff   call syn.imp.__tsoc99_scanf ; (2)
0x004006b9 8b45f0       mov eax, dword [local_10h]
0x004006bc 83e81a       sub eax, 0x1a
0x004006bf 8945f0       mov dword [local_10h], eax
0x004006c2 8b45f0       mov eax, dword [local_10h]
0x004006c5 3d00400000   cmp eax, 0x400 ; 1126
0x004006ca 0fb5060000   jne 0x400756 ; (3)
0x004006d0 0fbde451f   movsx eax, byte [local_1h] ; (4)
0x004006d4 89c7         mov edi, eax ; int c
0x004006d6 e825feffff   call syn.imp.putchar ; (4) ; int putchar(int c)
0x004006db 0fbde45fc   movsx eax, byte [local_4h]
```

The user is asked to collect a number. That value is stored in 'local\_10h' and then eax. Then, 0x1a is subtracted from the user entered number. This is done at 'sub eax, 0x1a'. Then a comparison is done on 0x466. If the result of the comparison isn't 0x400756 the program jumps and prints "No" (I found this out by typing in a wrong answer into the program). The comparison is checked with 'jne 0x400756'. When the jump isn't taken several character manipulations take place which I'm assuming is the flag. To get the input that allows the jump not to be taken I added 0x1a to 0x466. This is done to reverse the subtraction done earlier in the program. I used the calculator found at <http://www.calculator.net/hex-calculator.html?number1=1a&c2op=> and got the following results.

## Hex Calculator

### Hexadecimal Calculation—Add, Subtract, Multiply, or Divide

Result

Hex value: **480**

Decimal value: **1152**

1a	+	466	= ?
<div>Calculate</div>			

I entered the decimal value 1152, which is the result of the hex addition, into the program and got the flag.

```
will@will-VirtualBox: ~/CTF/reversing
will@will-VirtualBox:~/CTF/reversing$ ./number
hello!! Please collect number
1152
FIT{2.718281828}will@will-VirtualBox:~/CTF/reversing$
```

Flag = FIT{2.718281828}

## Got Time?

20 Points

### Problem:

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

Flag format: TAMPA{\*}

### Solution:

The first thing I did was download the file 'Got\_Time'. I ran './Got\_Time' after changing the permissions to allow execution.

```
will@will-VirtualBox: ~/CTF/reversing
Decrypting.. 99.94%
Estimated Time Left:1007.42 Minutes
^Z
[1]+  Stopped                  ./Got_Time
will@will-VirtualBox:~/CTF/reversing$
```

The program is decrypting the flag, but it takes 1000 minutes. This isn't practical, so I will have to analyze it with Radare. After opening Radare, I typed 'aaaa' to analyze the binary. I then used the 'afl' command to view functions.



```

[0x000006a0]> afl
0x00000000 3 73 -> 75 sym.imp.__libc_start_main
0x00000018 3 23 sym._init
0x00000040 1 6 sym.imp.puts
0x00000050 1 6 sym.imp.strlen
0x00000060 1 6 sym.imp.printf
0x00000070 1 6 sym.imp.malloc
0x00000080 1 6 sym.imp.usleep
0x00000090 1 6 sub.__cxa_finalize_248_690
0x000000a0 1 43 entry0
0x000000d0 4 50 -> 44 sym.deregister_tm_clones
0x000000e0 4 66 -> 57 sym.register_tm_clones
0x000000f0 5 50 sym.__do_global_dtors_aux
0x00000100 4 48 -> 42 entry1.init
0x00000110 4 177 sym.xorencrypt
0x00000120 4 324 main
0x00000130 4 101 sym.__libc_csu_init
0x00000140 1 2 sym.__libc_csu_fini
0x00000150 1 9 sym._fini
[0x000006a0]>

```

I noticed a function called 'sym.xorencrypt' and 'main'. This lead me to believe the flag is being xor encoded. I seeked to main with 's 0x881' which is the address of main. I then typed 'V' and then hit 'p' a couple times to view the instructions. After scrolling down main I notice a ton of local variables which are characters that are initialized right before the call to 'sys.xorencrypt'.

```

will@will-VirtualBox: ~/CTF/reversing
[0x000008d3 258 272 Got_Tline]> pd $r 0 main+82
: 0x000008d3 f30f4c1 divss xmm0, xmm1
: 0x000008d7 f30f1145f0 movss dword [local_10h], xmm0
: 0x000008dc 48bda45f4 cvtsd2sd xmm0, dword [local_ch]
: 0x000008e1 48bd3da0100. lea rdi, str.Decrypting...2f ; 0xa72; 'Decrypting... N.2fNN\n'; const char * format
: 0x000008e3 b8100000 nov eax, 1
: 0x000008e6 e8e6fdfff call sym.imp.printf ;[1]; int printf(const char *format)
: 0x000008f2 f30f4c100. cvtsd2sd xmm0, dword [local_10h]
: 0x000008f7 48bd3da0100. lea rdi, str.Estimated_Time_Left:_2f_Minutes ; 0xa88; 'Estimated Time Left:N.2f Minutes\n'; const char * format
: 0x000008fa b8100000 nov eax, 1
: 0x000008fd e858fdfff call sym.imp.printf ;[1]; int printf(const char *format)
: 0x00000900 bfa060100 nov edi, 0x180a0
: 0x0000090d e86efdfff call sym.imp.usleep ;[2];
: 0x00000912 48bd3da0100. lea rdi, str.e_H_e_3 ; 0xaa; const char * format
: 0x00000919 b8000000 nov eax, 0
: 0x0000091e e83dfdfff call sym.imp.printf ;[1]; int printf(const char *format)
: 0x00000923 83dffc01 sub dword [local_4h], 1
: 0x00000927 jmp xref from 0x00000900 (main)
: 0x00000927 837dfc00 cmp dword [local_4h], 0
: 0x0000092b 0f8f0a7ffff jg 0x89b ;[3]
: 0x00000931 48c745d8000. mov qword [local_38h], 0
: 0x00000939 48c745d8000. mov qword [local_28h], 0
: 0x00000941 c745e000000. mov dword [local_20h], 0
: 0x00000948 66c745e4000. mov word [local_1ch], 0
: 0x0000094e c645d03d mov byte [local_38h], 0x3d ; 'd'; rsi
: 0x00000952 c645d101 mov byte [local_2fh], 0x01 ; 'a'
: 0x00000956 c645d226 mov byte [local_2eh], 0x26 ; 'A'; section_end..comment
: 0x0000095a c645d33e mov byte [local_2dh], 0x3e ; 'e'
: 0x0000095e c645d42e mov byte [local_2ch], 0x2e ; '.'
: 0x00000962 c645d50c mov byte [local_2bh], 0xc ; 'c'
: 0x00000966 c645d601 mov byte [local_2ah], 0x01 ; 'a'
: 0x0000096a c645d74c mov byte [local_29h], 0x4c ; 'L'
: 0x0000096e c645d83e mov byte [local_28h], 0x3e ; 'e'
: 0x00000972 c645d927 mov byte [local_27h], 0x27 ; '7'
: 0x00000976 c645da7f mov byte [local_26h], 0x7f ; 'f'
: 0x0000097a c645db23 mov byte [local_25h], 0x23 ; '3'
: 0x0000097e c645dc20 mov byte [local_24h], 0x20 ; '0'
: 0x00000982 c645dd72 mov byte [local_23h], 0x72 ; 'r'
: 0x00000986 c645de39 mov byte [local_22h], 0x39 ; '9'
: 0x0000098a c645df37 mov byte [local_21h], 0x37 ; '7'
: 0x0000098e c645e032 mov byte [local_20h], 0x32 ; '2'
: 0x00000992 c645e109 mov byte [local_1fh], 0x09 ; '9'
: 0x00000996 c645e227 mov byte [local_1eh], 0x27 ; '7'
: 0x0000099a c645e324 mov byte [local_1dh], 0x24 ; '4'
: 0x0000099e c645e400 mov byte [local_1ch], 0 ; ' '
: 0x000009a2 48bd45d0 lea rax, [local_30h]
: 0x000009a6 4889c7 mov rdi, rax
: 0x000009a9 e877fefff call sym.xorencrypt ;[4]
: 0x000009ae 488945e8 mov qword [local_18h], rax
: 0x000009b2 488b45e8 mov rax, qword [local_18h]
: 0x000009b6 4889c7 mov rdi, rax ; const char * s
: 0x000009b9 e882fcfff call sym.imp.puts ;[5]; int puts(const char *s)
: 0x000009bc b8000000 nov eax, 0
: 0x000009c3 c9 leave

```

These have to be important. I then scroll up to the XOR function and notice the string "i know how to use cat too"

```

will@will-VirtualBox: ~/CTF/reversing
[0x000007c8 22% 270 Got_Time]> pd $r @ entry1.init+40
: 0x000007c8 ffd0 call rax
: 0x000007ca 5d pop rbp
<= 0x000007cb e940ffffff jmp sym.register_tm_clones ;[1]
(fcn) sym.xorencrypt 177
sym.xorencrypt ();
; var int local_38h @ rbp-0x38
; var int local_28h @ rbp-0x28
; var int local_20h @ rbp-0x20
; var int local_18h @ rbp-0x18
; var int local_10h @ rbp-0x10
; var int local_4h @ rbp-0x4
; CALL XREF From 0x000009a9 (main)
0x000007d0 55 push rbp
0x000007d1 4889e5 mov rbp, rsp
0x000007d4 4883ec40 sub rsp, 0x40 ; '@'
0x000007d8 48897dc8 mov qword [local_38h], rdi
0x000007dc 488d05750200 lea rax, str.i_know_how_to_use_cat_too ; 0xa58 ; "i know how to use cat too"
0x000007e3 488945f0 mov qword [local_10h], rax
0x000007e7 488b45c8 mov rax, qword [local_38h]
0x000007f1 488b45c8 mov rax, qword [local_38h]

```

The rest of the function uses many local variables and looks really intimidating. It uses all sorts of different hex values, moves, and adds. I tried the value '0x40' as the key because it is the only hard coded byte in the function. This didn't work. Since I can't seem to find a solid key for the XOR encryption in the function, I will choose to ignore the rest of the code. "i know how to use cat too" appears to be the important part of the function because it is the only string. Going out on a limb, I wrote a python script which XORs the first character of the string "i know how to use cat too" with the first local variable in main's hex value, XORs the second character of the string with the second local variables in main's hex value, and continued this process for every local variable in main. The code is attached as 'reversexor.py'. I chose to ignore the very first local variable's value because it was 0, and I assumed it was generated on accident by Radare. Below is a brief part of the code.

```

z=0x3d^ord('i')
print chr(z)
z=0x61^ord(' ')
print chr(z)
z=0x26^ord('k')
print chr(z)
z=0x3e^ord('n')
print chr(z)
z=0x2e^ord('o')
print chr(z)

```

The script used every character of the string 'i know how to use cat too' except for ' too'. After running the python script I found the flag.

```
will@will-VirtualBox: ~/CTF/reversing
will@will-VirtualBox:~/CTF/reversing$ python reversexor.py

T
A
M
P
A
{
A
$
A
P
_
W
O
R
L
D
W
I
D
E
}

will@will-VirtualBox:~/CTF/reversing$
```

Flag = TAMPA{A\$AP\_WORLDWIDE}