CRACKME WALKTHROUGHS

maxpl0it

What is this?!

- At F-Secure, we have regular hacknights where people teach each other things in order to skill up the company as a whole.
- Due to the current global circumstances, we were unable to do an in-person one, so we decided to try something new: A live stream.
- The aim of this particular hacknight (or as I've called it, cracknight) was to get people into reverse engineering using freely available tools in a fun way.
- The targets in question were 6 binary challenges for beginners. The aim was to get past the serial key!
- Since the event was recorded for F-Secure, I wanted to release both these challenges AND a writeup of how you'd go about solving them.

$ whoami

- Max (@maxpl0it)
- Works at F-Secure in the Research Team
- Lover of binary exploitation
- Pwn2own Tokyo 2019 - F-Secure Team
    - Bit of a router obsession

$ whyami -?

- Knowledge sharing is VERY important
- A lot of people aren’t sure how to get started with anything binary-related
- The aim is to get people started by showing them the tools and basics so that they feel confident enough to go out and further their knowledge
- Although this was originally an F-Secure hacknight, it was decided that we’d the challenges - Something fun to do for the rest of the internet
- I decided the challenges themselves weren’t going to be of any use if there wasn’t a document showing how to do them (Thus, this was born!)

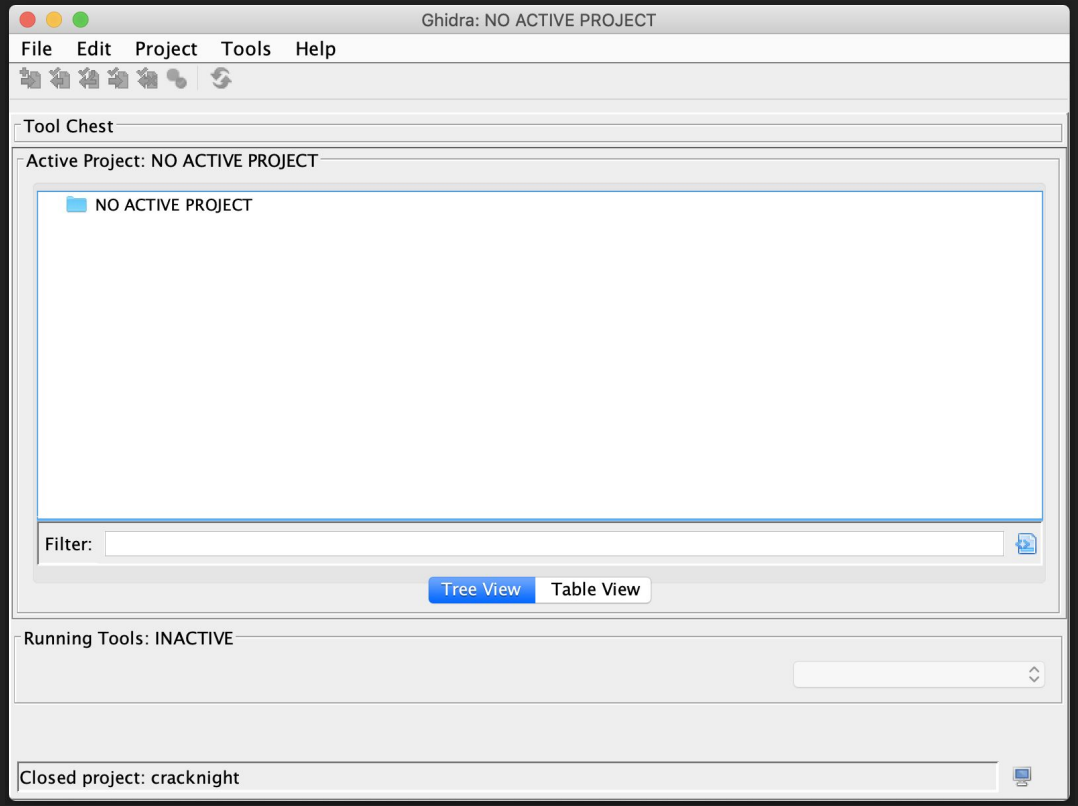
Setup

You Will Need:

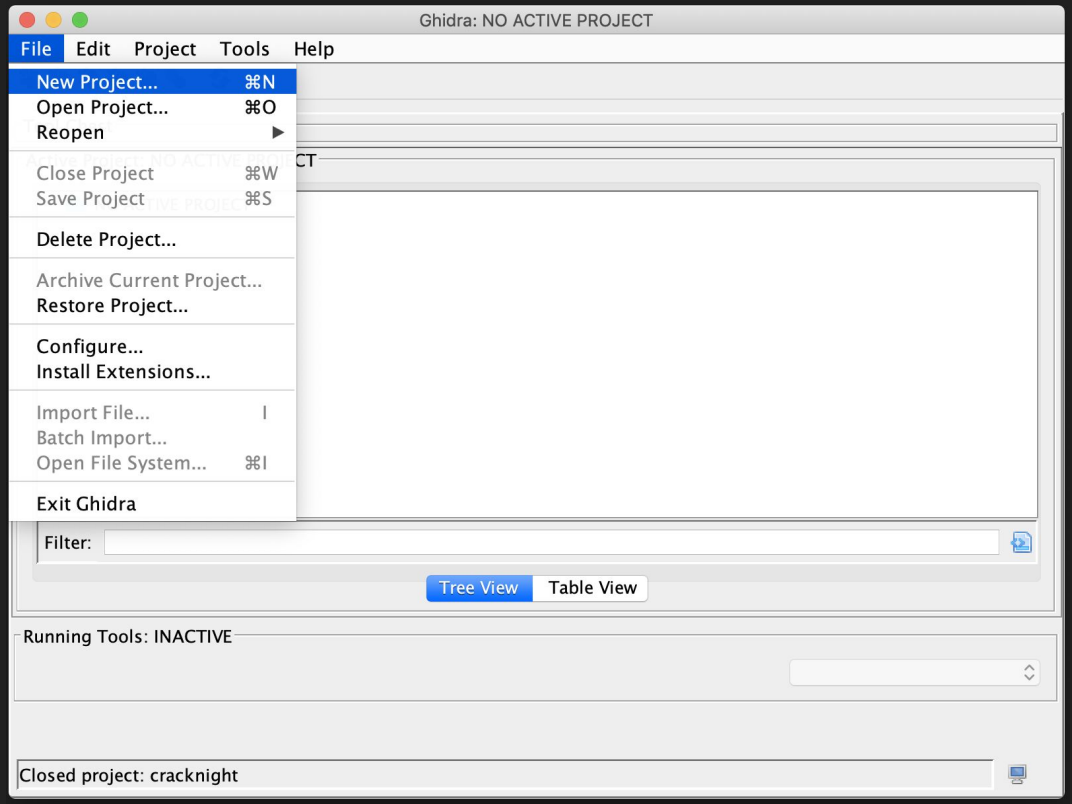
- 64-bit Linux VM (I chose Ubuntu 19)
  - Install GDB on it (apt-get install gdb) - This will be our debugger
- Ghidra (This is our disassembler && decompiler)
  - Either on the host or on the VM itself
  - If Ghidra is on the host, then make sure you've got a way to easily transfer files between the host and the VM

Ghidra Setup

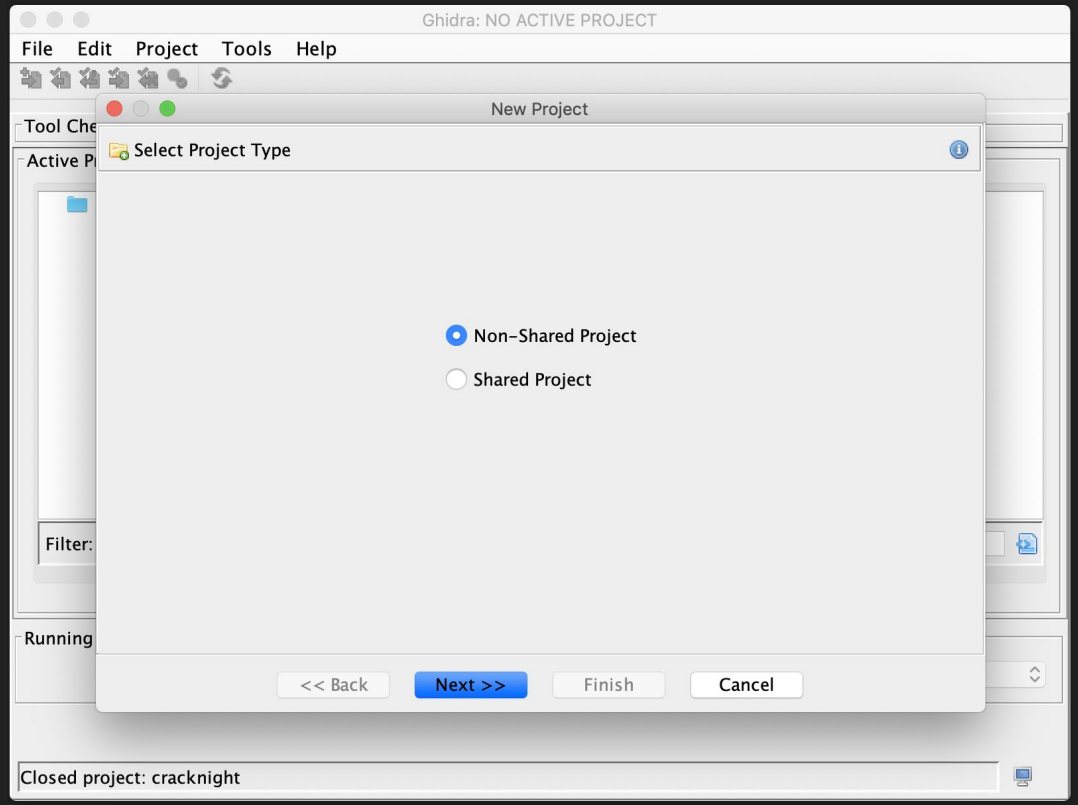
1. Open Ghidra



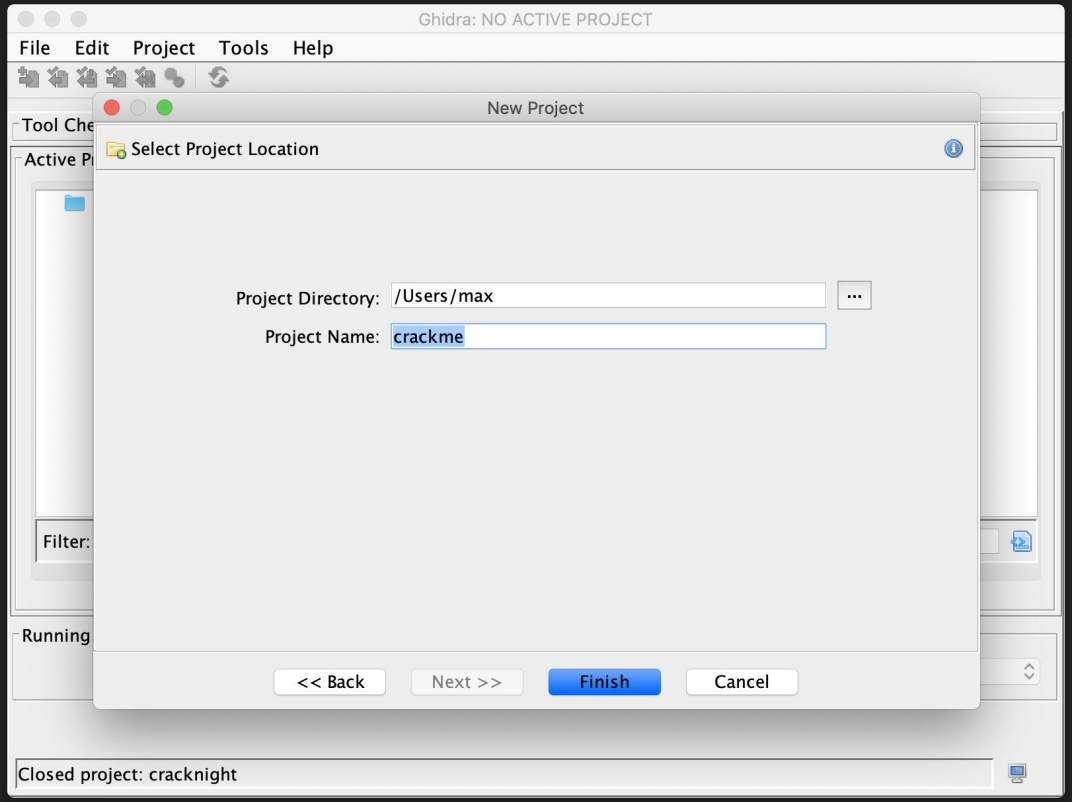
2. Create a new project



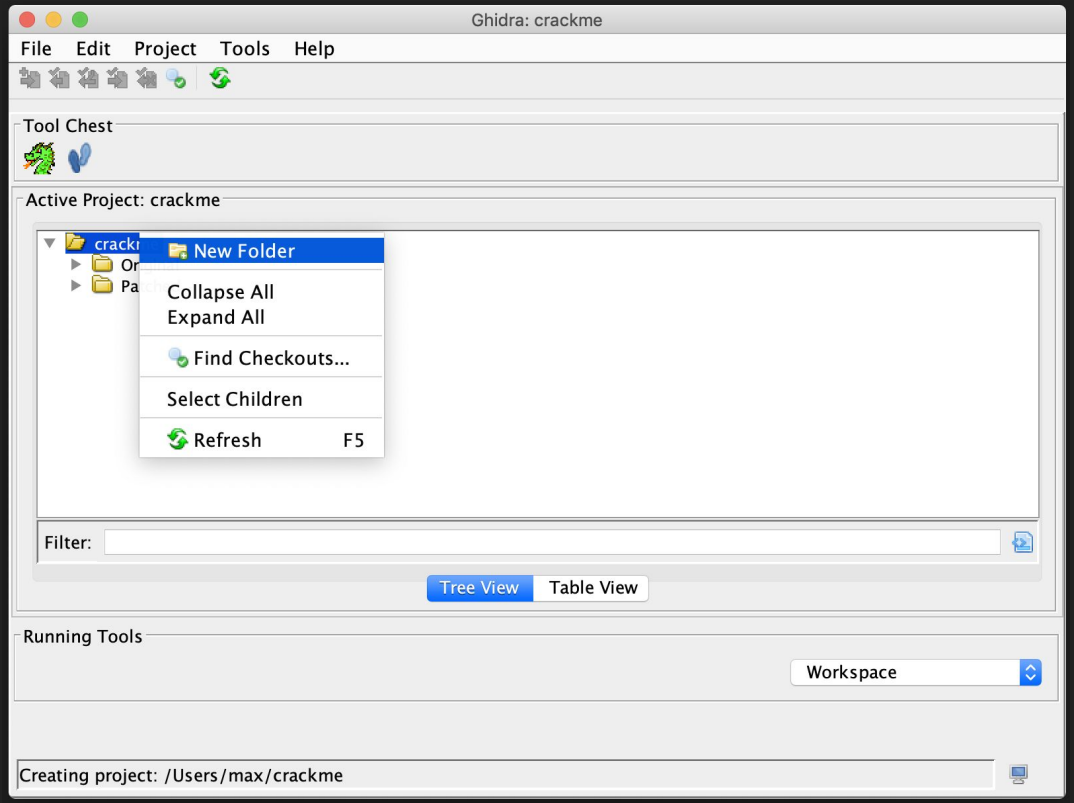
3. Non-Shared


Ghidra: NO ACTIVE PROJECT
File
Edit
Project
Tools
Help
New Project
Select Project Type
Non-Shared Project
Shared Project
<< Back
Next >>
Finish
Cancel
Filter:
Running
Closed project: cracknight

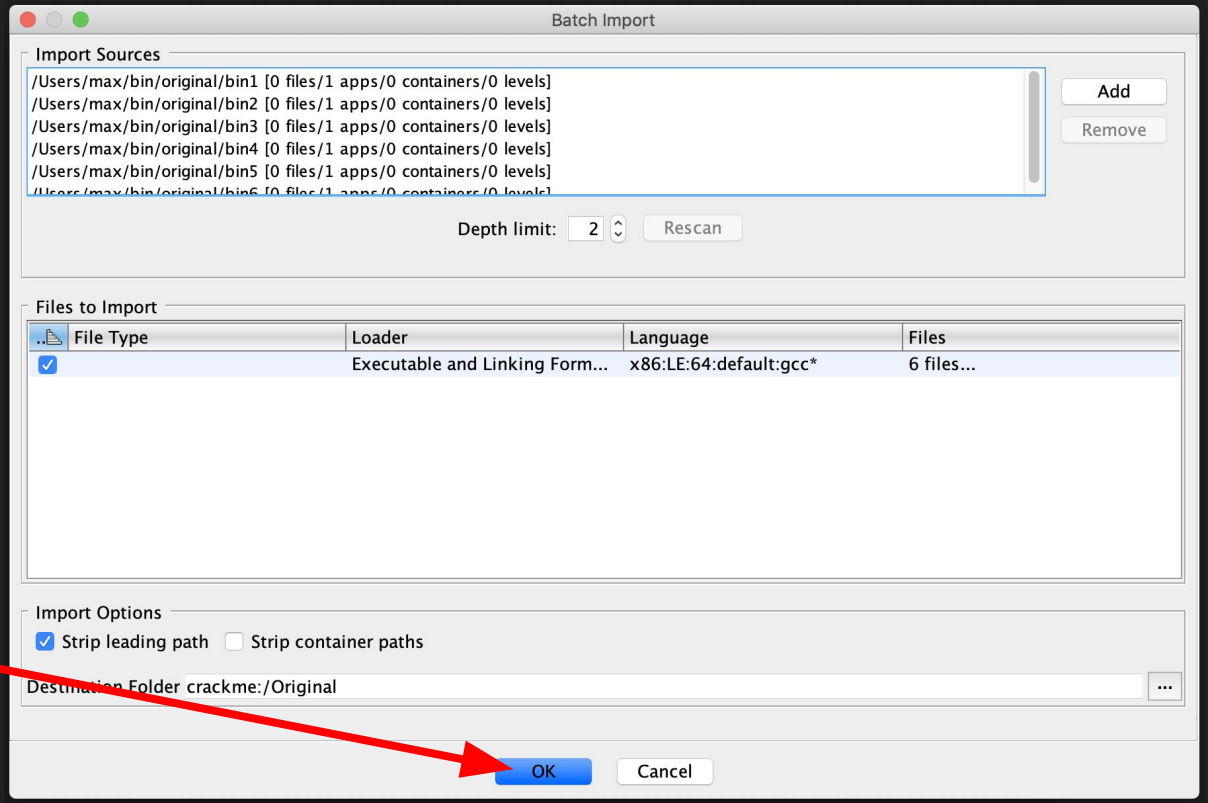
4. Pick a project name



5. Create folders for “Original” && “Patched” binaries



6. Drag ‘n Drop the binaries into the Original folder

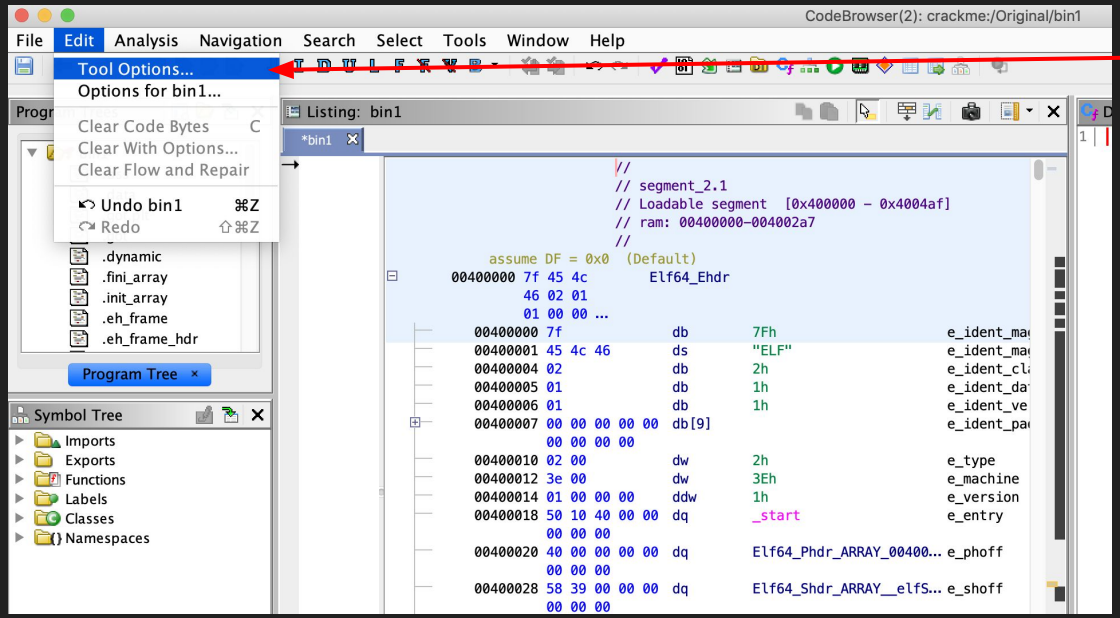


Click OK

8. Setup Highlighting in Tool Options

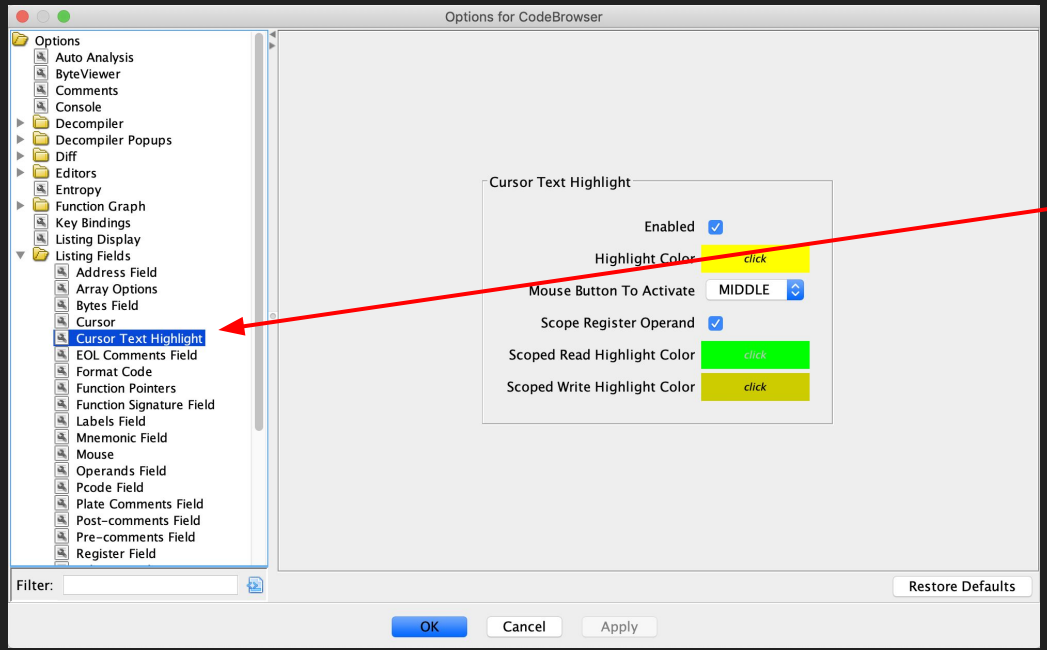
- Highlighting is where you click on a variable and it also highlights other uses of the variable
- This is incredibly useful for when you have large functions and want to see whether or where a variable is used without having to strain your eyes searching!

8. Setup Highlighting in Tool Options



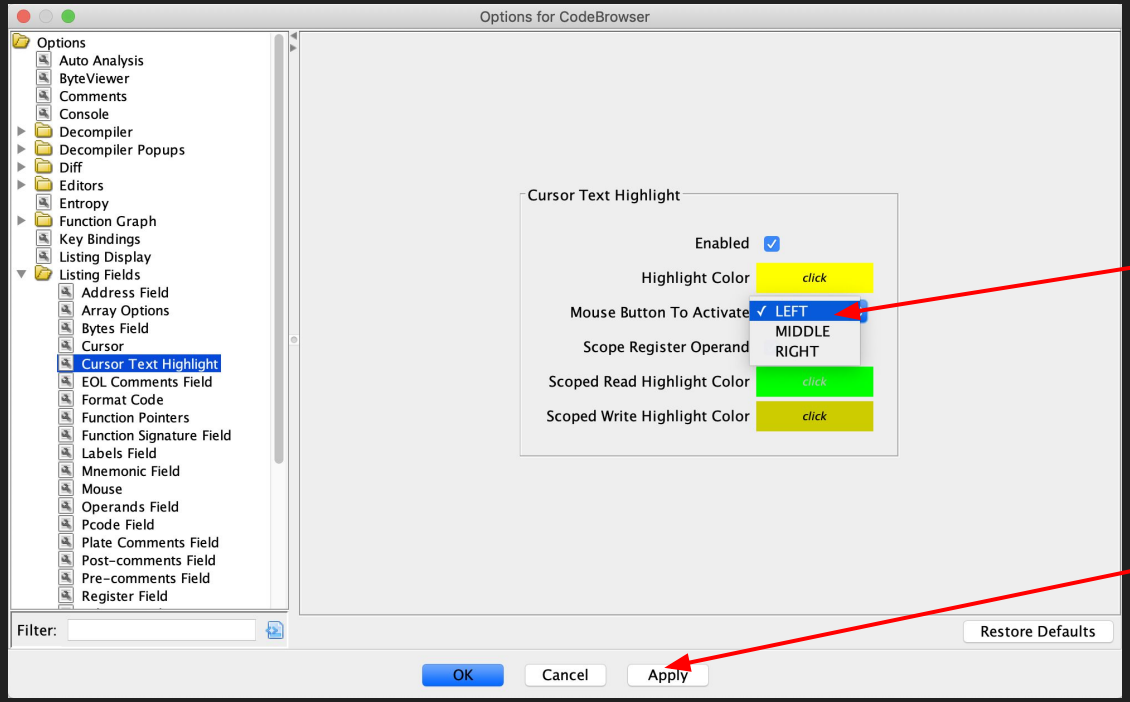
Open up one of the binaries and then click on Tool Options

8. Setup Highlighting in Tool Options



Go to Listing Fields -> Cursor Text Highlight

8. Setup Highlighting in Tool Options



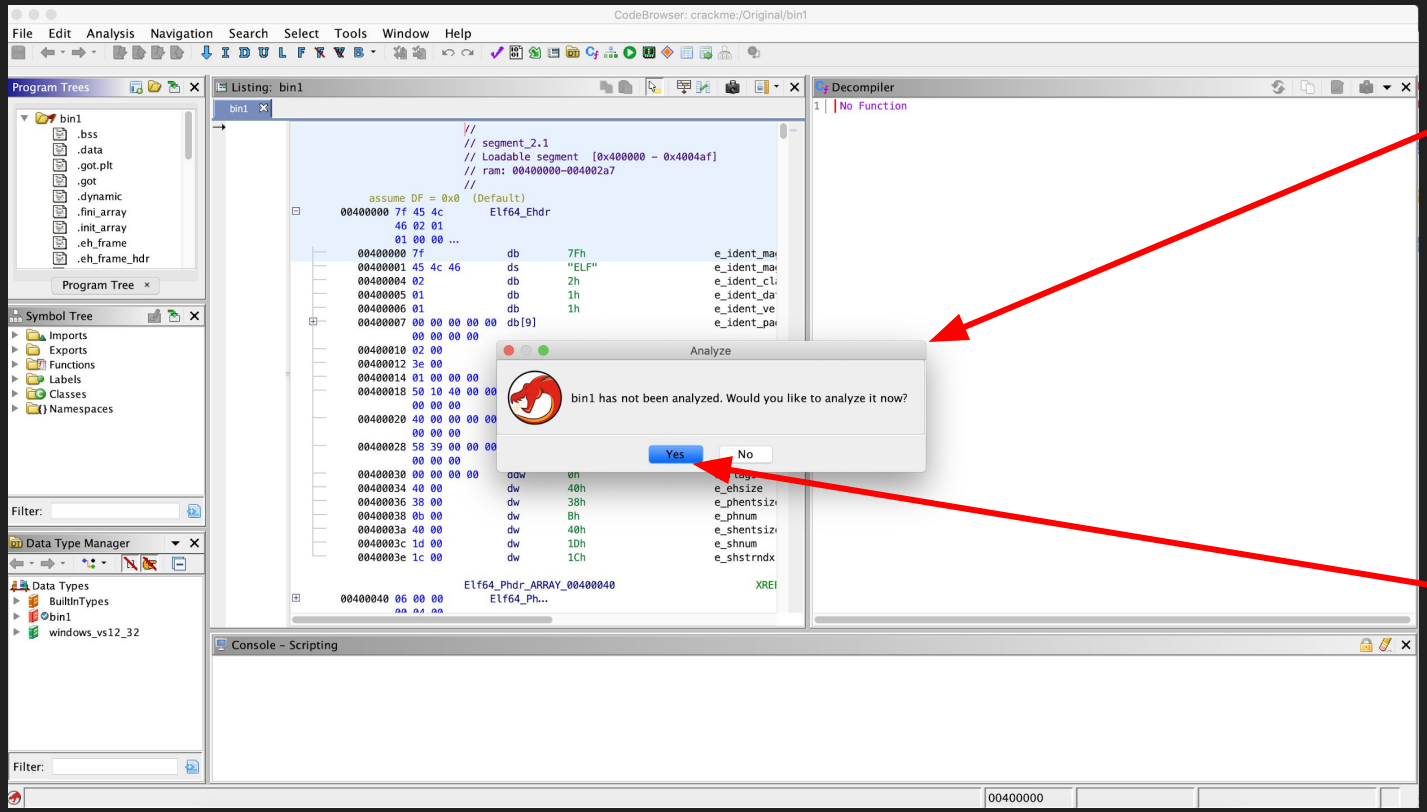
Change Mouse Button To Activate to “Left”

Apply the change

bin1

So let’s get started!

1. Open up bin1 by double clicking on it in Ghidra

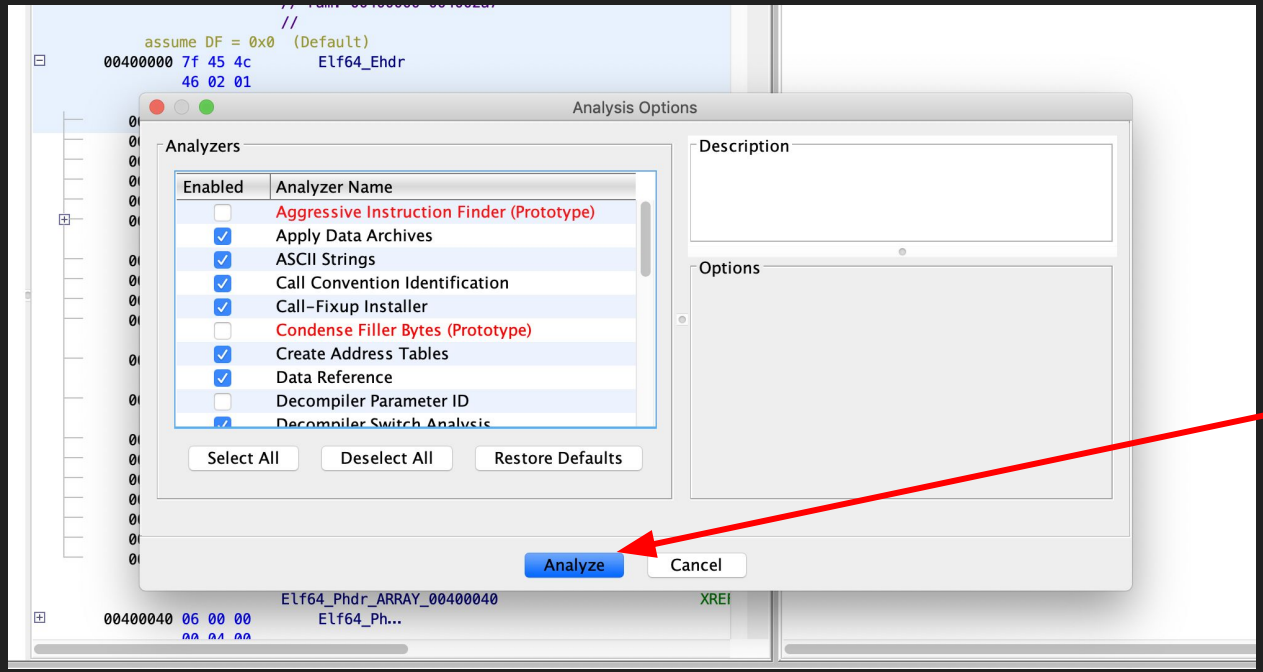


You will be prompted to analyze the file

This is how Ghidra identifies instructions, functions, and links between things

Click “Yes”

2. Analyze!!

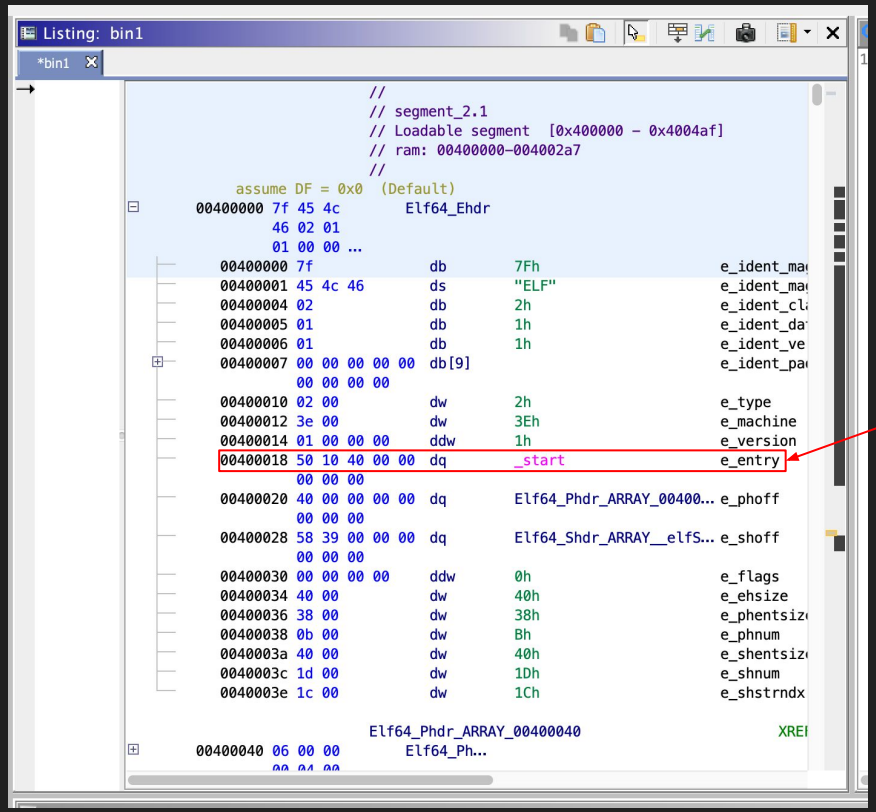


The default options work really well for most binaries, so go ahead and click Analyze!

Where do we _*start*?

- ELF binaries have a bunch of headers.
- One of these headers is called e_entry
  - This defines the *entry point* of the program - Where it begins executing

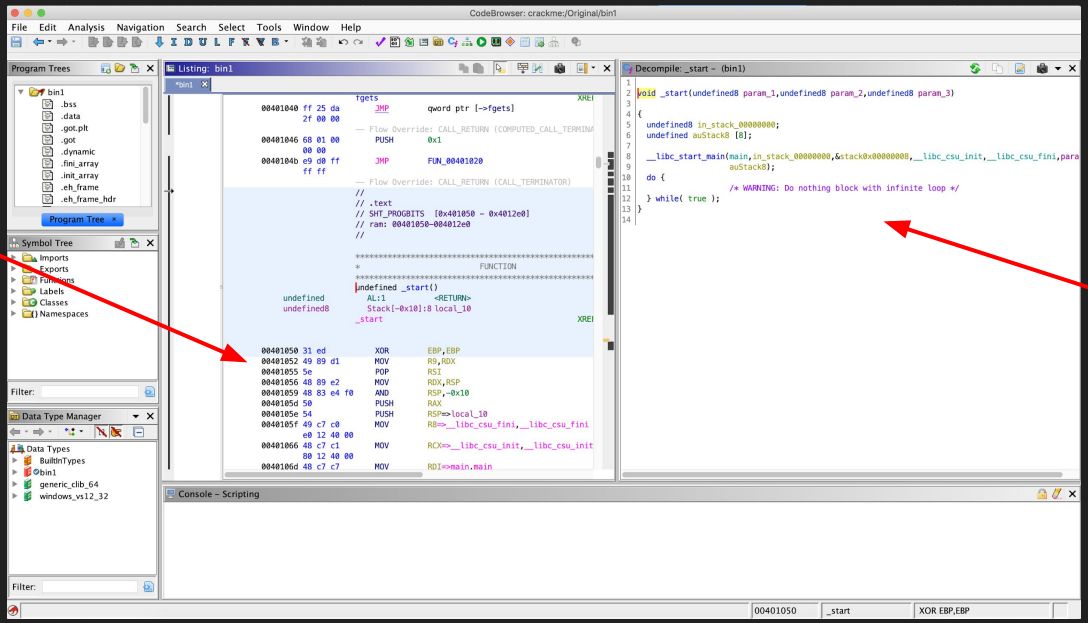
Where do we _*start*?



Here it is!

3. Double click on _*start*

Disassembly - The assembly instructions in the program

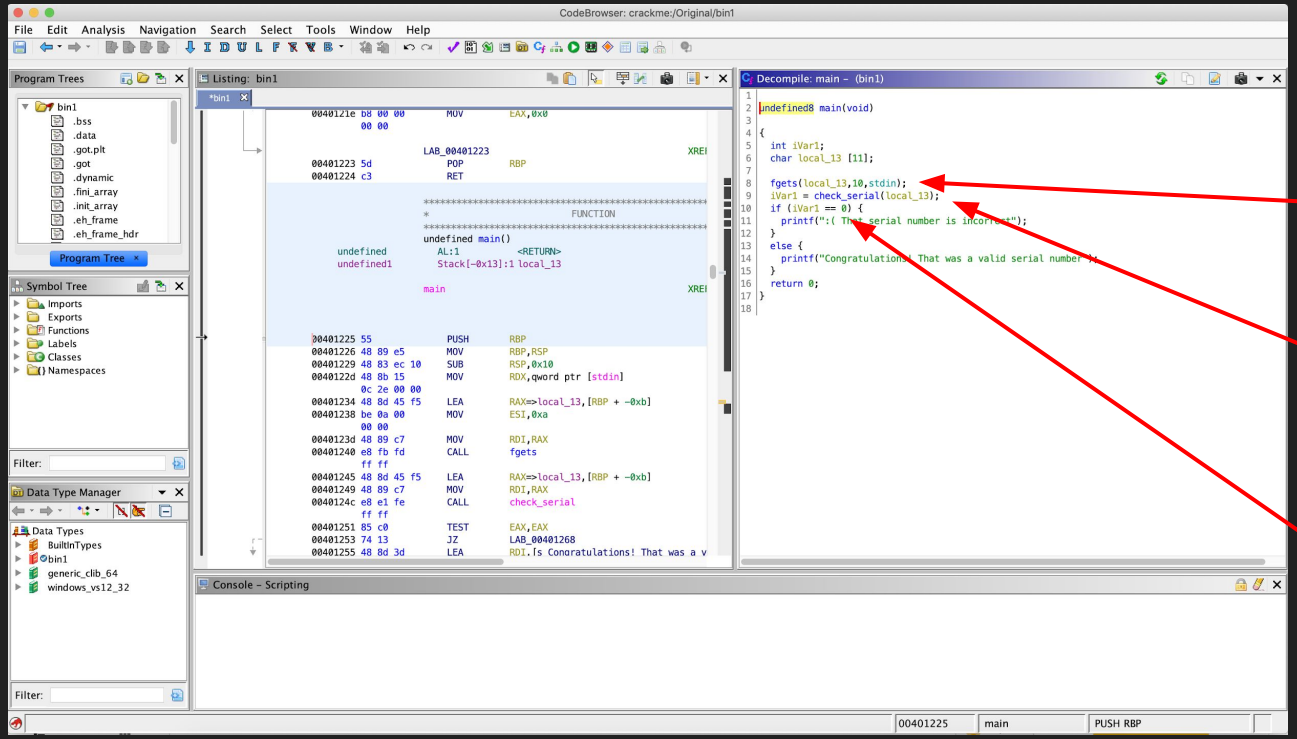


Decompiler - C pseudocode that is generated based on the assembly

Errr, what is all this?

- The *start* function makes a call to *__libc_start_main*
- Might look a little threatening but it's fairly simple!
  - Libc is the C Standard Library.
  - It contains a bunch of core functionality, including functions such as printf() and open()
  - *__libc_start_main* does three things:
    - Defines the *main* function
    - Defines the init functions - Small procedures that are run before the main function is executed (such as initializing memory structures)
    - Defines the fini functions - Small procedures that are run after the main function is executed (such as cleaning up leftover memory structures)

4. Double click on *main*



So this is the main function! It does several things:

1) It calls *fgets*

2) It calls *check_serial* with the variable *local_13*

3) It checks whether the result of *check_serial* is 0

fgets? Can you fclarify what this fdoes?

- *fgets(buffer, n, fd);*
- fgets is a function that reads data from a file descriptor into a buffer
- It reads up to *(n-1)* bytes (The last byte is reserved for a null terminator)

- But what value do we use for *fd*?
  - File Descriptors can be thought of as numbers that represent open files (Because, you know, “everything is a file in Linux”)
  - The input you provide a program, the output it gives, and the errors it gives are all file descriptors too! These three even had set file descriptor numbers:
  - 0 - stdin (Standard Input -> what you’re typing)
  - 1 - stdout (Standard Output -> What the program prints to the screen)
  - 2 - stderr (Standard Error -> A separate file descriptor that prints to the screen when displaying errors)

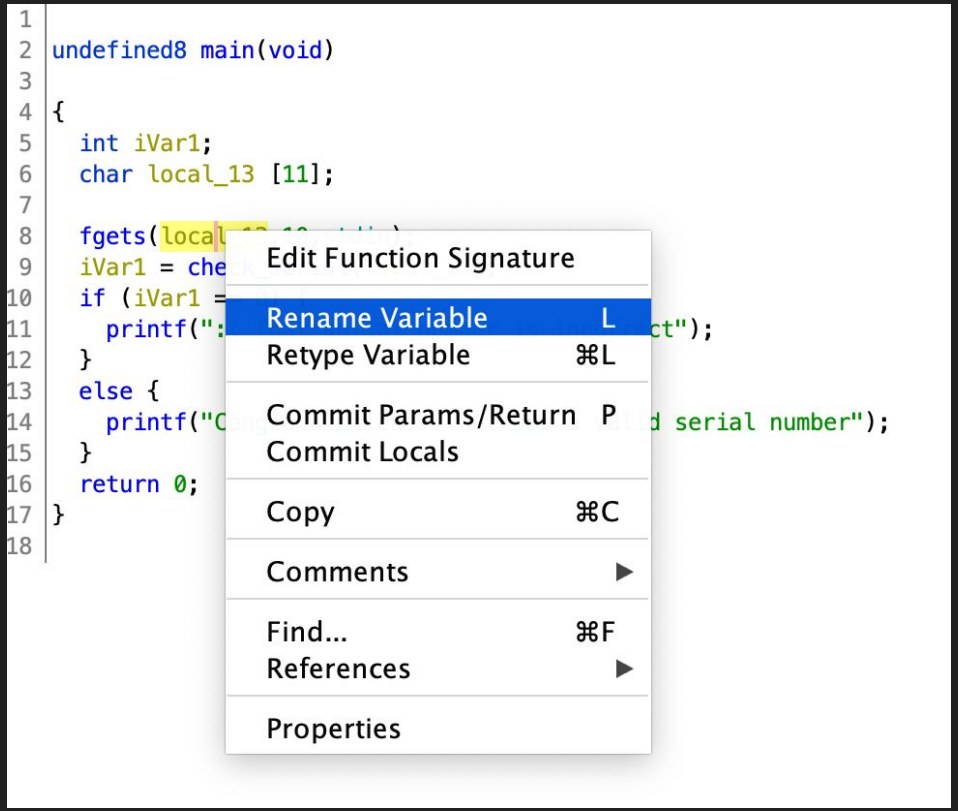
```
1
2  undefined8 main(void)
3
4  {
5    int iVar1;
6    char local_13 [11];
7
8    fgets(local_13,10,stdin);
9    iVar1 = check_serial(local_13);
10   if (iVar1 == 0) {
11     printf(":( That serial number is incorrect");
12   }
13   else {
14     printf("Congratulations! That was a valid serial number");
15   }
16   return 0;
17 }
18
```

Using what we've just learned, we know that:
fgets(local_13,10,stdin) == "Read 9 bytes from the *input* into buffer *local_13*"

Since *check_serial* is called with *local_13*, we can make the assumption that *local_13* is going to contain our serial key

Let's go ahead and rename it

4. Rename *local_13* to *serial_key*



- Renaming variables will make code much easier to understand.
- A lot of the time, you'll come across large functions that don't have nicely-named symbols such *check_serial*.
- Once you start figuring out what each variable and function is used for and renaming them, eventually you'll have a much clearer idea of what other functions do!

What does *check_serial* do?

```
1
2  ulong check_serial(void)
3
4  {
5    uint local_10;
6    uint local_c;
7
8    local_c = 0x7fafaf31;
9    local_10 = 0;
10   while ((int)local_10 < 10) {
11     if ((local_10 & 1) != 0) {
12       local_c = (int)local_c % (local_10 * local_10 + local_10);
13     }
14     if (3 < (int)local_10 % 0xe) {
15       local_c = local_c + (2 - local_10 * local_10);
16     }
17     if ((int)local_10 % 9 == 0) {
18       local_c = (int)local_10 % 0x4a + 0xfaU ^ local_10;
19     }
20     local_10 = local_10 + 1;
21   }
22   return (ulong)(local_c == 0x662);
23 }
24
```

NOPE.

What does *check_serial* do?

```
1
2  ulong check_serial(void)
3
4  {
5    uint local_10;
6    uint local_c;
7
8    local_c = 0x7fafaf31;
9    local_10 = 0;
10   while ((int)local_10 < 10) {
11     if ((local_10 & 1) != 0) {
12       local_c = (int)local_c % (local_10 * local_10 + local_10);
13     }
14     if (3 < (int)local_10 % 0xe) {
15       local_c = local_c + (2 - local_10 * local_10);
16     }
17     if ((int)local_10 % 9 == 0) {
18       local_c = (int)local_10 % 0x4a + 0xfaU ^ local_10;
19     }
20     local_10 = local_10 + 1;
21   }
22   return (ulong)(local_c == 0x662);
23 }
24
```

Incorrect number of arguments

Crazy modulus magic

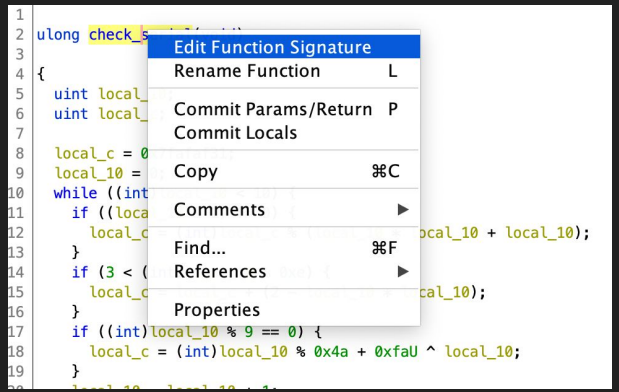
N.O.P.E.

Why are the arguments messed up?

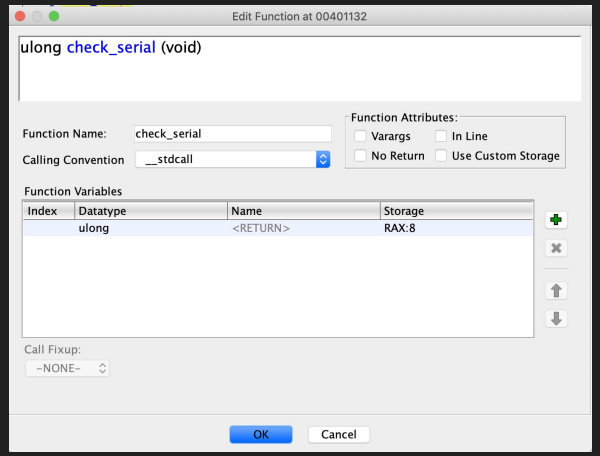
- Decompilers aren’t always correct.
- They work by making lots of guesses and assumptions and detecting patterns.
  - “Ah, while this bit of assembly is repeating, this variable is decrementing. This must be a for-loop!”
- Although decompilers are incredibly helpful for quickly understanding functions, learning assembly will help tremendously later on.
  - If you can understand a function without relying on a decompiler, you’ll be able to understand the code much better (and even understand where the decompiler has failed)

Can we fix the arguments?

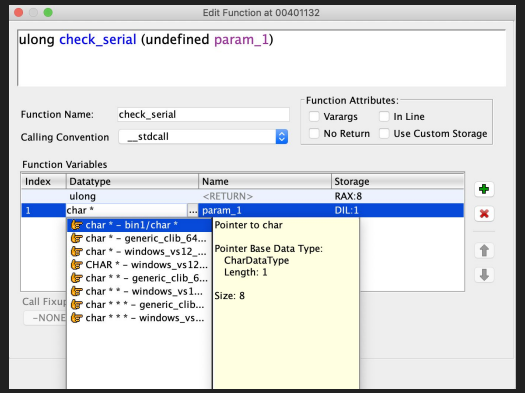
1) Edit Function Signature



2) Click **+** to add an argument

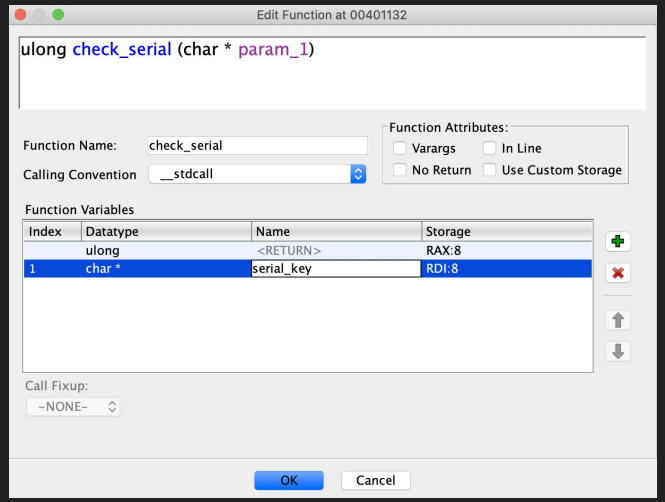


3) Change the Datatype to *char **



Can we fix the arguments?

4) Rename the parameter



So… What happened?

```
1
2  ulong check_serial(char *serial_key)
3
4  {
5    uint local_10;
6    uint local_c;
7
8    local_c = 0x7fafaf31;
9    local_10 = 0;
10   while ((int)local_10 < 10) {
11     if ((local_10 & 1) != 0) {
12       local_c = (int)local_c % (local_10 * local_10 + local_10);
13     }
14     if (3 < (int)local_10 % 0xe) {
15       local_c = local_c + (2 - local_10 * local_10);
16     }
17     if ((int)local_10 % 9 == 0) {
18       local_c = (int)local_10 % 0x4a + 0xfaU ^ local_10;
19     }
20     local_10 = local_10 + 1;
21   }
22   return (ulong)(local_c == 0x662);
23 }
```

Turns out the argument wasn't used anywhere anyway! This serial key will always fail :(

What can we learn from this?

- Software cracking isn’t always about bypassing licensing features, sometimes it’s just about making the program work.
- An example of this is a form of copy protection that was used by developers on the Commodore 64
  - In order to check whether the disk was legitimate, the software would try to slam the read/write head. This could lead to the head becoming misaligned and actually breaking the system.
  - Software cracking was used to get around this in order to avoid the system being broken.

Where do we go from here?

- It’s time for the moment you’ve all been waiting for…

THE PATCHING

Let’s understand the assembly

```
00401245 48 8d 45 f5     LEA        RAX=>serial_key,[RBP + -0xb]
00401249 48 89 c7        MOV        RDI,RAX
0040124c e8 e1 fe        CALL       check_serial
         ff ff
00401251 85 c0           TEST       EAX,EAX
00401253 74 13           JZ         LAB_00401268
00401255 48 8d 3d        LEA        RDI,[s_Congratulations!_That_was_
         ac 0d 00 00
0040125c b8 00 00        MOV        EAX,0x0
         00 00
```

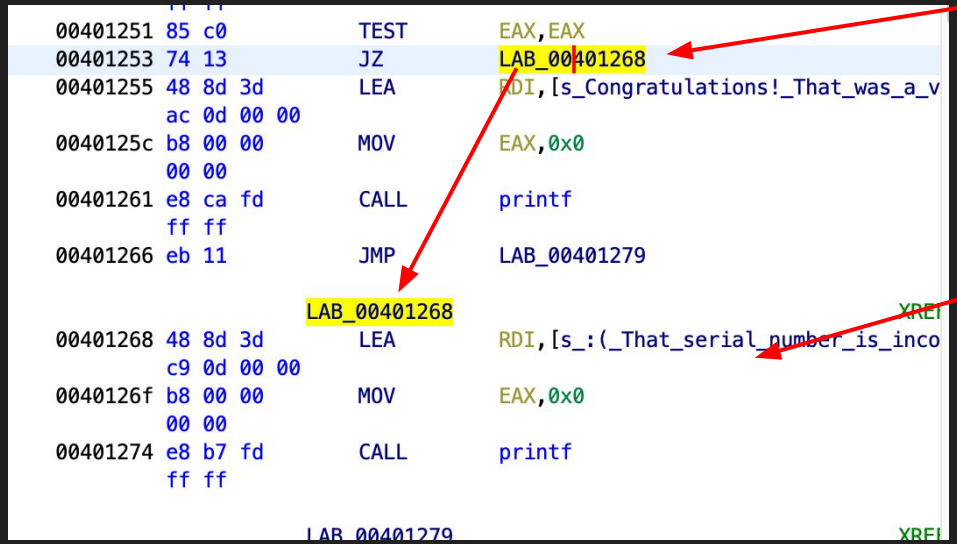
The result of a function call in assembly is stored in the RAX register

EAX (32-bits wide) is the lower half of the RAX register (64-bits wide)

TEST EAX, EAX checks whether EAX is 0
If it is, the zero-flag (ZF) is set

JZ means “jump if the zero-flag is set”

To jump, or not to jump?

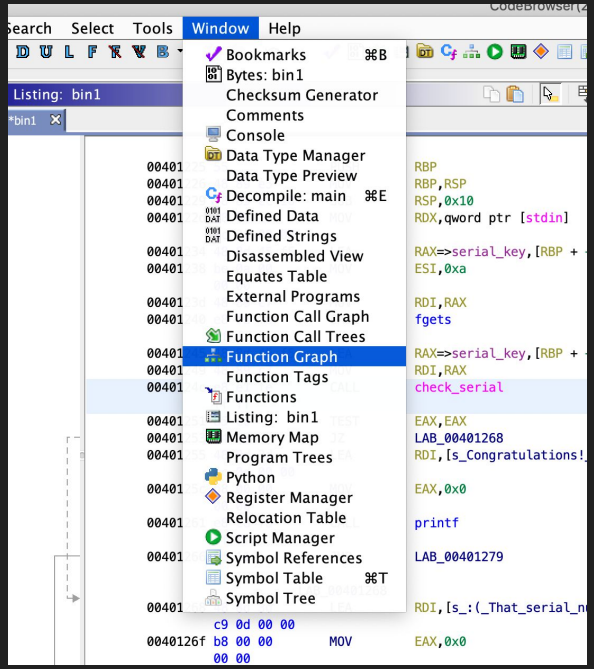


Well, if we *do* jump, then we're going to end up going to *0x401268*

And that brings us to here, which says our serial number is incorrect :(

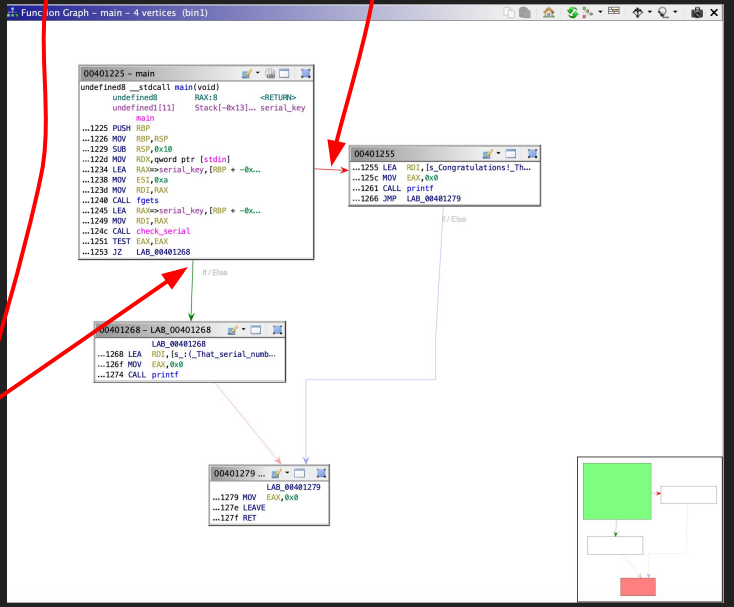
So **nope**, we do NOT want to jump. We wanna skip right over this jump

Side Note: Function Graphs

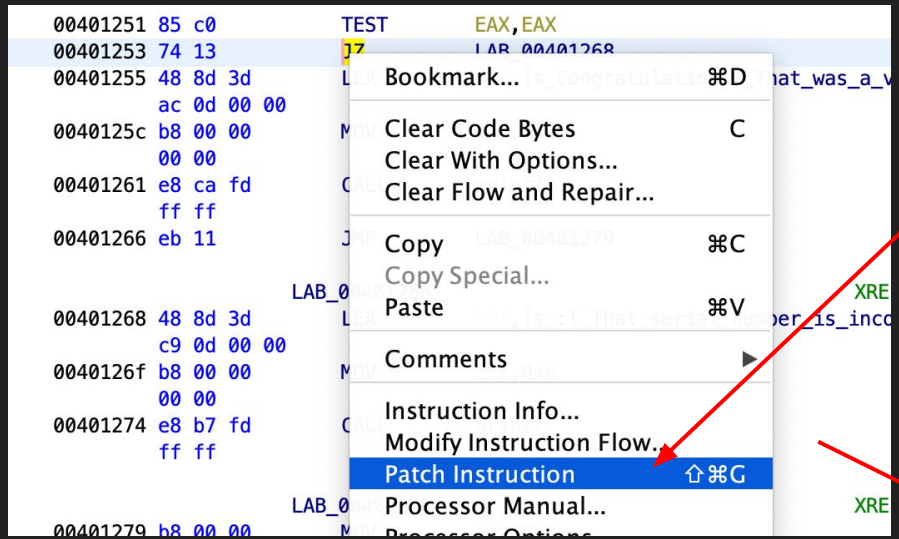


Function Graphs allow you to see at how different blocks of code in functions relate when jumping.

Here you can see that on success, JZ jumps to the ":(That Serial numb…" printf, but on failure, it jumps to the "Congratulations!" printf

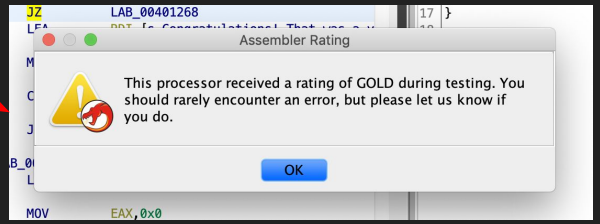


5. Shall we jump? NOP

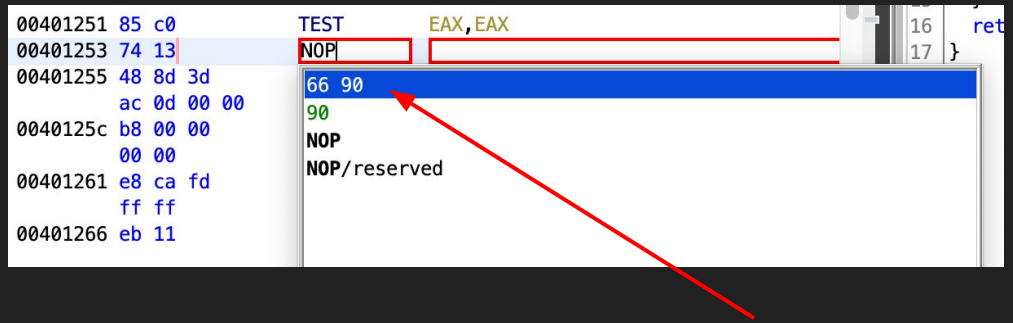


If we right click on the instruction we want to change, we can alter it using “Patch Instruction”

You may get a popup that tells you how great the assembler is for this architecture. Go ahead and click “OK”



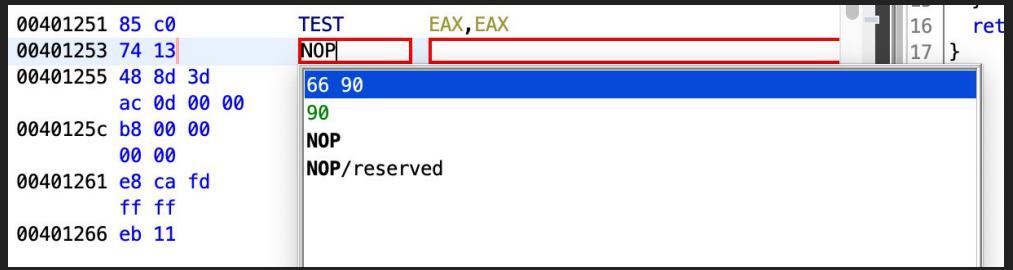
5. Shall we jump? NOP



We can now replace our JZ with a NOP
NOP == No-Operation == Do absolutely nothing

Note that normally, a NOP can be done with one byte (0x90), however we're using two bytes.

5. Shall we jump? NOP



The reason for this is that we can't insert instructions (Since it would change the offset of instructions like jmp), but we can edit the ones that are there. That means if we want to get rid of two bytes, we need to replace it with two bytes

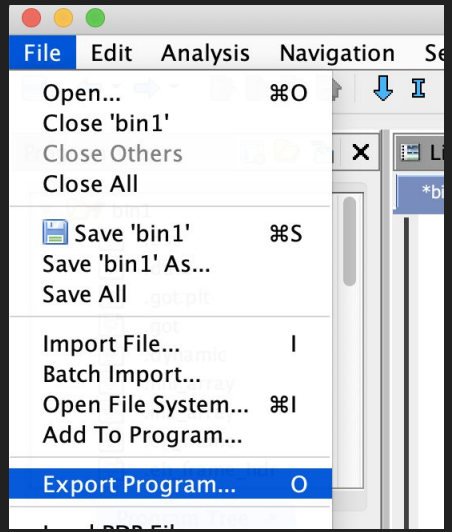
Luckily, Ghidra has given us some choices!

Nowhere to go but success!

```
1
2  undefined8 main(void)
3
4  {
5    char serial_key [11];
6
7    fgets(serial_key,10,stdin);
8    check_serial(serial_key);
9    printf("Congratulations! That was a valid serial number");
10   return 0;
11 }
12
```

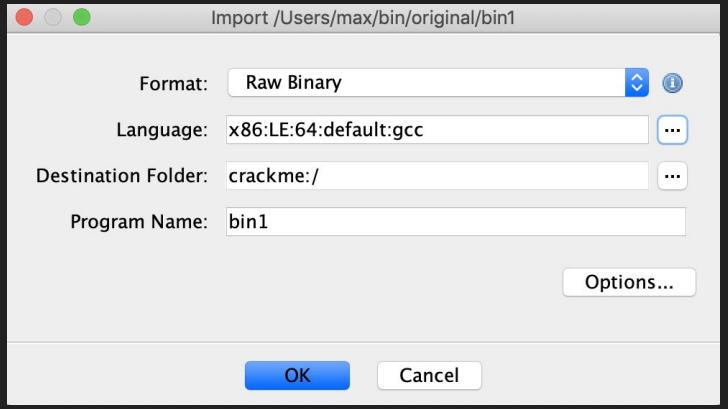
Notice that the decompiled output has now been updated.
The *check_serial* result is no longer checked.

how2export?



Ghidra has built-in exporting functionality, however it’s a liiiittle bit broken and only seems to export parts of the file, causing an immediate segfault

how2export?



One solution is to not import the file as an ELF but as a raw binary instead.

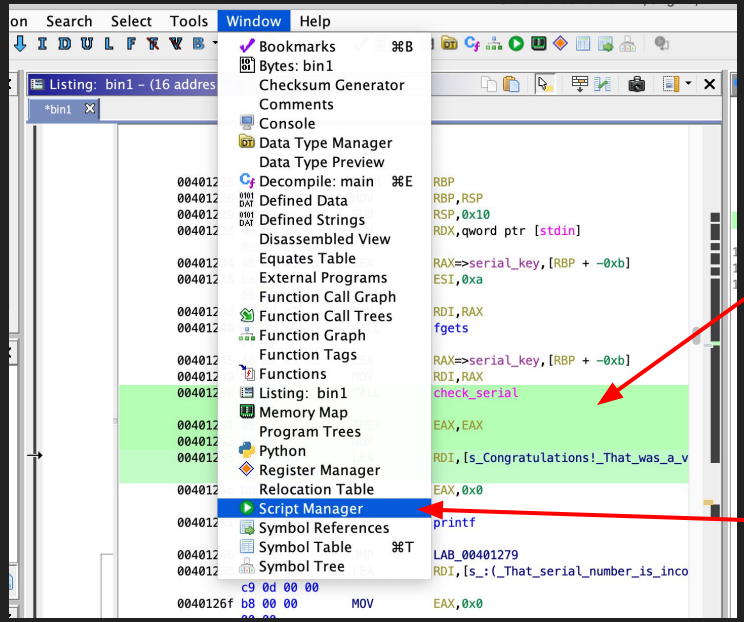
HOWEVER, that strips all the symbols from the binary, making it harder to reverse.

SavePatch

- https://github.com/schlafwandler/ghidra_SavePatch
- This script copies changed bytes and allows you to save the changed binary.
- A much better solution!

- To add the script, copy the Python file into Ghidra_public/Ghidra/Features/Python/ghidra_scripts

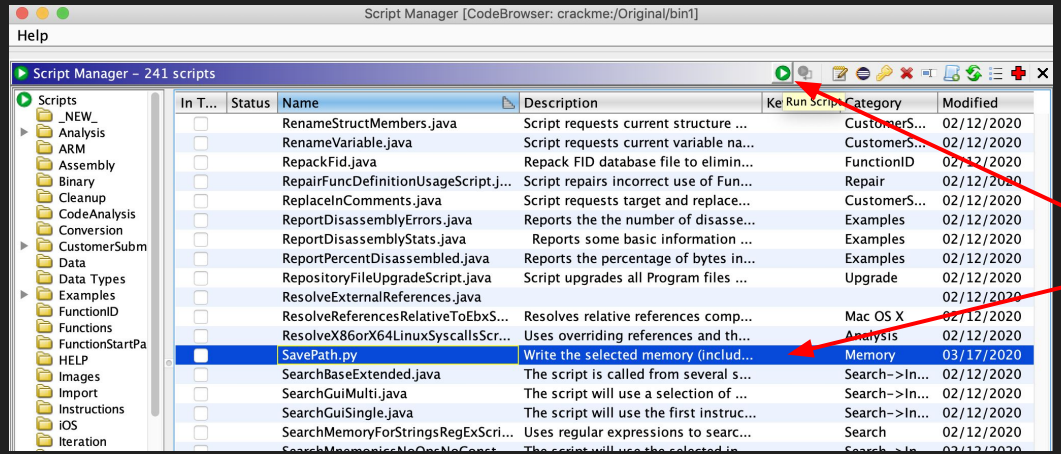
6. how2savepath



Highlight the changed byte area

Go to Window -> Script Manager

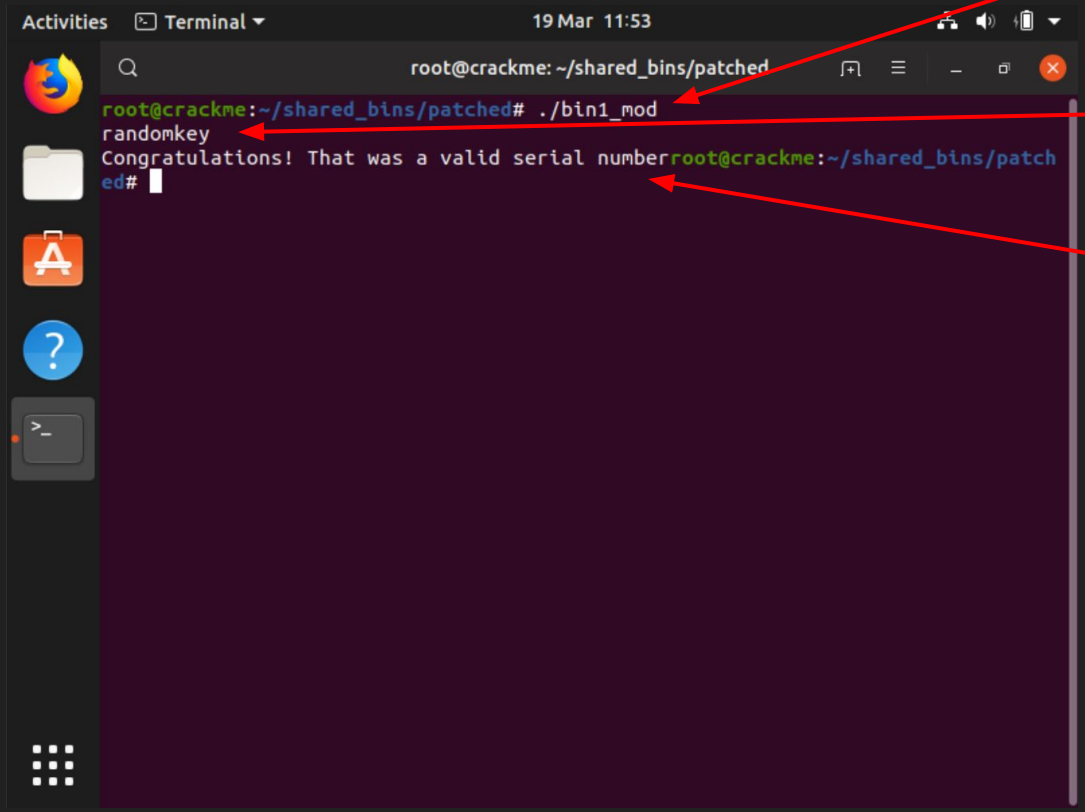
6. how2savepath



Click the script, click “Run Script”

Save the new binary in the “Patched” folder

7. winner="$(whoami)"



Run the modified binary on the testing VM

Provide any random key

Insert evil laugh here

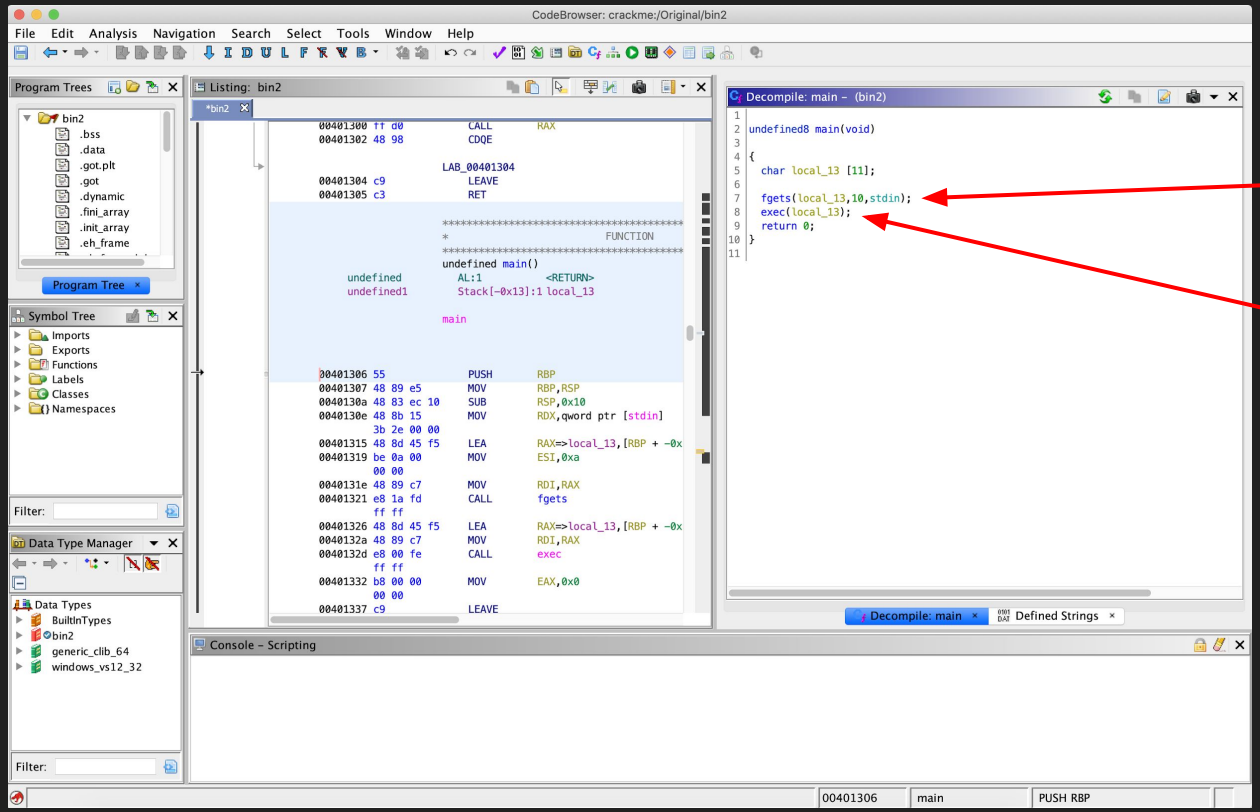
Recap

1. Loaded the binary
2. Edited the flow of the program
3. Saved the modified binary
4. Ran it!

bin2

From the top!

1. Locate the main() function



The main function looks different this time

It calls fgets, like before

It uses the serial key in a function called exec

2. View the exec() function

```
1
2  long exec(undefined8 param_1)
3
4  {
5    undefined8 uVar1;
6    int iVar2;
7    undefined8 *puVar3;
8    long lVar4;
9
10   puVar3 = (undefined8 *)mmap((void *)0x0,0xec,7,0x22,-1,0);
11   uVar1 = shellc0de._8_8_;
12   if (puVar3 == (undefined8 *)0xffffffffffffffff) {
13     lVar4 = 0;
14   }
15   else {
16     *puVar3 = shellc0de._0_8_;
17     puVar3[1] = uVar1;
18     uVar1 = shellc0de._24_8_;
19     puVar3[2] = shellc0de._16_8_;
20     puVar3[3] = uVar1;
21     uVar1 = shellc0de._40_8_;
22     puVar3[4] = shellc0de._32_8_;
23     puVar3[5] = uVar1;
24     uVar1 = shellc0de._56_8_;
25     puVar3[6] = shellc0de._48_8_;
26     puVar3[7] = uVar1;
27     uVar1 = shellc0de._72_8_;
28     puVar3[8] = shellc0de._64_8_;
29     puVar3[9] = uVar1;
30     uVar1 = shellc0de._88_8_;
31     puVar3[10] = shellc0de._80_8_;
32     puVar3[0xb] = uVar1;
33     uVar1 = shellc0de._104_8_;
34     puVar3[0xc] = shellc0de._96_8_;
35     puVar3[0xd] = uVar1;
36     uVar1 = shellc0de._120_8_;
37     puVar3[0xe] = shellc0de._112_8_;
38     puVar3[0xf] = uVar1;
39     uVar1 = shellc0de._136_8_;
40     puVar3[0x10] = shellc0de._128_8_;
41     puVar3[0x11] = uVar1;
42     uVar1 = shellc0de._152_8_;
43     puVar3[0x12] = shellc0de._144_8_;
44     puVar3[0x13] = uVar1;
45     uVar1 = shellc0de._168_8_;
```

The function calls mmap()

The contents of the global variable *'shellc0de'* is copied into the mmap'd area

The mmap'd area is then called as a function with the *serial_key* parameter

```
46     puVar3[0x14] = shellc0de._160_8_;
47     puVar3[0x15] = uVar1;
48     uVar1 = shellc0de._184_8_;
49     puVar3[0x16] = shellc0de._176_8_;
50     puVar3[0x17] = uVar1;
51     uVar1 = shellc0de._200_8_;
52     puVar3[0x18] = shellc0de._192_8_;
53     puVar3[0x19] = uVar1;
54     uVar1 = shellc0de._216_8_;
55     puVar3[0x1a] = shellc0de._208_8_;
56     puVar3[0x1b] = uVar1;
57     puVar3[0x1c] = shellc0de._224_8_;
58     *(undefined4 *)(puVar3 + 0x1d) = shellc0de._232_4_;
59     iVar2 = (*(code *)puVar3)(param_1);
60     lVar4 = (long)iVar2;
61   }
62   return lVar4;
63 }
64
```

3. Rename variables to make the function clearer

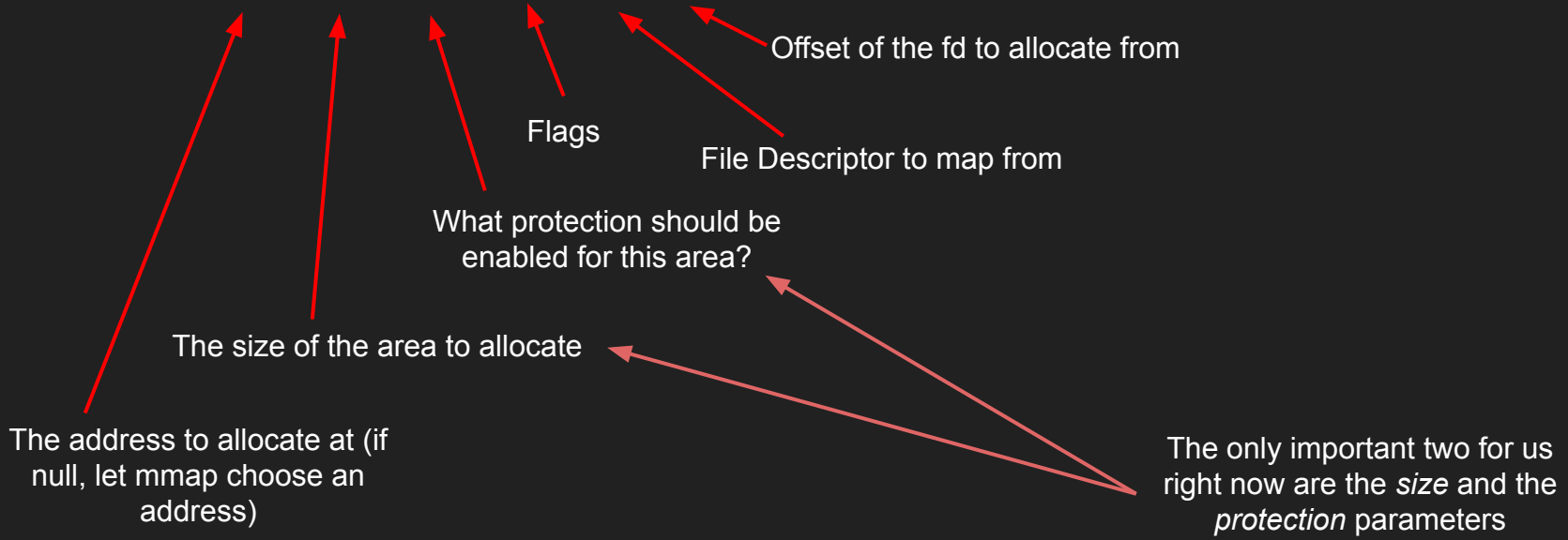
```
  function_pointer[0x1b] = uVar1;
  function_pointer[0x1c] = shellc0de._224_8_;
  *(undefined4 *)(function_pointer + 0x1d) = shellc0de._232_4_;
  iVar2 = (*(code *)function_pointer)(serial_key);
  lVar3 = (long)iVar2;
}
return lVar3;
```

Mmap? What’s that?

- Mmap is a system call that allows you to map files into memory or allocate areas of the process address space.

Okay, but what’re those parameters?

- mmap(0, 0xec, 7, 0x22, -1, 0)



mmap… protection 7?

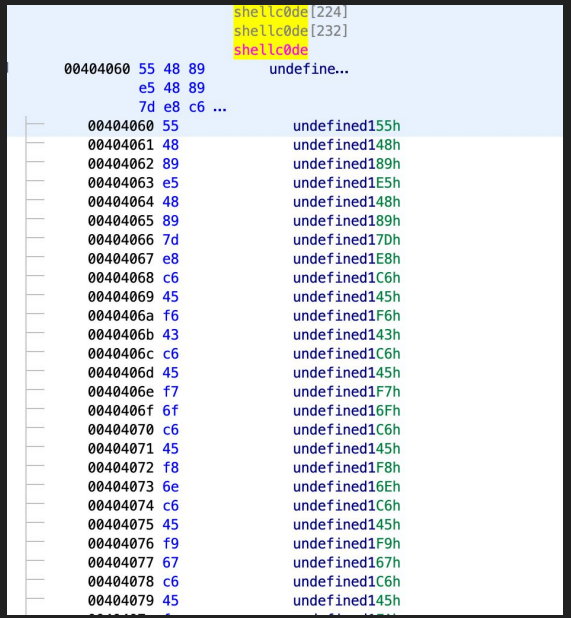
- Where does the number 7 derive from?
  - Linux header mman.h
  - 7 is the combination of PROT_READ (0x1), PROT_WRITE (0x2), and PROT_EXEC (0x4)

```
/*
 * sys/mman.h
 * mman-win32
 */

#ifndef _SYS_MMAN_H_
#define _SYS_MMAN_H_

#ifndef _WIN32_WINNT            // Allow use of features specific to Windows XP or later.
#define _WIN32_WINNT 0x0501     // Change this to the appropriate value to target other versions of Windows.
#endif

/* All the headers include this file. */
#ifndef _MSC_VER
#include <_mingw.h>
#endif

#include <sys/types.h>

#ifdef __cplusplus
extern "C" {
#endif

#define PROT_NONE       0
#define PROT_READ       1
#define PROT_WRITE      2
#define PROT_EXEC       4
```

So what does it mean overall?

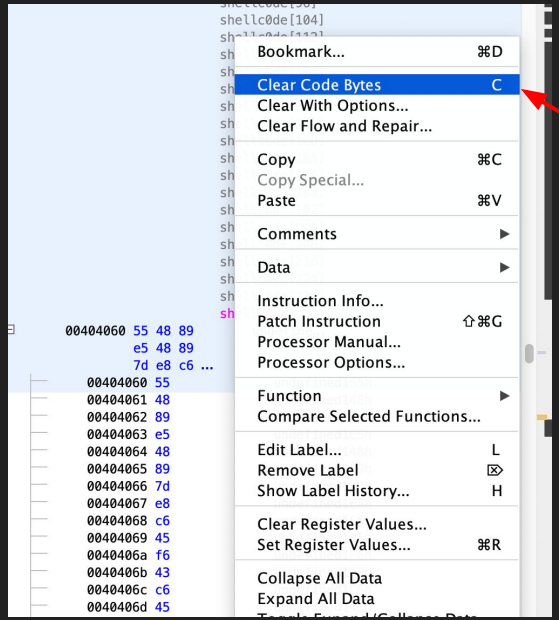
- The mmap function call here is allocating 0xec bytes of data that is readable, writeable, and executable (One super dangerous combination!)
- The contents of *shellc0de* (which we can now assume is 0xec bytes in size) is copied into this allocated area
- Finally, this is executed as a function

4. Double click on the *shellc0de* variable



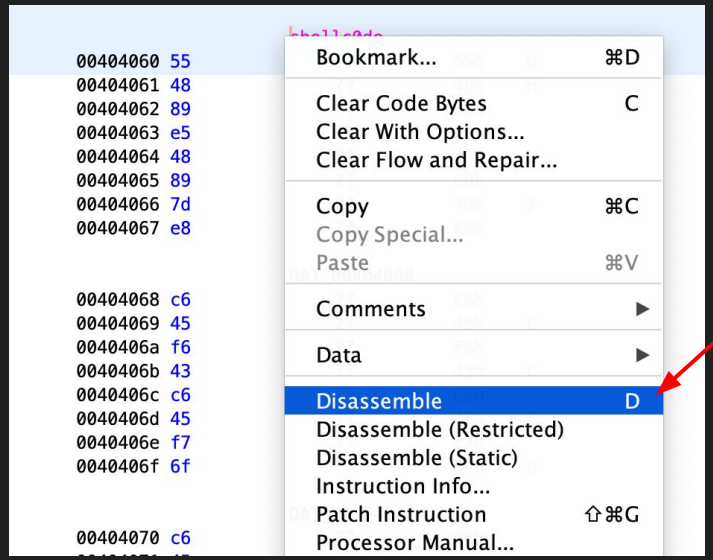
Since this will later be executed, we know that these bytes must be actual x86_64 instructions!

5. Clear code/data bytes



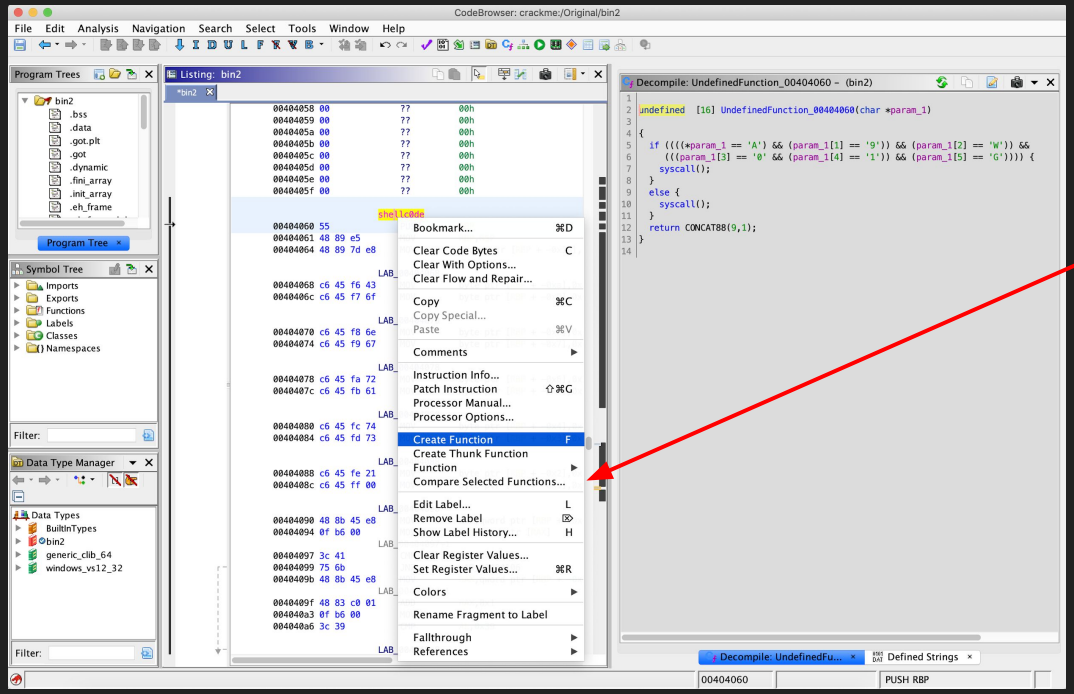
First we need to make these bytes undefined (neither code nor data). Right click *shellc0de* and click “Clear Code Bytes”)

6. Avengers, disassemble!



Right click *shellc0de* again and click on Disassemble

7. Convert this code to a function



Now we have instructions, we need to make this a defined function. Right click *shellc0de* again and click on “Create Function”

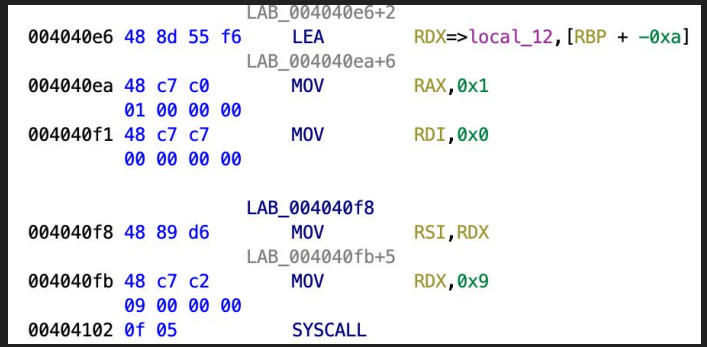
Syscall?

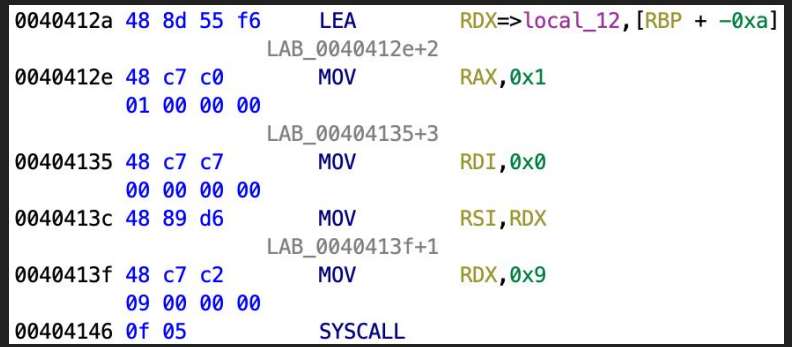
```
1
2  undefined  [16] shellc0de(char *param_1)
3
4  {
5    if ((((*param_1 == 'A') && (param_1[1] == '9')) && (param_1[2] == 'W')) &&
6       (((param_1[3] == '0' && (param_1[4] == '1')) && (param_1[5] == 'G')))) {
7      syscall();
8    }
9    else {
10     syscall();
11   }
12   return CONCAT88(9,1);
13 }
14
```

The decompiler has noted that there is a syscall here, but hasn't mentioned any arguments

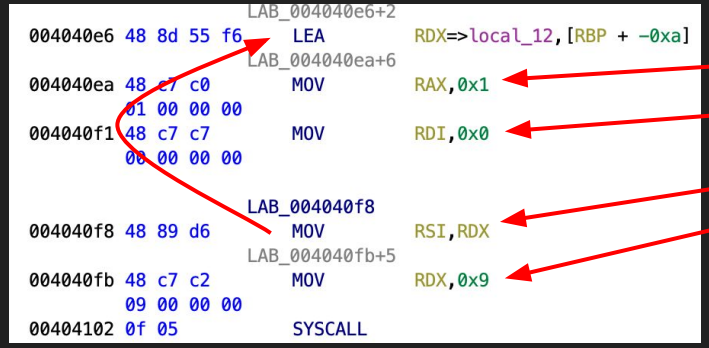
Syscall?

- RAX - System Call Number
- RDI, RSI, RDX, R10, R8, R9 - Arguments





Syscall?



RAX = 0x1 (sys_write)
RDI = 0x0 (stdin...?!)
RSI = *local_12*
RDX = 0x9 (*local_12* is 9 bytes long)

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

Writing… to stdin…?

- Yep! Linux allows this. The output of this will be printed on the screen but won’t be redirected to files.
  - E.g. *./write_to_stdin.sh* > *output* won’t cause stdin writes to be put in output

- Reverse Engineering is all about understanding weird and wonderful tricks!

Why use a syscall and not printf?

- The shellcode was compiled separately. If printf was called, it would try to resolve the function's address from a specific offset in a table located in memory (A table called the Global Offset Table. It holds a list of addresses of functions imported from libraries imported at runtime).

- So if my program uses "printf" and "fgets", I will have these two entries in my table (Offset 0 - printf, offset 1 - fgets), however if I create a second program with just "fgets" (Offset 0 - fgets), and try to run my first program as shellcode in it, the function wouldn't resolve properly (Since it's expecting Offset 0 to be printf, not fgets) and likely crash!

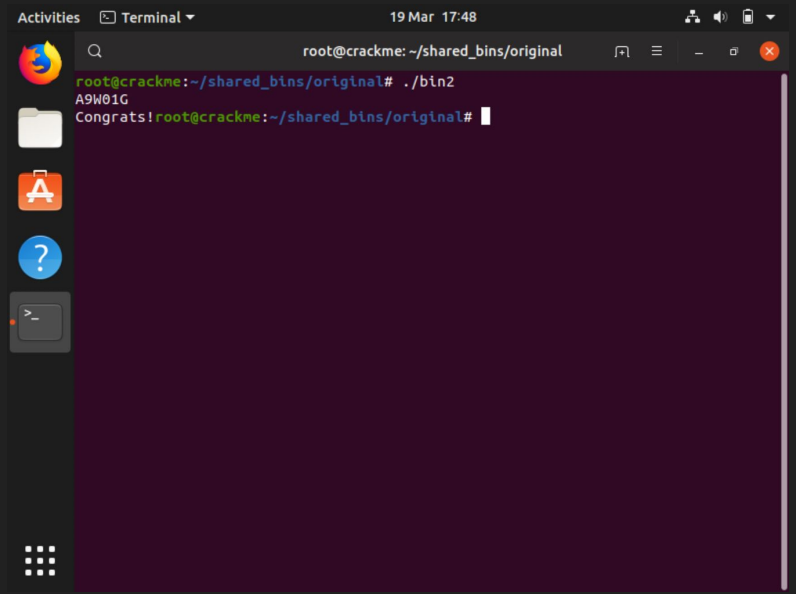
8. Make an assumption!

```
1
2  undefined  [16] shellc0de(char *param_1)
3
4  {
5    if ((((*param_1 == 'A') && (param_1[1] == '9')) && (param_1[2] == 'W')) &&
6       (((param_1[3] == '0' && (param_1[4] == '1')) && (param_1[5] == 'G')))) {
7      syscall();
8    }
9    else {
10     syscall();
11   }
12   return CONCAT88(9,1);
13 }
14
```

If this conditional ends up being true, then the first syscall is executed, otherwise the second one. We can make a guess that the first syscall is the “Success” and the second is the “Fail” because the first syscall requires more restrictions to be true

What now?

- We don’t even need to patch!
- The serial key is shown in the *shellc0de* disassembly!



Recap

1. Loaded the binary
2. Identified rwx memory allocation
3. Identified shellcode
4. Disassembled shellcode
5. Found the serial key!

bin3

Now it’s getting serious

1. Back to main()

```
1
2  undefined8 main(void)
3
4  {
5    char local_13 [11];
6
7    fgets(local_13,10,stdin);
8    exec(local_13);
9    return 0;
10 }
11
```

Ah, it’s our old friend exec()

2. Look at exec() for some Deja-Vu

```
long exec(undefined8 param_1)

{
  undefined8 uVar1;
  int iVar2;
  undefined8 *puVar3;
  long lVar4;
  uint local_c;

  puVar3 = (undefined8 *)mmap((void *)0x0,0xec,7,0x22,-1,0);
  if (puVar3 == (undefined8 *)0xffffffffffffffff) {
    lVar4 = 0;
  }
  else {
    local_c = 0;
    while (uVar1 = shellc0de._8_8_, local_c < 0xec) {
      shellc0de[(int)local_c] = shellc0de[(int)local_c] ^ 0xab;
      local_c = local_c + 1;
    }
    *puVar3 = shellc0de._0_8_;
    puVar3[1] = uVar1;
    uVar1 = shellc0de._24_8_;
    puVar3[2] = shellc0de._16_8_;
    puVar3[3] = uVar1;
    uVar1 = shellc0de._40_8_;
    puVar3[4] = shellc0de._32_8_;
    puVar3[5] = uVar1;
    uVar1 = shellc0de._56_8_;
    puVar3[6] = shellc0de._48_8_;
    puVar3[7] = uVar1;
    uVar1 = shellc0de._72_8_;
    puVar3[8] = shellc0de._64_8_;
    puVar3[9] = uVar1;
    uVar1 = shellc0de._88_8_;
    puVar3[10] = shellc0de._80_8_;
    puVar3[0xb] = uVar1;
    uVar1 = shellc0de._104_8_;
    puVar3[0xc] = shellc0de._96_8_;
    puVar3[0xd] = uVar1;
    uVar1 = shellc0de._120_8_;
    puVar3[0xe] = shellc0de._112_8_;
    puVar3[0xf] = uVar1;
    uVar1 = shellc0de._136_8_;
    puVar3[0x10] = shellc0de._128_8_;
```

mmap again

Shellc0de… again…

Function pointer...

```
    puVar3[0x10] = shellc0de._128_8_;
    puVar3[0x11] = uVar1;
    uVar1 = shellc0de._152_8_;
    puVar3[0x12] = shellc0de._144_8_;
    puVar3[0x13] = uVar1;
    uVar1 = shellc0de._168_8_;
    puVar3[0x14] = shellc0de._160_8_;
    puVar3[0x15] = uVar1;
    uVar1 = shellc0de._184_8_;
    puVar3[0x16] = shellc0de._176_8_;
    puVar3[0x17] = uVar1;
    uVar1 = shellc0de._200_8_;
    puVar3[0x18] = shellc0de._192_8_;
    puVar3[0x19] = uVar1;
    uVar1 = shellc0de._216_8_;
    puVar3[0x1a] = shellc0de._208_8_;
    puVar3[0x1b] = uVar1;
    puVar3[0x1c] = shellc0de._224_8_;
    *(undefined4 *)(puVar3 + 0x1d) = shellc0de._232_4_;
    iVar2 = (*(code *)puVar3)(param_1);
    lVar4 = (long)iVar2;
  }
  return lVar4;
}
```

BUT WAIT!

```
1
2  long exec(undefined8 param_1)
3
4  {
5    undefined8 uVar1;
6    int iVar2;
7    undefined8 *puVar3;
8    long lVar4;
9    uint local_c;
10
11   puVar3 = (undefined8 *)mmap((void *)0x0,0xec,7,0x22,-1,0);
12   if (puVar3 == (undefined8 *)0xffffffffffffffff) {
13     lVar4 = 0;
14   }
15   else {
16     local_c = 0;
17     while (uVar1 = shellc0de._8_8_, local_c < 0xec) {
18       shellc0de[(int)local_c] = shellc0de[(int)local_c] ^ 0xab;
19       local_c = local_c + 1;
20     }
21     *puVar3 = shellc0de._0_8_;
22     puVar3[1] = uVar1;
23     uVar1 = shellc0de._24_8_;
24     puVar3[2] = shellc0de._16_8_;
25     puVar3[3] = uVar1;
26     uVar1 = shellc0de._40_8_;
27     puVar3[4] = shellc0de._32_8_;
28     puVar3[5] = uVar1;
29     uVar1 = shellc0de._56_8_;
30     puVar3[6] = shellc0de._48_8_;
31     puVar3[7] = uVar1;
32     uVar1 = shellc0de._72_8_;
33     puVar3[8] = shellc0de._64_8_;
34     puVar3[9] = uVar1;
35     uVar1 = shellc0de._88_8_;
36     puVar3[10] = shellc0de._80_8_;
37     puVar3[0xb] = uVar1;
38     uVar1 = shellc0de._104_8_;
39     puVar3[0xc] = shellc0de._96_8_;
40     puVar3[0xd] = uVar1;
41     uVar1 = shellc0de._120_8_;
42     puVar3[0xe] = shellc0de._112_8_;
43     puVar3[0xf] = uVar1;
44     uVar1 = shellc0de._136_8_;
45     puVar3[0x10] = shellc0de._128_8_;
```

What is THIS?!

```
45     puVar3[0x10] = shellc0de._128_8_;
46     puVar3[0x11] = uVar1;
47     uVar1 = shellc0de._152_8_;
48     puVar3[0x12] = shellc0de._144_8_;
49     puVar3[0x13] = uVar1;
50     uVar1 = shellc0de._168_8_;
51     puVar3[0x14] = shellc0de._160_8_;
52     puVar3[0x15] = uVar1;
53     uVar1 = shellc0de._184_8_;
54     puVar3[0x16] = shellc0de._176_8_;
55     puVar3[0x17] = uVar1;
56     uVar1 = shellc0de._200_8_;
57     puVar3[0x18] = shellc0de._192_8_;
58     puVar3[0x19] = uVar1;
59     uVar1 = shellc0de._216_8_;
60     puVar3[0x1a] = shellc0de._208_8_;
61     puVar3[0x1b] = uVar1;
62     puVar3[0x1c] = shellc0de._224_8_;
63     *(undefined4 *)(puVar3 + 0x1d) = shellc0de._232_4_;
64     iVar2 = (*(code *)puVar3)(param_1);
65     lVar4 = (long)iVar2;
66   }
67   return lVar4;
68 }
```

3. Figure out what it’s doing

local_c goes to 0xec

```
local_c = 0;
while (uVar1 = shellc0de._8_8_, local_c < 0xec) {
  shellc0de[(int)local_c] = shellc0de[(int)local_c] ^ 0xab;
  local_c = local_c + 1;
}
```

Each byte of *shellc0de* is replaced with (itself ^ 0xab)

local_c is iterating

What is (itself ^ 0xab) ?

- ^ means XOR (Exclusive-Or)
- It’s a bitwise operator that sets a bit only if ONE of the two bits are set.

1 ^ 0 = 1

1 ^ 1 = 0

0 ^ 1 = 1

0 ^ 0 = 0

1101 ^ 1010 = 0111

Is the same as 0xd ^ 0xa = 0x7

XOR Properties

- XOR is great fun to play around with and it has some pretty neat properties that make it useful in encryption
- One-Time Pad Encryption: Plaintext ^ Key = Ciphertext
- One-Time Pad Decryption: Ciphertext ^ Key = Plaintext
- Figure out the key: Ciphertext ^ Plaintext = Key

Shellc0de Decryption

- *Shellc0de* must already be encrypted with 0xab
- This loop decrypts it before copying it into the rwx memory area

Make an exec()-utive decision

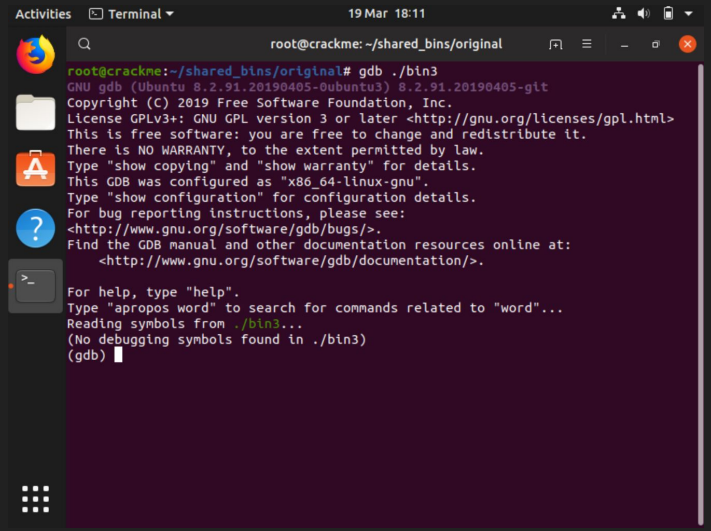
Two options:

- Decrypt it with Python and then disassemble it
- Let the program decrypt it for us

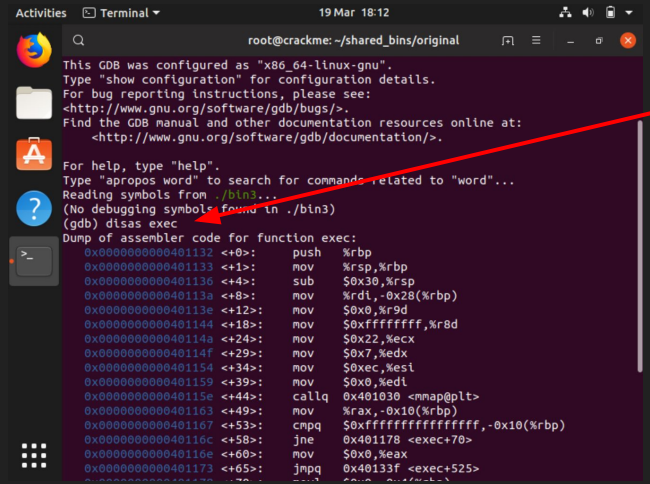
Let the program decrypt it for us

- Enter: GDB, the GNU Debugger
- Allows us to breakpoint on instructions, read memory, see registers, and perform all sorts of magic
  - Break just as it’s about to call the function pointer and see what the decrypted assembly looks like
- Really easy to use once you get the basics down

4. Open GDB on bin3

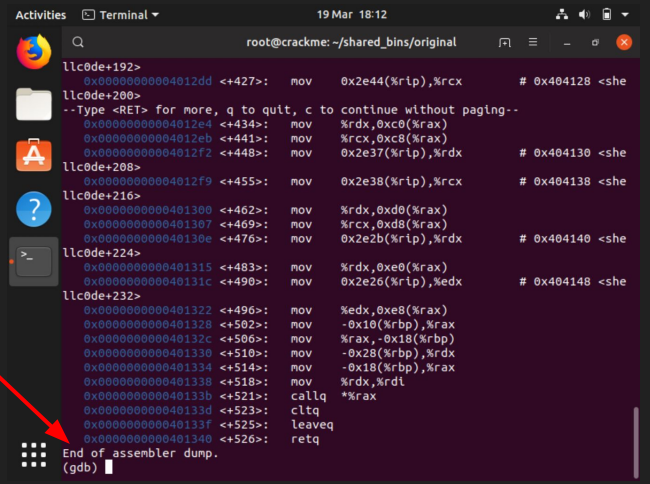


5. Disassemble the exec function

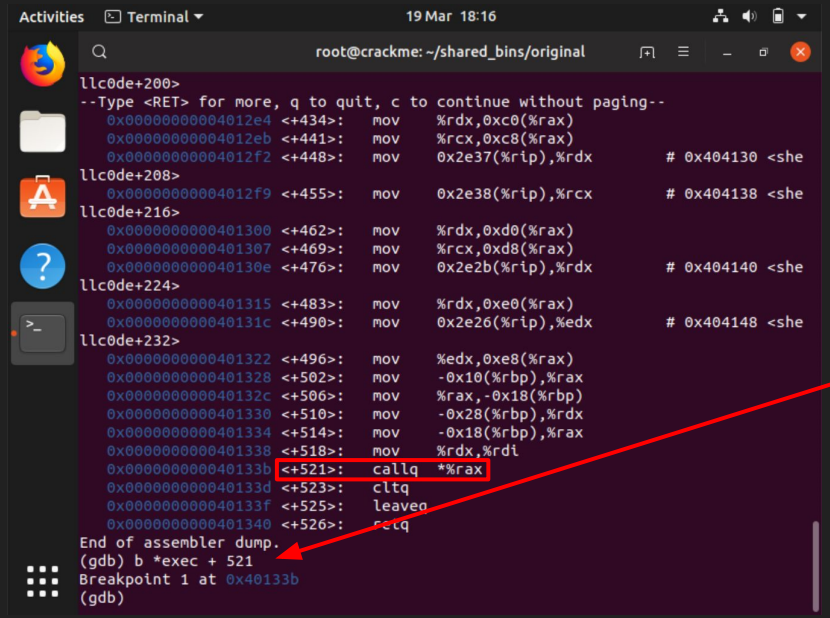


Type *disas exec* to disassemble the function

Press enter until you read the end of the function



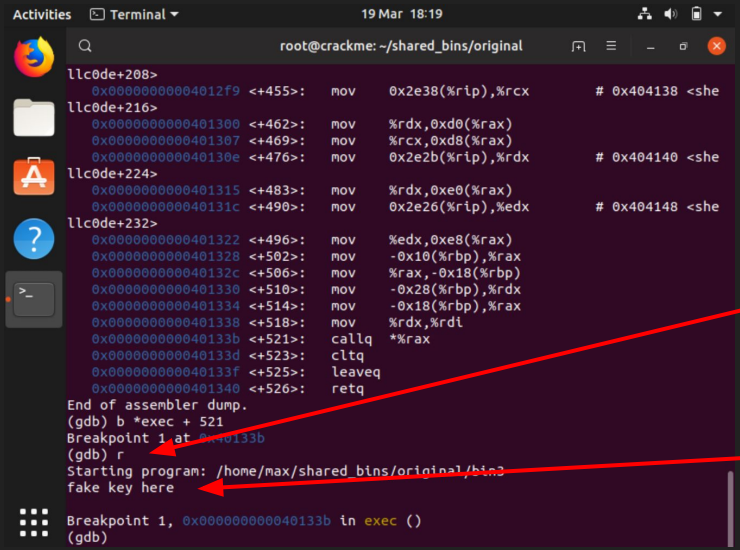
6. Break on the call



Type *b *exec + 521* to set a breakpoint.

Note that 521 is the offset from the start of *exec* that has the instruction *callq *%rax* (This calls the address stored in the *RAX* register)

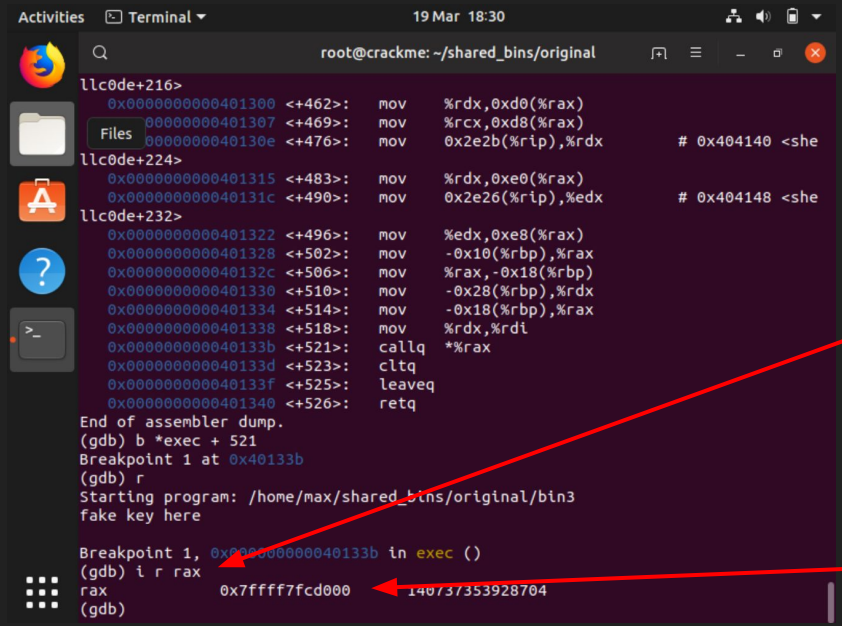
7. Run the program



Type *r* to run the progam

Add a fake key here

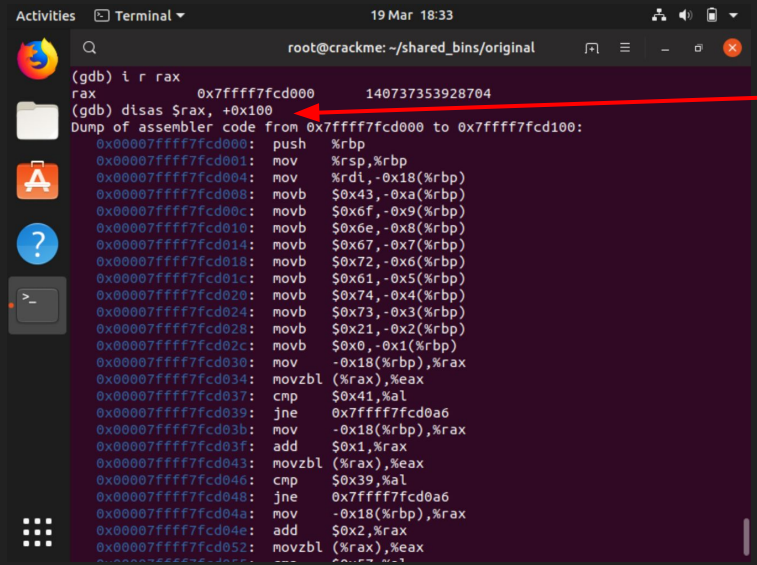
8. Get the address of the mmap'd area



Type *i r rax* to get the *info* about *register rax* once we hit the breakpoint

Here's our mmap'd location we're about to jump to!

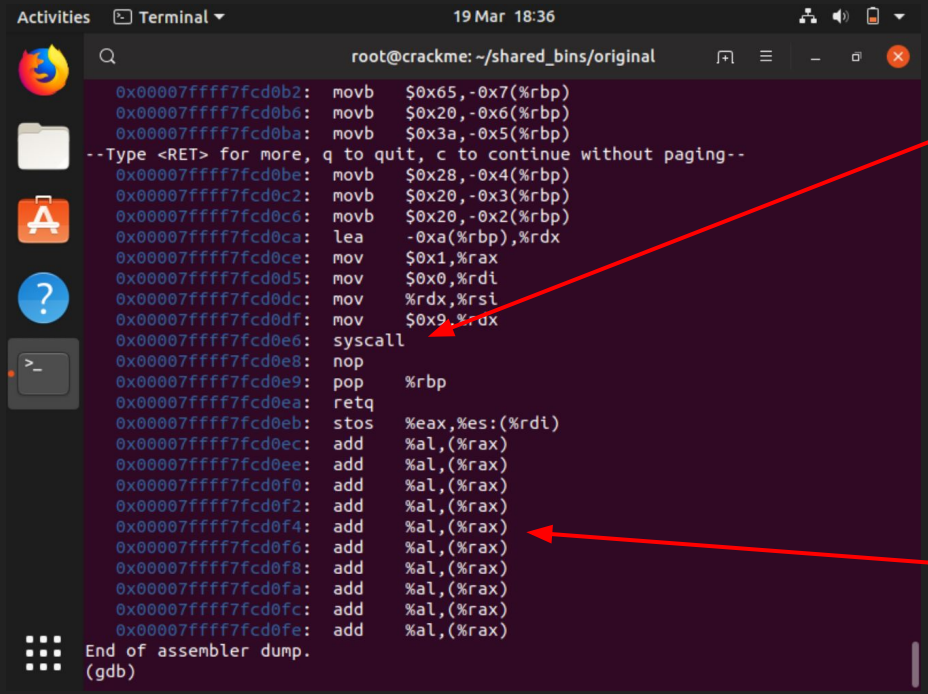
9. Disassemble the mmap’d area



Type *disas $rax, +0x100* to disassemble starting the address in RAX for 0x100 bytes

In this case, we need the *+0x100* since the address in RAX isn’t defined as a function in GDB. For all GDB knows, this could just be random bytes of data. Therefore we explicitly tell it how much to disassemble.

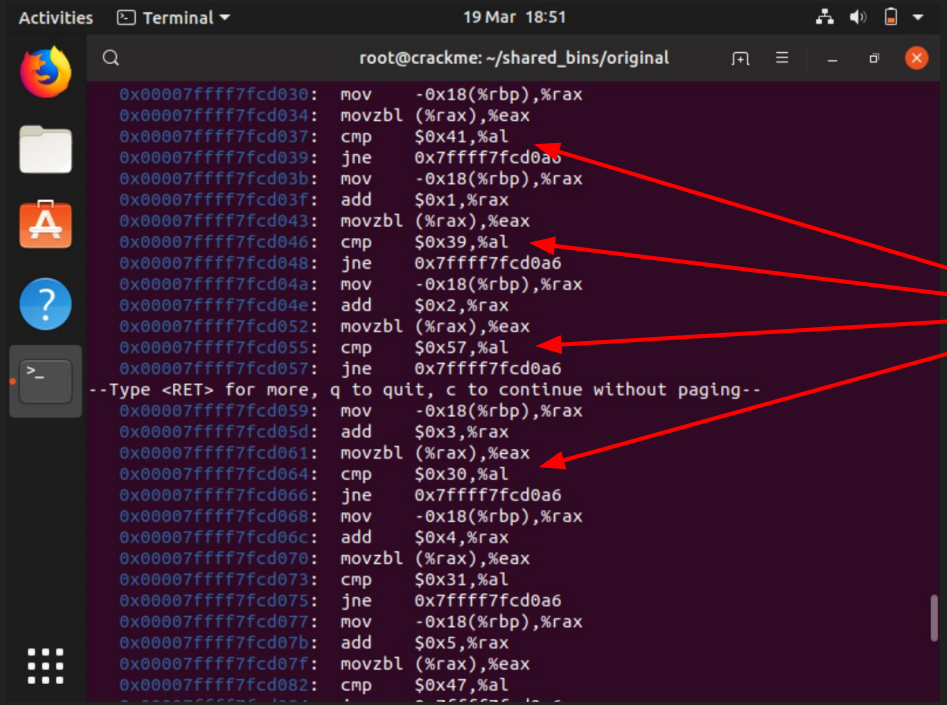
10. Look around



There are two syscalls here, same as before

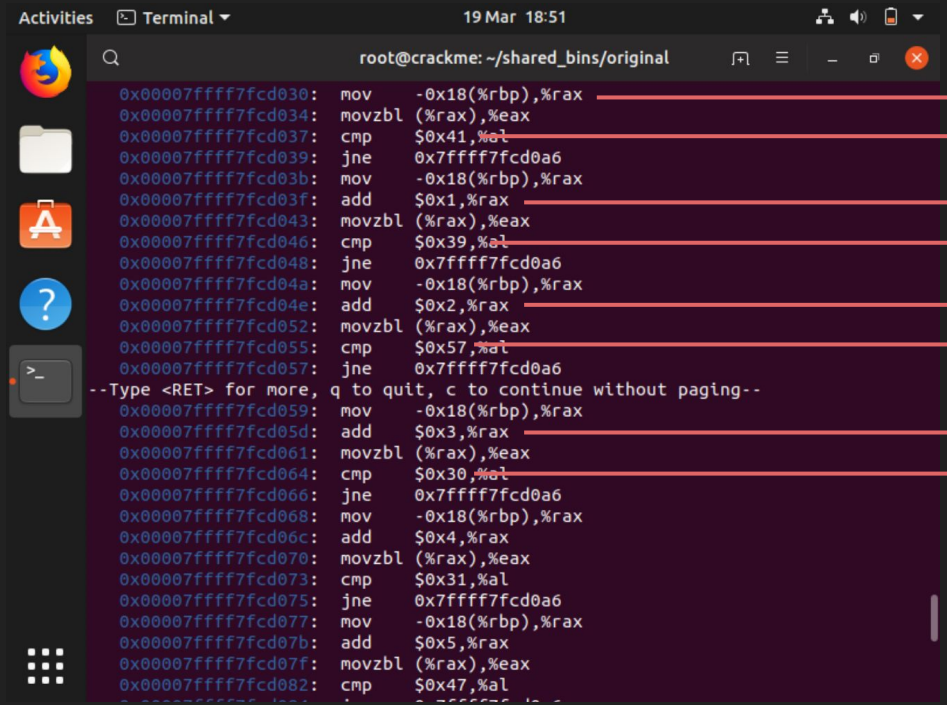
If you see *add %al,(%rax)* then you've reached an area of null-bytes since these are the opcodes and operands for this instruction.

10. Look around



The cmp instruction compares two things and sets a bunch of the flags bits accordingly

10. Look around



Get serial[0]

Compare with A

Get serial[0 + 1]

Compare with 9

Get serial[0 + 2]

Compare with W

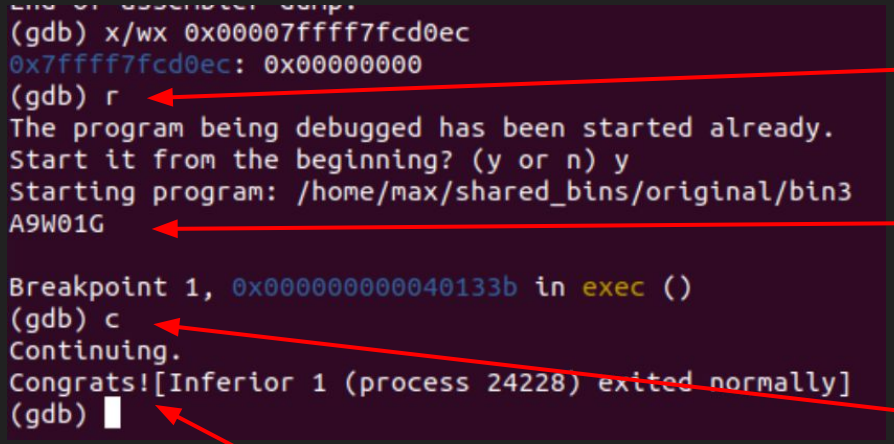
Get serial[0 + 3]

Compare with 0

```
Max-MBP:~ max$ python
Python 2.7.10 (default, Feb 22 2019, 21:55:15)
[GCC 4.2.1 Compatible Apple LLVM 10.0.1 (clang-1001.0.37.14)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> chr(0x41)
'A'
>>> chr(0x39)
'9'
>>> chr(0x57)
'W'
>>> chr(0x30)
'0'
>>>
```

You can convert numbers into characters easily with Python!

11. Enter the serial



Run from the beginning

Enter the serial key (Funnily enough it's the same key as the last challenge)

Continue on the breakpoint

Party, because you completed the challenge!

Wait wait wait… No patching AGAIN?!

- I know I know, I’m the worst!
- But sometimes there’s just no need
- If you can recover the key, or find an easy way to generate keys, that’s a much neater solution! After all, keygens are equally a part of the software cracking world

Recap

1. Loaded the binary
2. Identified rwx memory allocation
3. Identified shellcode
4. Let the program decrypt the shellcode
5. Disassembled in-memory shellcode
6. Identified the key from assembly instructions!

bin4

Revenge of the developer

1. The main() course

```
1
2  undefined8 main(void)
3
4  {
5    char local_13 [11];
6
7    fgets(local_13,10,stdin);
8    exec(local_13);
9    return 0;
10 }
11
```

Your friendly neighbourhood exec()

2. Same-same but different!

```
1
2  long exec(undefined8 param_1)
3
4  {
5    undefined8 uVar1;
6    int iVar2;
7    undefined8 *puVar3;
8    long lVar4;
9    uint local_c;
10
11   puVar3 = (undefined8 *)mmap((void *)0x0,0xec,7,0x22,-1,0);
12   if (puVar3 == (undefined8 *)0xffffffffffffffff) {
13     lVar4 = 0;
14   }
15   else {
16     local_c = 0;
17     while (uVar1 = shellc0de._8_8_, local_c < 0xec) {
18       shellc0de[(int)local_c] =
19            shellc0de[(int)local_c] ^ *(byte *)((long)&key + (ulong)(local_c & 7));
20       local_c = local_c + 1;
21     }
22     *puVar3 = shellc0de._0_8_;
23     puVar3[1] = uVar1;
24     uVar1 = shellc0de._24_8_;
25     puVar3[2] = shellc0de._16_8_;
26     puVar3[3] = uVar1;
27     uVar1 = shellc0de._40_8_;
28     puVar3[4] = shellc0de._32_8_;
29     puVar3[5] = uVar1;
```

shellc0de is XOR'd with something else

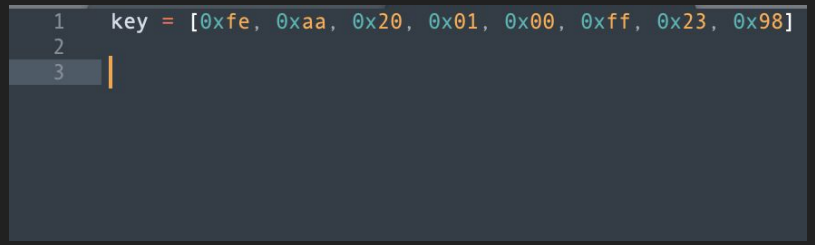
*(byte *)((long)&key + (ulong)(local_c & 7))

- *local_c & 7*?
  - This is another way or writing modulus (the % operator)
  - 7 in binary is 111 - (number & 7) discards bits from the 4th bit onwards
  - Therefore (number & 7) gives the remainder. It's the same as (number % 8)
- *Key* is shown in light blue in the decompiler, the same as *shellc0de*.
  - This means that it's a global variable, just like *shellc0de*.
- The block of code in the title reduces down to *key[local_c % 8]*
  - This allows us to deduce that *key* is 8 bytes long since it would only iterate over the number of characters in key

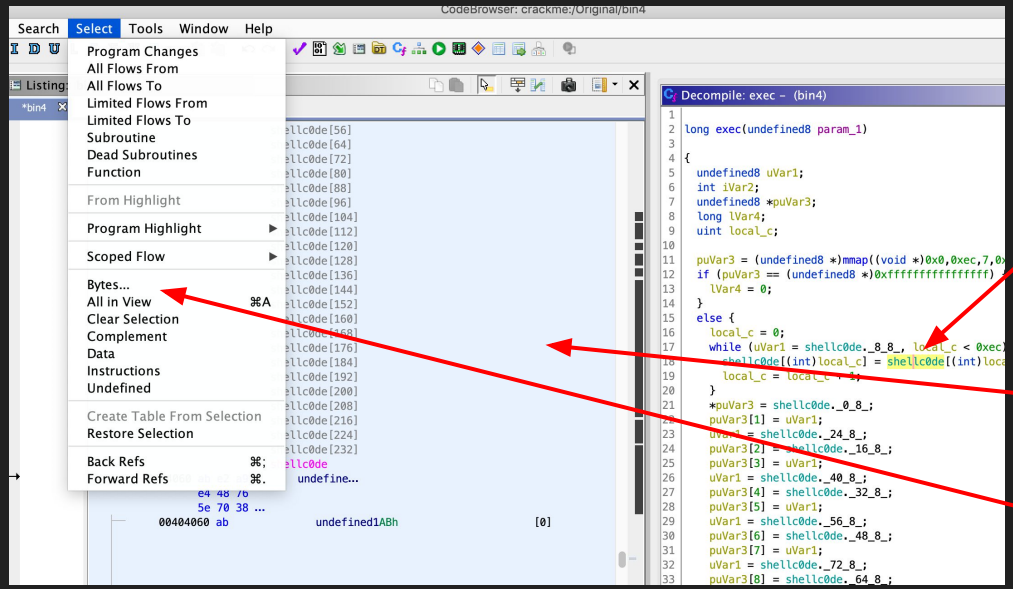
3. Find the *key*

Double-click on *key*

4. Start a Python script with the *key*



5. Select the *shellc0de* bytes

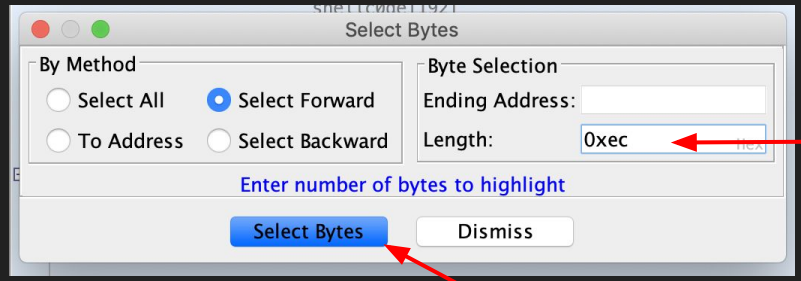


Double-click on *shellc0de*

Select it in the disassembly window

Go to Select -> Bytes...

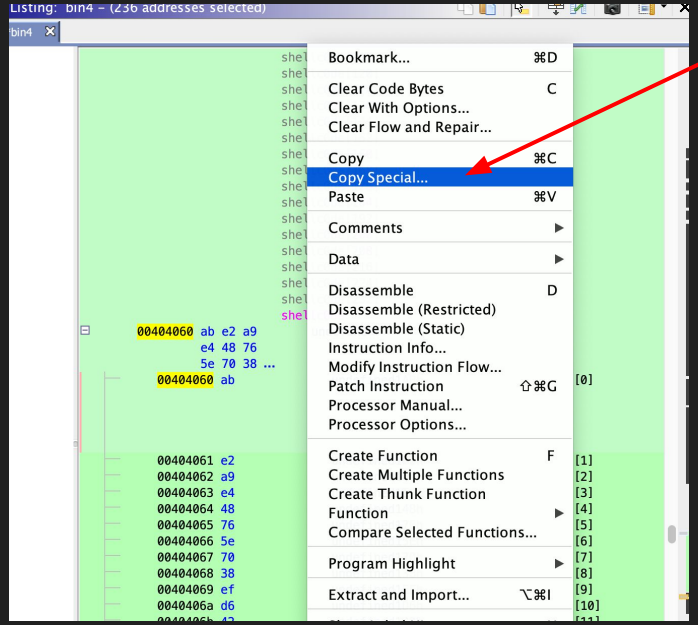
5. Select the *shellc0de* bytes



Set the Length to 0xec

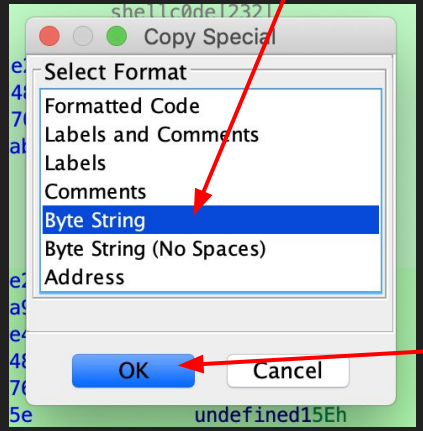
Click “Select Bytes”

5. Select the *shellc0de* bytes



Right-click the selected area and click on “Copy Special…”

Select “Byte String”



Click OK

6. Turn the copied bytes into a Python string

- In your clipboard you will now have a bunch of copied bytes:

```
ab e2 a9 e4 48 76 5e 70 38 ef d6 42 c6 ba d4 f7 38 ef
d8 6f c6 ba da ff 38 ef da 73 c6 ba d8 f9 38 ef dc 75
c6 ba de eb 38 ef de 20 c6 ba dc 98 b6 21 65 e9 0f 49
23 a4 bf df 4b 49 8b ba cb d0 7d 6a 21 0e b6 ff 1f a1
8b f6 68 8a 45 17 6b 1b 3e a8 2f b7 00 c3 74 ed b3 e2
ab 44 e8 b7 a0 58 fd a5 96 01 3c cf 56 a6 b6 21 65 e9
48 7c e3 9c f1 1c 20 3d 31 8a 0c d0 75 ef c8 49 83 3f
26 97 48 aa 1c 46 75 df 6b 15 ab 5c 68 c6 c0 fe 23 98
fe e2 e7 c6 00 ff 23 98 b6 23 f6 49 c7 3d 2a 98 fe aa
2f 04 eb bd e5 dd 08 e4 e6 44 f7 90 e5 dd 06 da e6 44
f9 9a e5 dd 04 8a e6 44 fb c5 e5 dd 02 82 e6 44 fd df
e5 dd 00 8a 68 8c 55 09 6b 5f 3e ab 20 01 00 b7 e4 5f
fe aa 20 01 48 76 f5 d0 39 68 29 01 00 ff 2c 9d 6e f7
e3 00
```

- Replace the spaces with \x and add a \x to the start of the first byte:

```
\xab\xe2\xa9\xe4\x48\x76\x5e\x70\x38\xef\xd6\x42\xc6\xba
\xd4\xf7\x38\xef\xd8\x6f\xc6\xba\xda\xff\x38\xef\xda\x73
\xc6\xba\xd8\xf9\x38\xef\xdc\x75\xc6\xba\xde\xeb\x38\xef
\xde\x20\xc6\xba\xdc\x98\xb6\x21\x65\xe9\x0f\x49\x23\xa4
\xbf\xdf\x4b\x49\x8b\xba\xcb\xd0\x7d\x6a\x21\x0e\xb6\xff
\x1f\xa1\x8b\xf6\x68\x8a\x45\x17\x6b\x1b\x3e\xa8\x2f\xb7
\x00\xc3\x74\xed\xb3\xe2\xab\x44\xe8\xb7\xa0\x58\xfd\xa5
\x96\x01\x3c\xcf\x56\xa6\xb6\x21\x65\xe9\x48\x7c\xe3\x9c
\xf1\x1c\x20\x3d\x31\x8a\x0c\xd0\x75\xef\xc8\x49\x83\x3f
\x26\x97\x48\xaa\x1c\x46\x75\xdf\x6b\x15\xab\x5c\x68\xc6
\xc0\xfe\x23\x98\xfe\xe2\xe7\xc6\x00\xff\x23\x98\xb6\x23
\xf6\x49\xc7\x3d\x2a\x98\xfe\xaa\x2f\x04\xeb\xbd\xe5\xdd
\x08\xe4\xe6\x44\xf7\x90\xe5\xdd\x06\xda\xe6\x44\xf9\x9a
\xe5\xdd\x04\x8a\xe6\x44\xfb\xc5\xe5\xdd\x02\x82\xe6\x44
\xfd\xdf\xe5\xdd\x00\x8a\x68\x8c\x55\x09\x6b\x5f\x3e\xab
\x20\x01\x00\xb7\xe4\x5f\xfe\xaa\x20\x01\x48\x76\xf5\xd0
\x39\x68\x29\x01\x00\xff\x2c\x9d\x6e\xf7\xe3\x00
```

6. Turn the copied bytes into a Python string

- Copy this into your script as a string:

```
1   key = [0xfe, 0xaa, 0x20, 0x01, 0x00, 0xff, 0x23, 0x98]
2
3   data = "\xab\xe2\xa9\xe4\x48\x76\x5e\x70\x38\xef\xd6\x42\xc6\x
4
```

Tada! Python bytes in a string!

7. Write the decryption function

```python
key = [0xfe, 0xaa, 0x20, 0x01, 0x00, 0xff, 0x23, 0x98]

data = "\xab\xe2\xa9\xe4\x48\x76\x5e\x70\x38\xef\xd6\x42\xc6\xba\xd4\xf7\x38\xef\xd8\x6f\xc6\x

def decrypt(_data, _key):
    output = ""
    for i in range(len(_data)):
        output += chr(ord(_data[i]) ^ _key[i % 8])
    return output

decrypt(data, key)
```

Variable to store the result

Iterate through data size

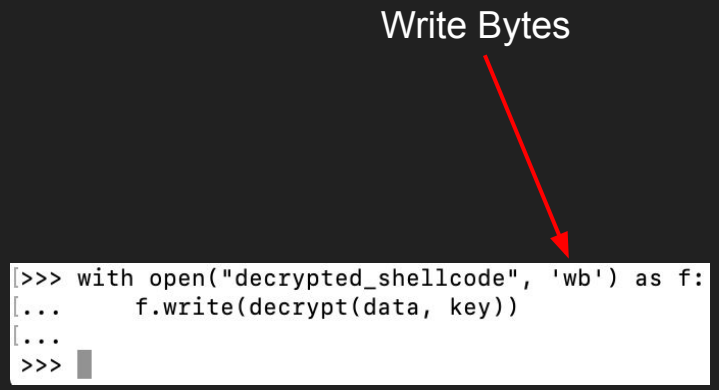
XOR data[i] with key[i % 8]

ord() converts a byte to a number

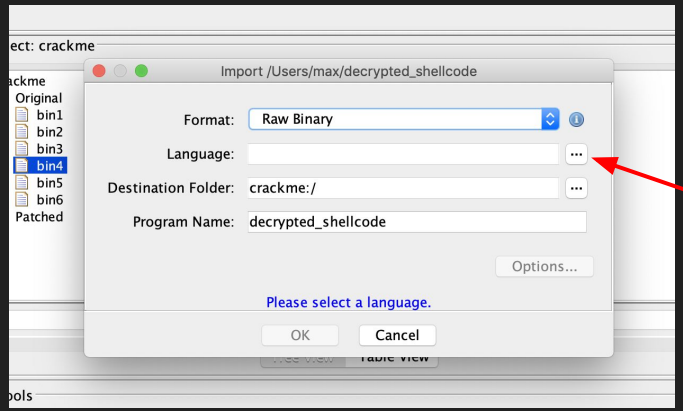
chr() converts a number to a byte

```
>>> decrypt(data, key)
'UH\x89\xe5H\x89}\xe8\xc6E\xf6C\xc6E\xf7o\xc6E\xf8n\xc6E\xf9g\xc6E\xfar\xc6E\xfb
a\xc6E\xfct\xc6E\xfds\xc6E\xfe!\xc6E\xff\x00H\x8bE\xe8\x0f\xb6\x00<AukH\x8bE\xe8
H\x83\xc0\x01\x0f\xb6\x00<9u\\H\x8bE\xe8H\x83\xc0\x02\x0f\xb6\x00<WuMH\x8bE\xe8H
\x83\xc0\x03\x0f\xb6\x00<0u>H\x8bE\xe8H\x83\xc0\x04\x0f\xb6\x00<1u/H\x8bE\xe8H\x
83\xc0\x05\x0f\xb6\x00<Gu H\x8dU\xf6H\xc7\xc0\x01\x00\x00\x00H\xc7\xc7\x00\x00\x
00\x00H\x89\xd6H\xc7\xc2\t\x00\x00\x00\x0f\x05\xebB\xc6E\xf6N\xc6E\xf7o\xc6E\xf8
p\xc6E\xf9e\xc6E\xfa \xc6E\xfb:\xc6E\xfc(\xc6E\xfd \xc6E\xfe H\x8dU\xf6H\xc7\xc0
\x01\x00\x00\x00H\xc7\xc7\x00\x00\x00\x00H\x89\xd6H\xc7\xc2\t\x00\x00\x00\x0f\x0
5\x90]\xc3\x01'
>>>
```

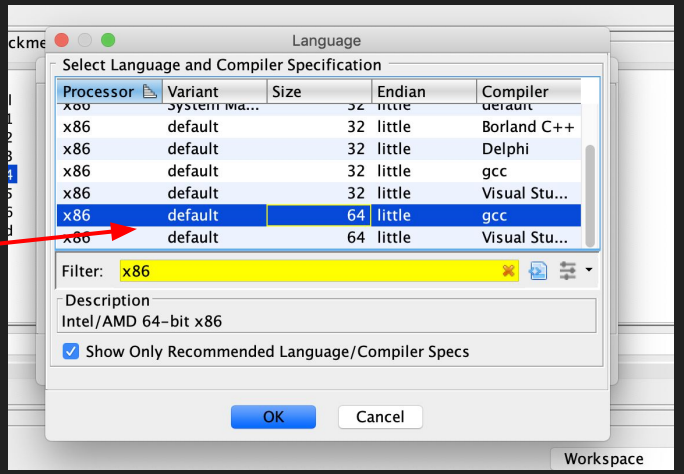
8. Write the decrypted data to a file



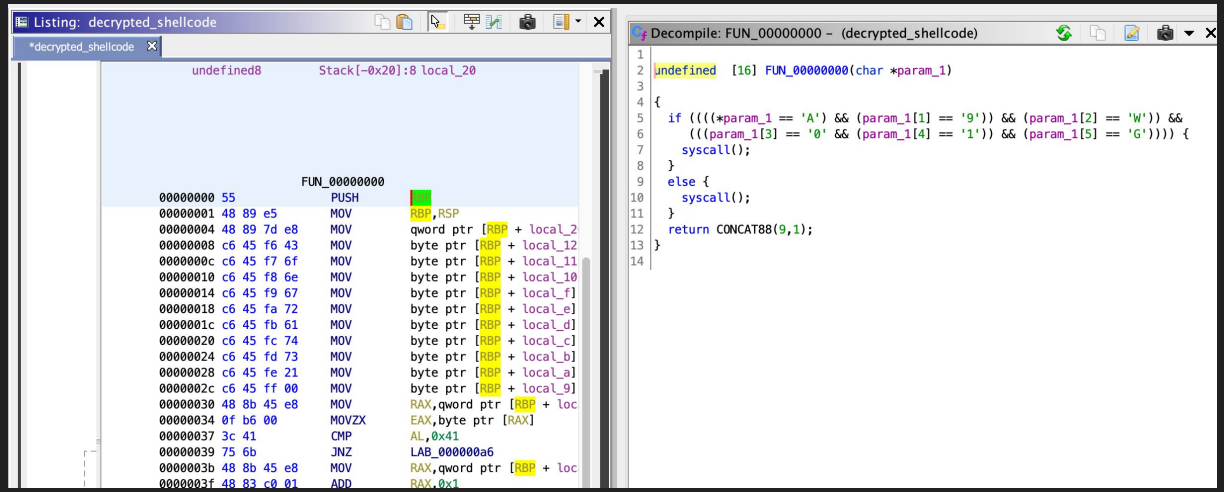
9. Drop the decrypted shellcode into Ghidra



Select the Language as x86_64

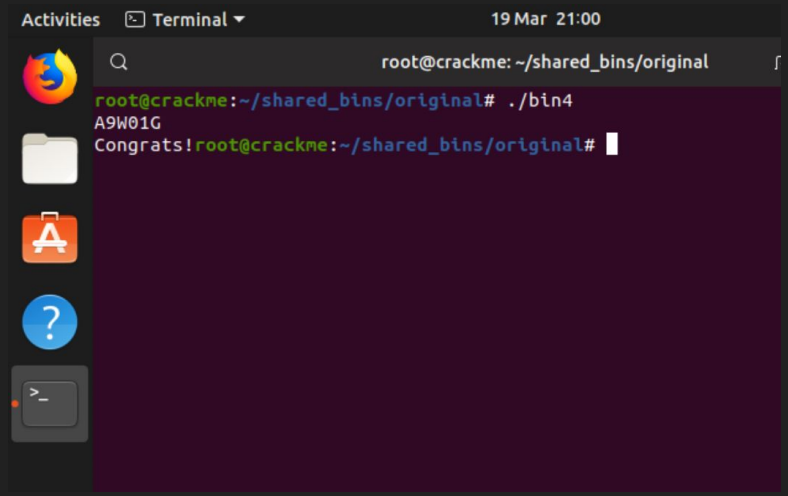


10. Analyze the shellcode file



Now we know the function!
(And the serial key)

11. Enter the key



Woo!

(I should probably change the key at some point…)

Recap

1. Loaded the binary
2. Identified rwx memory allocation
3. Wrote a script to decrypt the shellcode
4. Disassembled and decompiled the shellcode in Ghidra
5. Found the serial key!

bin5

The Patchinator

I’m bringing patching back (YEAH!)

1. main()-ly the same steps

Same old, same old

```
undefined8 main(void)

{
  char local_13 [11];

  fgets(local_13,10,stdin);
  exec(local_13);
  return 0;
}
```

Decrypt the shellcode with a key… again

A different way to copy?

Execute it… again

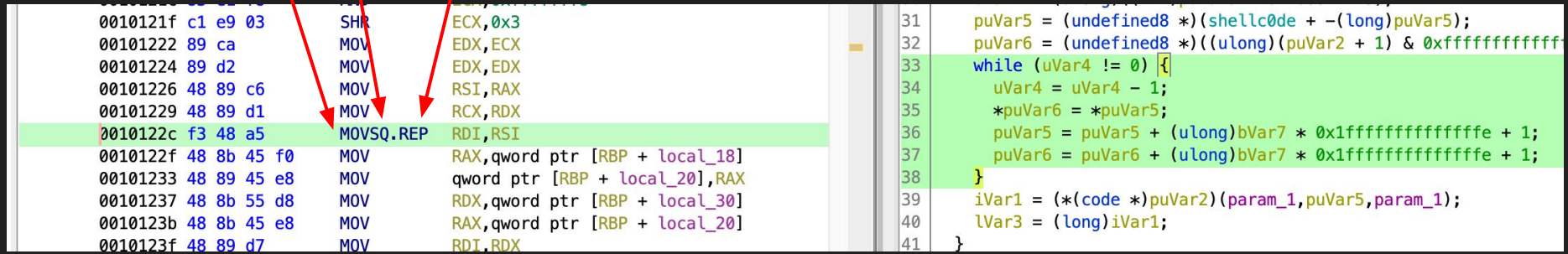
```
long exec(undefined8 param_1)

{
  int iVar1;
  undefined8 *puVar2;
  long lVar3;
  ulong uVar4;
  undefined8 *puVar5;
  undefined8 *puVar6;
  byte bVar7;
  uint local_c;

  bVar7 = 0;
  puVar2 = (undefined8 *)mmap((void *)0x0,0x188,7,0x22,-1,0);
  if (puVar2 == (undefined8 *)0xffffffffffffffff) {
    lVar3 = 0;
  }
  else {
    local_c = 0;
    while (local_c < 0x188) {
      shellc0de[(int)local_c] =
           shellc0de[(int)local_c] ^ *(byte *)((long)&key + (ulong)(local_c & 7));
      local_c = local_c + 1;
    }
    *puVar2 = shellc0de._0_8_;
    puVar2[0x30] = shellc0de._384_8_;
    puVar5 = (undefined8 *)
             ((long)puVar2 - (long)(undefined8 *)((ulong)(puVar2 + 1) & 0xfffffffffffffff8));
    uVar4 = (ulong)((int)puVar5 + 0x188U >> 3);
    puVar5 = (undefined8 *)(shellc0de + -(long)puVar5);
    puVar6 = (undefined8 *)((ulong)(puVar2 + 1) & 0xfffffffffffffff8);
    while (uVar4 != 0) {
      uVar4 = uVar4 - 1;
      *puVar6 = *puVar5;
      puVar5 = puVar5 + (ulong)bVar7 * 0x1ffffffffffffff8 + 1;
      puVar6 = puVar6 + (ulong)bVar7 * 0x1ffffffffffffff8 + 1;
    }
    iVar1 = (*(code *)puVar2)(param_1,puVar5,param_1);
    lVar3 = (long)iVar1;
  }
  return lVar3;
}
```

But a bigger *shellc0de* size?

Notes on optimisation: Different copy?

So, the assembly has changed from copying blocks directly, to using an instruction that just MOV-es Q-uadwords REP-eatedly.

A Quadword is a 64-bit value, therefore MOVSQ is only available in 64-bit architectures.



Notes on optimisation: How does it know how much to copy?

```
001011e7 b9 88 01        MOV         ECX,0x188
         00 00
001011ec 48 8b 30        MOV         RSI,qword ptr [RAX]=>shellc0de
001011ef 48 89 32        MOV         qword ptr [RDX],RSI
001011f2 89 ce           MOV         ESI,ECX
001011f4 48 01 d6        ADD         RSI,RDX
001011f7 48 8d 7e 08     LEA         RDI,[RSI + 0x8]
001011fb 89 ce           MOV         ESI,ECX
001011fd 48 01 c6        ADD         RSI,RAX
00101200 48 83 c6 08     ADD         RSI,0x8
00101204 48 8b 76 f0     MOV         RSI,qword ptr [RSI + -0x10]=>shellc0d
00101208 48 89 77 f0     MOV         qword ptr [RDI + -0x10],RSI
0010120c 48 8d 7a 08     LEA         RDI,[RDX + 0x8]
00101210 48 83 e7 f8     AND         RDI,-0x8
00101214 48 29 fa        SUB         RDX,RDI
00101217 48 29 d0        SUB         RAX,RDX
0010121a 01 d1           ADD         ECX,EDX
0010121c 83 e1 f8        AND         ECX,0xfffffff8
0010121f c1 e9 03        SHR         ECX,0x3
00101222 89 ca           MOV         EDX,ECX
00101224 89 d2           MOV         EDX,EDX
00101226 48 89 c6        MOV         RSI,RAX
00101229 48 89 d1        MOV         RCX,RDX
0010122c f3 48 a5        MOVSQ.REP   RDI,RSI
0010122f 48 8b 45 f0     MOV         RAX,qword ptr [RBP + local_18]
```

ECX is our “counter”

Here we move the size of our data

SHR - SHift Right
SHR can be used for some division
SHR 1 = divide by 2
SHR 2 = divide by 4
SHR 3 = divide by 8

Notes on optimisation: But… why?

- Compilers make optimisations wherever possible.
- Sometimes these optimisations might not be clear.
- In this case it appears that the compiler decided that mov'ing chunks with a few instructions at a time for 0x188 bytes crossed a barrier, be it for performance or for the amount of assembly generated.
- The compiler has replaced it with a REP operation.

2. Decrypt the new shellcode with Python

```python
key = [0xfe, 0xaa, 0x20, 0x01, 0x00, 0xff, 0x23, 0x98]

data = "\xab\xe2\xa9\xe4\x48\x76\x5e\x40\x38\xef\xce\x42\xc6\xba\xcc\x

def decrypt(_data, _key):
    output = ""
    for i in range(len(_data)):
        output += chr(ord(_data[i]) ^ _key[i % 8])
    return output

with open("decrypted_shellcode", 'wb') as f:
    f.write(decrypt(data, key))
```

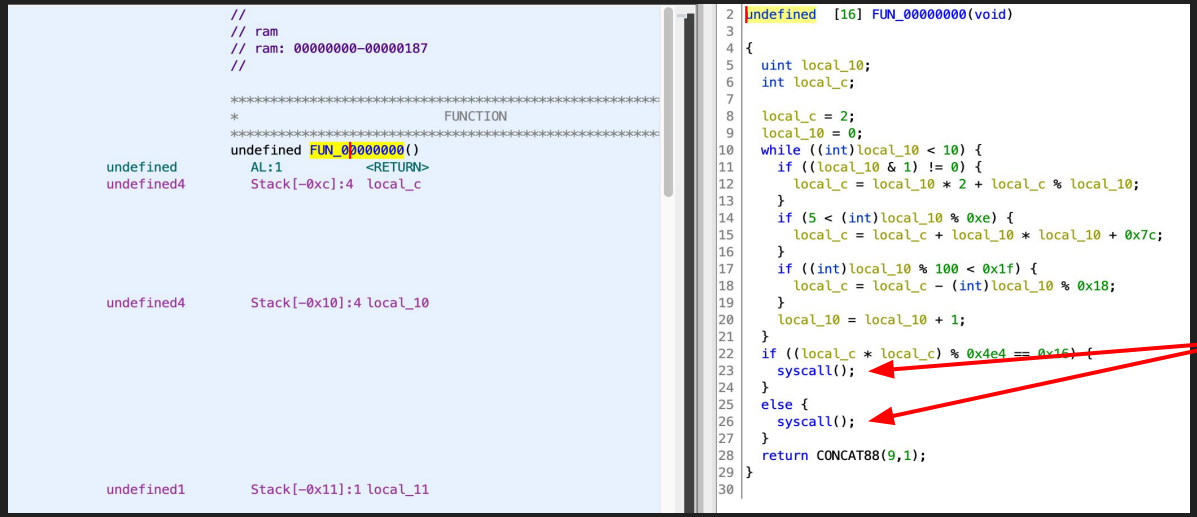
Patterns

- If you look at the first few bytes of the decrypted shellcode, you'll notice that it's the same as the first few bytes of the decrypted shellcode in the last challenge ("UH\x89\xe5")

```
Max-MBP:~ max$ hexdump -C decrypted_shellcode | head
00000000  55 48 89 e5 48 89 7d d8  c6 45 ee 43 c6 45 ef 6f  |UH..H.}..E.C.E.o|
00000010  c6 45 f0 6e c6 45 f1 67  c6 45 f2 72 c6 45 f3 61  |.E.n.E.g.E.r.E.a|
00000020  c6 45 f4 74 c6 45 f5 73  c6 45 f6 21 c6 45 f7 00  |.E.t.E.s.E.!.E..|
00000030  c7 45 fc 02 00 00 00 c7  45 f8 00 00 00 00 e9 9f  |.E......E.......|
```

- This is because the first few bytes of each function are the same:
  - \x55 -> push rbp
  - \x48\x89\xe5 -> mov rbp, esp

3. Load the new shellcode file into Ghidra



```
//
// ram
// ram: 00000000-00000187
//

*************************************************************
*                          FUNCTION
*************************************************************
undefined FUN_00000000()
undefined          AL:1           <RETURN>
undefined4         Stack[-0xc]:4  local_c

undefined4         Stack[-0x10]:4 local_10

undefined1         Stack[-0x11]:1 local_11
```

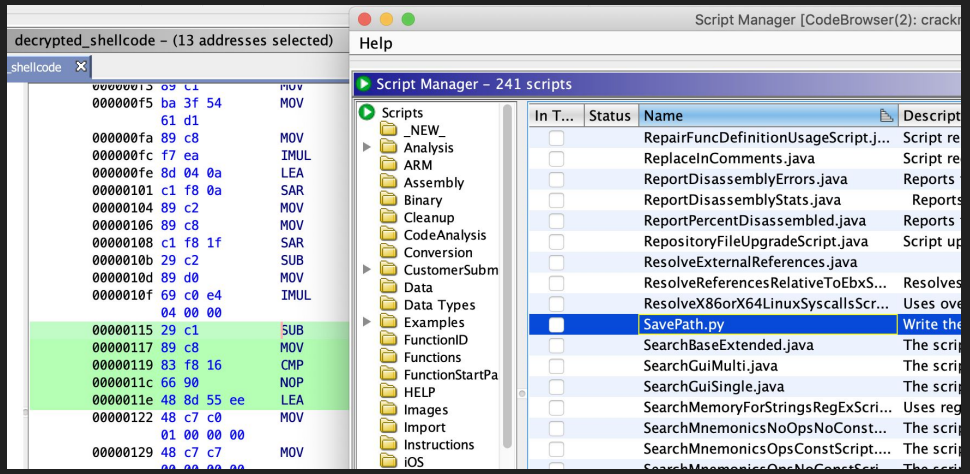
```
undefined  [16] FUN_00000000(void)

{
  uint local_10;
  int local_c;

  local_c = 2;
  local_10 = 0;
  while ((int)local_10 < 10) {
    if ((local_10 & 1) != 0) {
      local_c = local_10 * 2 + local_c % local_10;
    }
    if (5 < (int)local_10 % 0xe) {
      local_c = local_c + local_10 * local_10 + 0x7c;
    }
    if ((int)local_10 % 100 < 0x1f) {
      local_c = local_c - (int)local_10 % 0x18;
    }
    local_10 = local_10 + 1;
  }
  if ((local_c * local_c) % 0x4e4 == 0x16) {
    syscall();
  }
  else {
    syscall();
  }
  return CONCAT88(9,1);
}
```

Another NOPE function

Two syscalls at the bottom

4. Patch the shellcode

5. Export the changes using SavePath



6. Re-Encrypt

- Remember those XOR properties we mentioned a while back? You can use the decryption Python script, but replace the data variable with the decrypted shellcode in order to re-encrypt it.
- Write the output to another file

```
key = [0xfe, 0xaa, 0x20, 0x01, 0x00, 0xff, 0x23, 0x98]

with open("decrypted_shellcode", 'rb') as f:
    data = f.read()

def encrypt(_data, _key):
    output = ""
    for i in range(len(_data)):
        output += chr(ord(_data[i]) ^ _key[i % 8])
    return output

with open("encrypted_shellcode", 'wb') as f:
    f.write(encrypt(data, key))

```

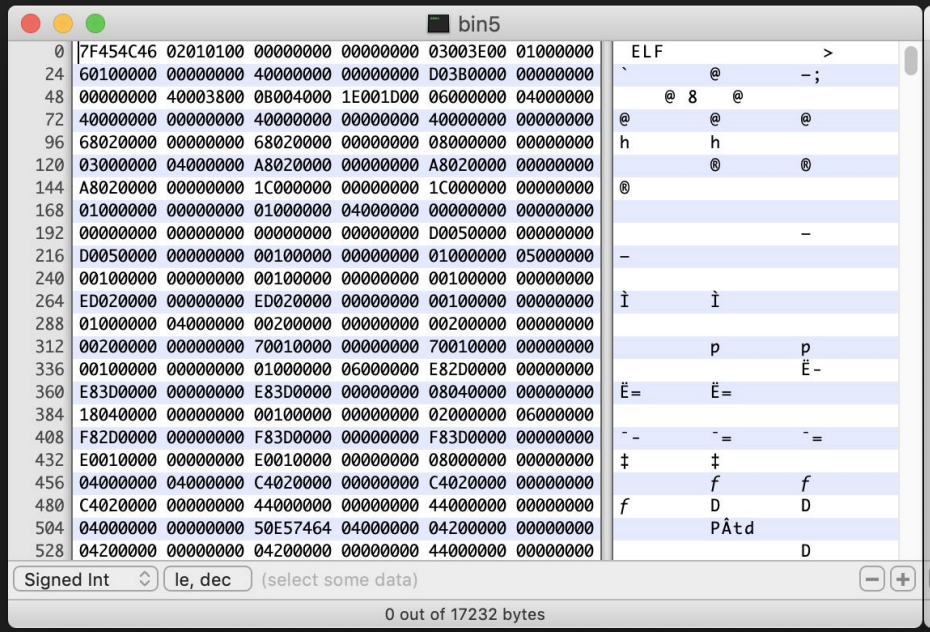
Now for the fun part...

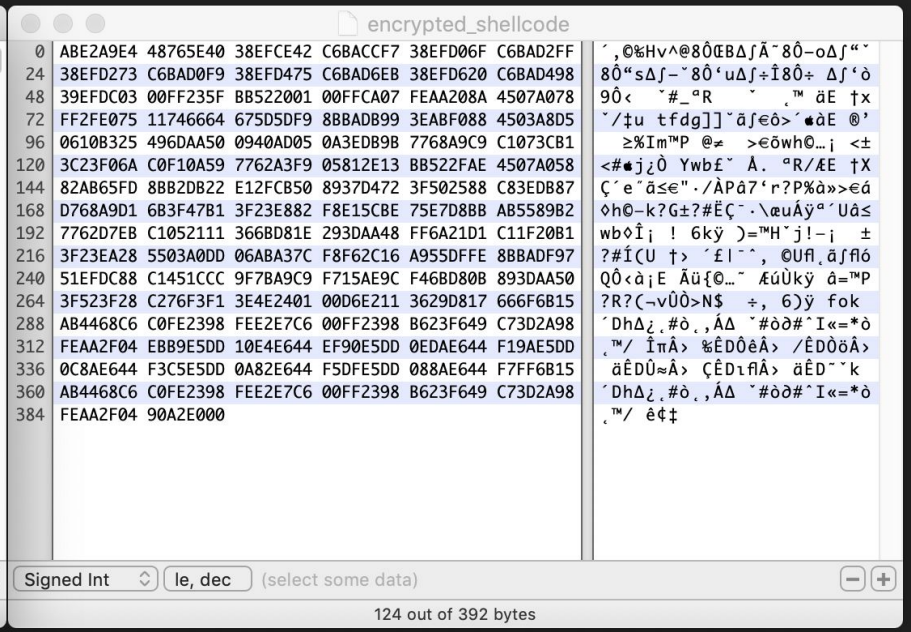
0x6 - H3X 3D171NG!

- When you think old-school hacking, what do you think of?
  - Oh yeah, Hex Editing is up near the top of that list
  - Good news, now you get to be as cool as those OG binary masters

0x6 - H3X 3D171NG!

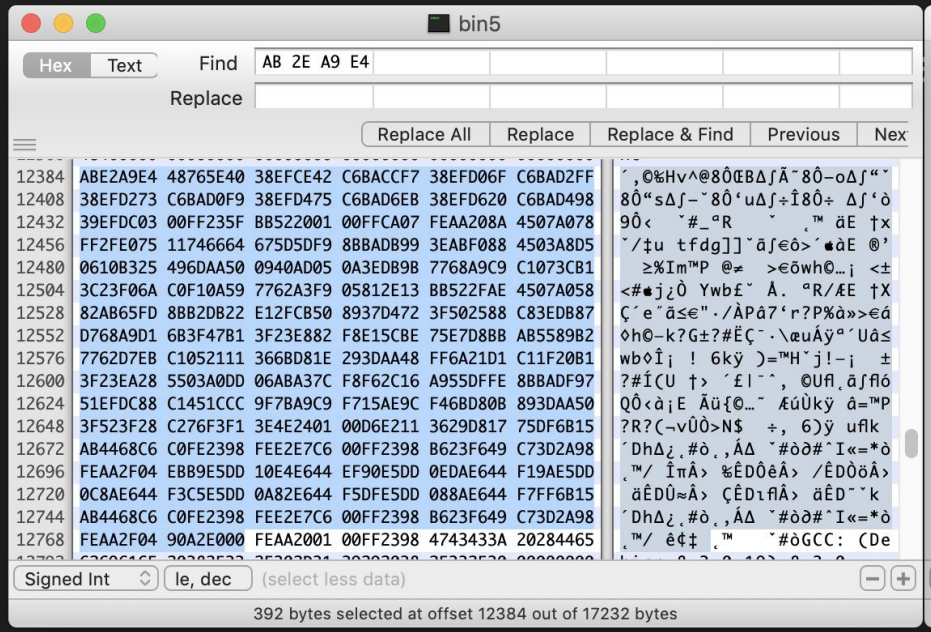
- Open both bin5 and the encrypted shellcode in your favourite hex editor

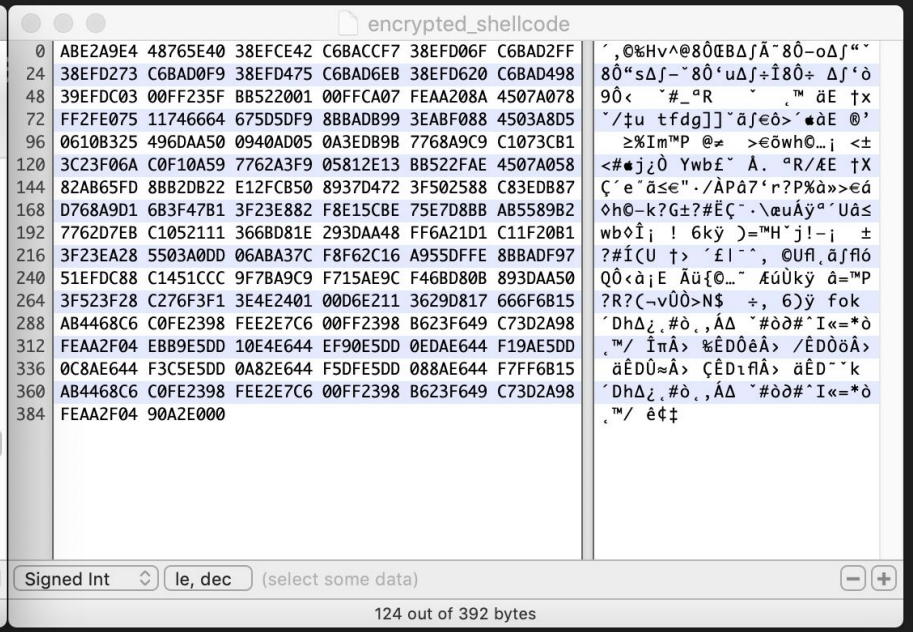




0x6 - H3X 3D171NG!

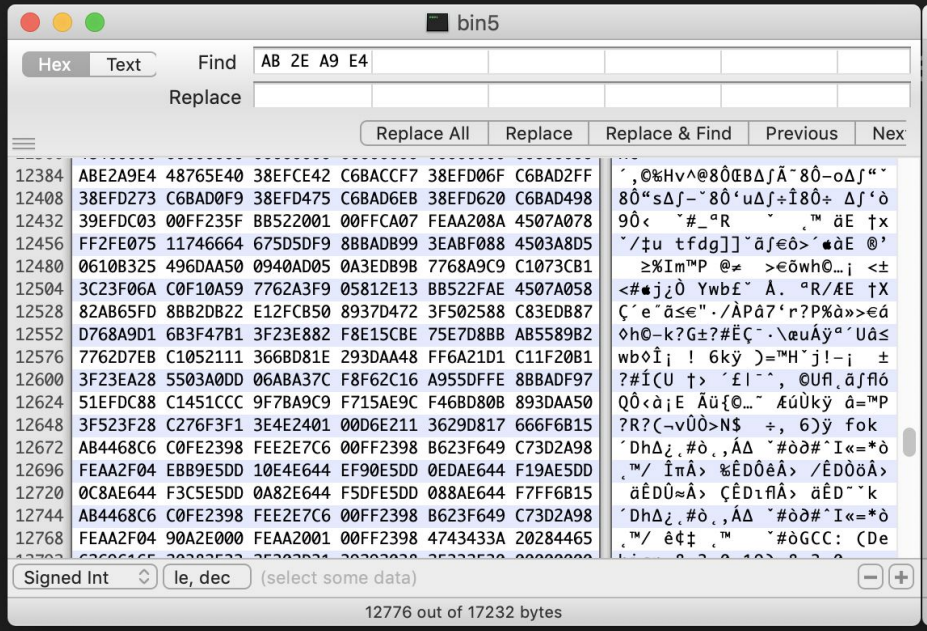
- Find the start and end of the encrypted shellcode in bin5

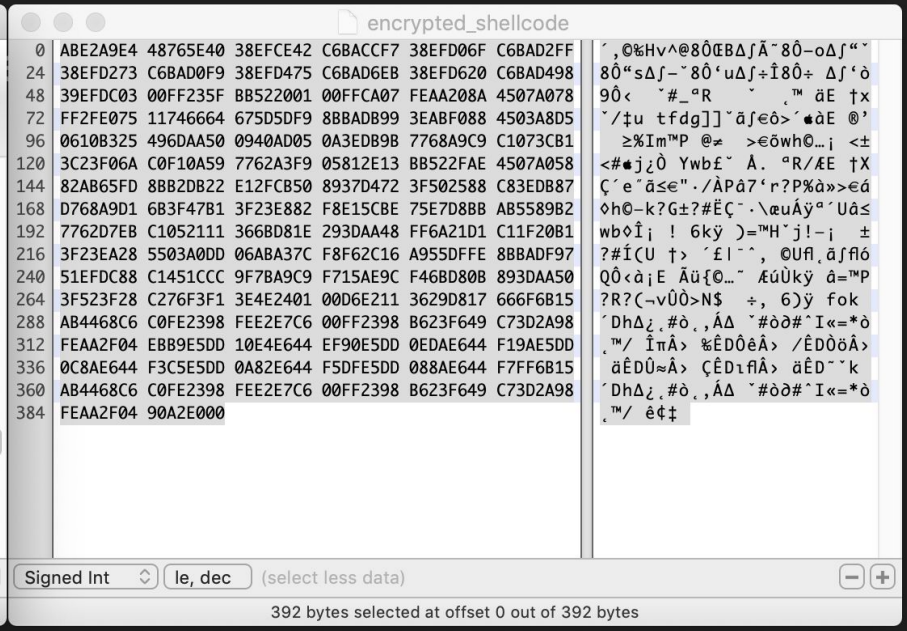




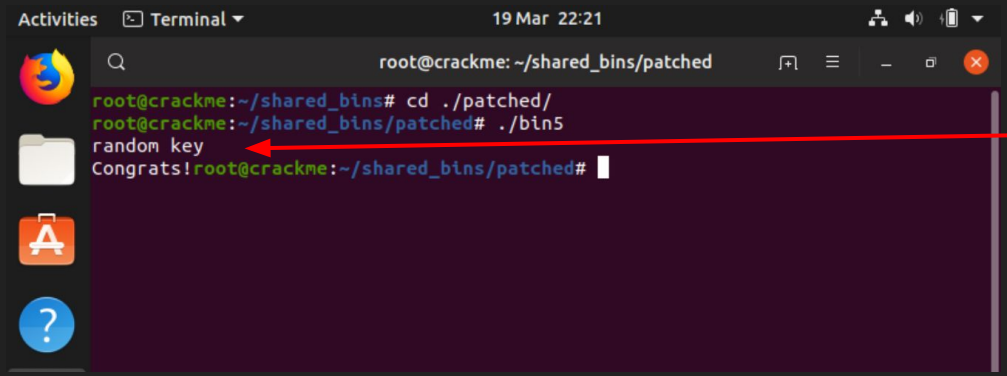
0x6 - H3X 3D171NG!

- Replace the shellcode in bin5 with the encrypted_shellcode file contents (and save)





7. Run the patched bin5 binary in the VM



All license keys are now accepted!

Recap

1. Loaded the binary
2. Identified rwx memory allocation
3. Decrypted shellcode
4. Patched decrypted shellcode
5. Re-encrypted patched shellcode
6. Hex-edited it into the original binary
7. Ran it!

bin6

The final frontier

1. You know what to do...

```
undefined8 main(void)

{
  char local_13 [11];

  fgets(local_13,0xb,stdin);
  exec(local_13);
  return 0;
}
```

mmap 0x4e bytes

Some decryption routine

...an strncmp?

...a memcpy?!

The infamous function call

```
long exec(long param_1)

{
  int iVar1;
  code *__dest;
  long lVar2;
  uint local_c;

  __dest = (code *)mmap((void *)0x0,0x4e,7,0x22,-1,0);
  if (__dest == (code *)0xffffffffffffffff) {
    perror("mmap");
    lVar2 = 0;
  }
  else {
    local_c = 0;
    while (local_c < 0x58) {
      shellc0de[(int)local_c] = shellc0de[(int)local_c] ^ *(byte *)(param_1 + (int)local_c % 10)
      local_c = local_c + 1;
    }
    iVar1 = strncmp(shellc0de,&DAT_00402009,10);
    if (iVar1 == 0) {
      memcpy(__dest,shellc0de + 10,0x4e);
      iVar1 = (*__dest)(param_1);
      lVar2 = (long)iVar1;
    }
    else {
      puts("Failed.");
      lVar2 = 0;
    }
  }
  return lVar2;
}
```

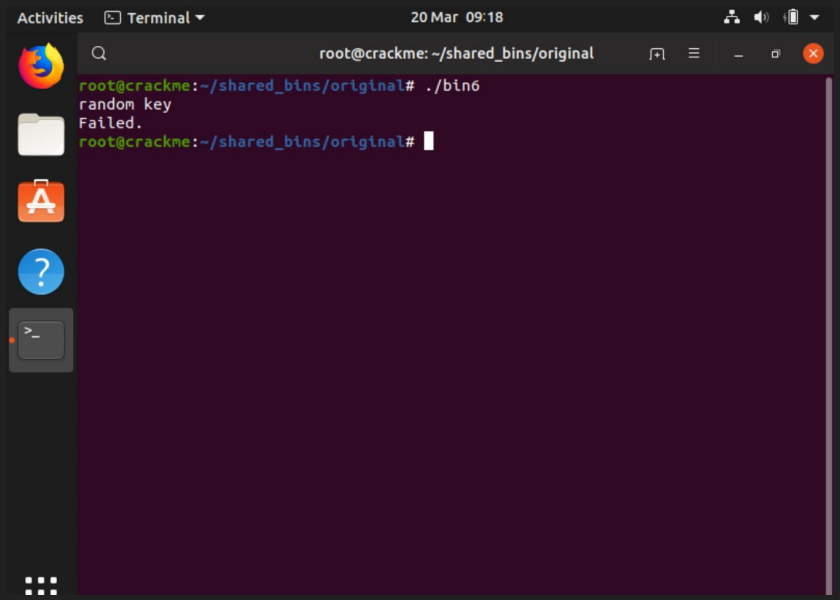
2. Magic Decryption

param_1 is our serial key

Take each byte of *shellc0de* and XOR it with the result of *param_1[local_c % 10]*

```
local_c = 0;
while (local_c < 0x58) {
  shellc0de[(int)local_c] = shellc0de[(int)local_c] ^ *(byte *)(param_1 + (int)local_c % 10)
  local_c = local_c + 1;
}
```

But… won’t it crash if we give it the wrong key?



Apparently not!

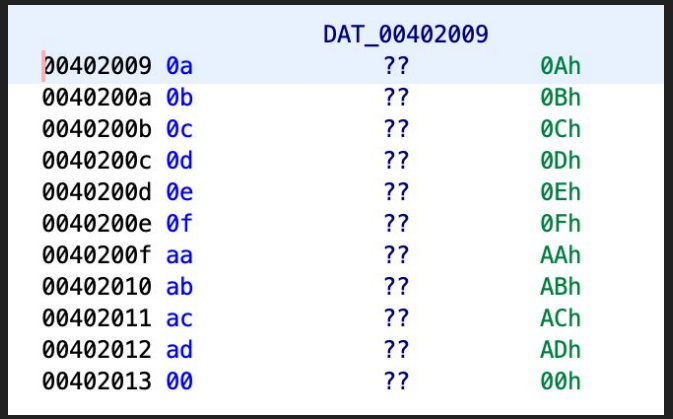
Why is that?

```
iVar1 = strncmp(shellc0de,&DAT_00402009,10);
if (iVar1 == 0) {
  memcpy(__dest,shellc0de + 10,0x4e);
  iVar1 = (*__dest)(param_1);
  lVar2 = (long)iVar1;
}
else {
  puts("Failed.");
  lVar2 = 0;
}
```

After the decryption, the program performs an strncmp of the first 10 bytes of *shellc0de*

If the strncmp fails, it prints “Failed.”

3. Magic Bytes



```
                    DAT_00402009
00402009 0a           ??         0Ah
0040200a 0b           ??         0Bh
0040200b 0c           ??         0Ch
0040200c 0d           ??         0Dh
0040200d 0e           ??         0Eh
0040200e 0f           ??         0Fh
0040200f aa           ??         AAh
00402010 ab           ??         ABh
00402011 ac           ??         ACh
00402012 ad           ??         ADh
00402013 00           ??         00h
```

Looks like DAT_00402009 has 10 interesting bytes in it!

It appears that these are magic-bytes used to check whether the decryption was correct.

3. Magic Bytes

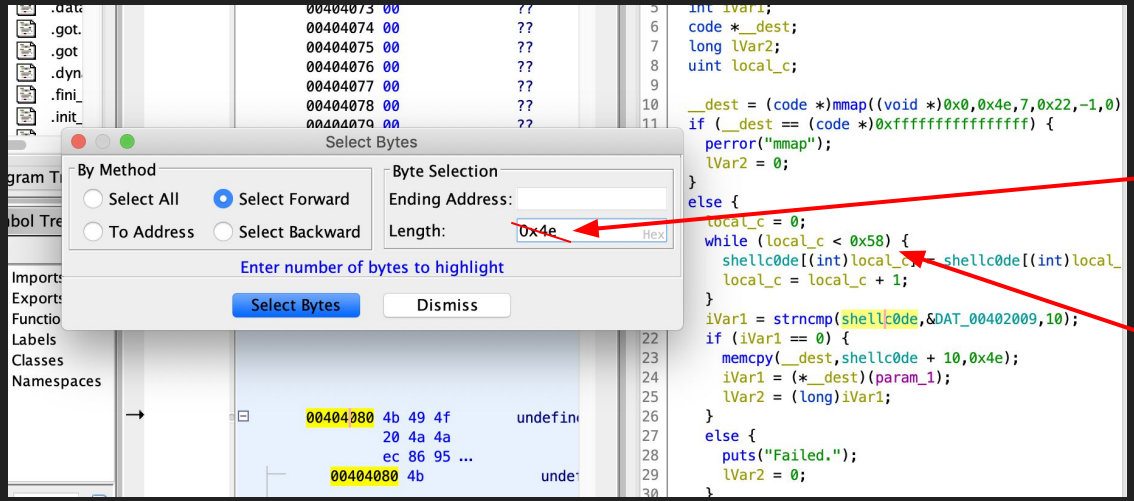
```
iVar1 = strncmp(shellc0de,&DAT_00402009,10);
if (iVar1 == 0) {
  memcpy(__dest,shellc0de + 10,0x4e);
  iVar1 = (*__dest)(param_1);
  lVar2 = (long)iVar1;
}
else {
  puts("Failed.");
  lVar2 = 0;
}
```

Also notice that the first 10 bytes are skipped when copying the *shellc0de*

4. Decrypting *shellc0de*

- Back to those XOR properties!
- shellc0de[0...9] ^ serial_key = magic_bytes
- Therefore: shellc0de[0...9] ^ magic_bytes = serial_key

Careful!



You might be thinking “mmap uses 0x4e, so that’s the size of *shellc0de*”

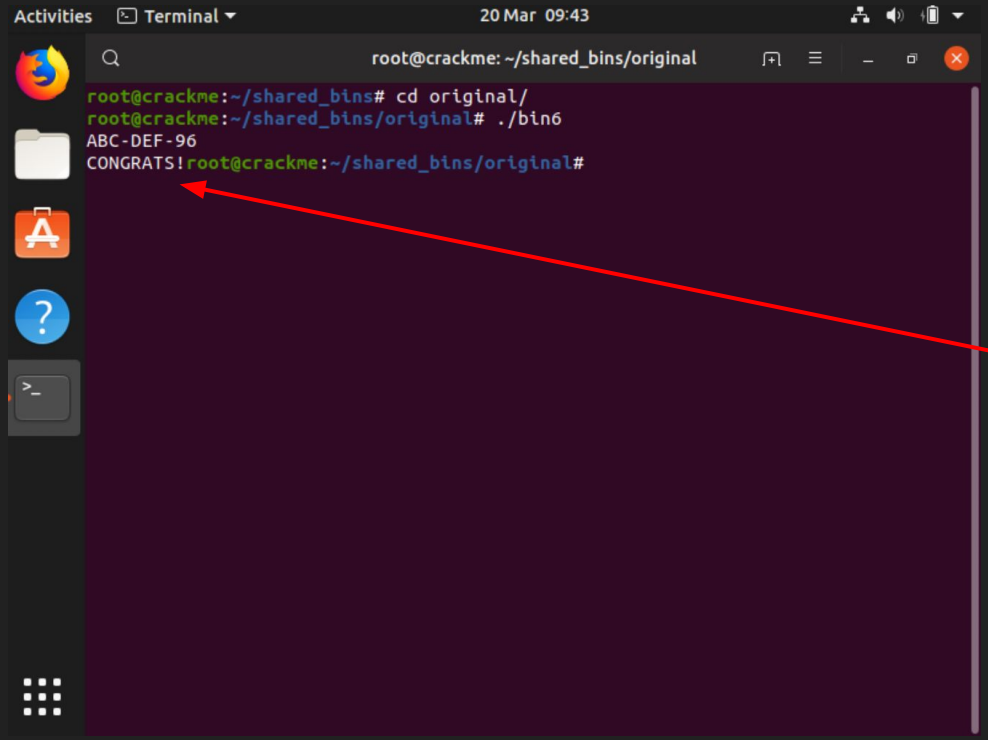
Don’t forget about the magic bytes! It’s really 0x4e + 10 (0x58)

4. Decrypting *shellc0de*

```
1   magic_bytes = [0xa, 0xb, 0xc, 0xd, 0xe, 0xf, 0xaa, 0xab, 0xac, 0xad]
2
3   data = "\x4b\x49\x4f\x20\x4a\x4a\xec\x86\x95\x9b\x14\x0a\xca\xc8\x0c\xcc\x3b\xc5\xff\
4
5   def decrypt(_data, _key):
6       output = ""
7       for i in range(len(_data)):
8           output += chr(ord(_data[i]) ^ _key[i % 10])
9       return output
10
11  decrypt(data, magic_bytes)[:10]
```

```
>>> def decrypt(_data, _key):
...     output = ""
...     for i in range(len(_data)):
...             output += chr(ord(_data[i]) ^ _key[i % 10])
...     return output
...
>>> decrypt(data, magic_bytes)[:10]
'ABC-DEF-96'
>>>
```

5. Run it!



A much more enthusiastic “Congrats!” message because you’ve just completed bin6 and you should be very proud!

Recap

1. Loaded the binary
2. Identified rwx memory allocation
3. Identified the magic bytes
4. Decrypted the serial key from the magic bytes
5. Ran it!

Final Words

- I hope you've enjoyed reading this.
- I had aimed to put in as much of the information taught during the F-Secure live stream as possible, as well as include much more (surprisingly it's much easier to think of things when you're not on the spot!)
- While this isn't meant to teach EVERYTHING about reverse engineering, the aim is to get more people into it by showing them some of the things you can do and the free tools that can be used in the hope that it'll feed the curious part of the brain to go out and keep learning.

- Any questions? Feel free to message!