

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 make 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!)

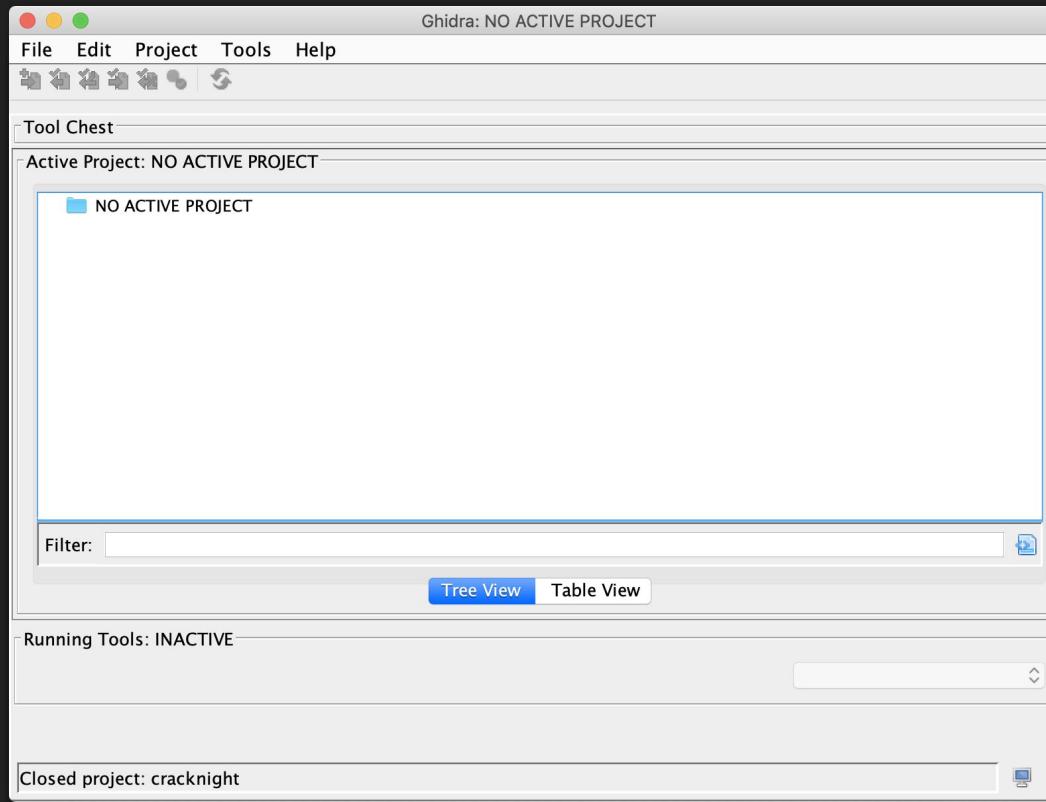
Setup

You Will Need:

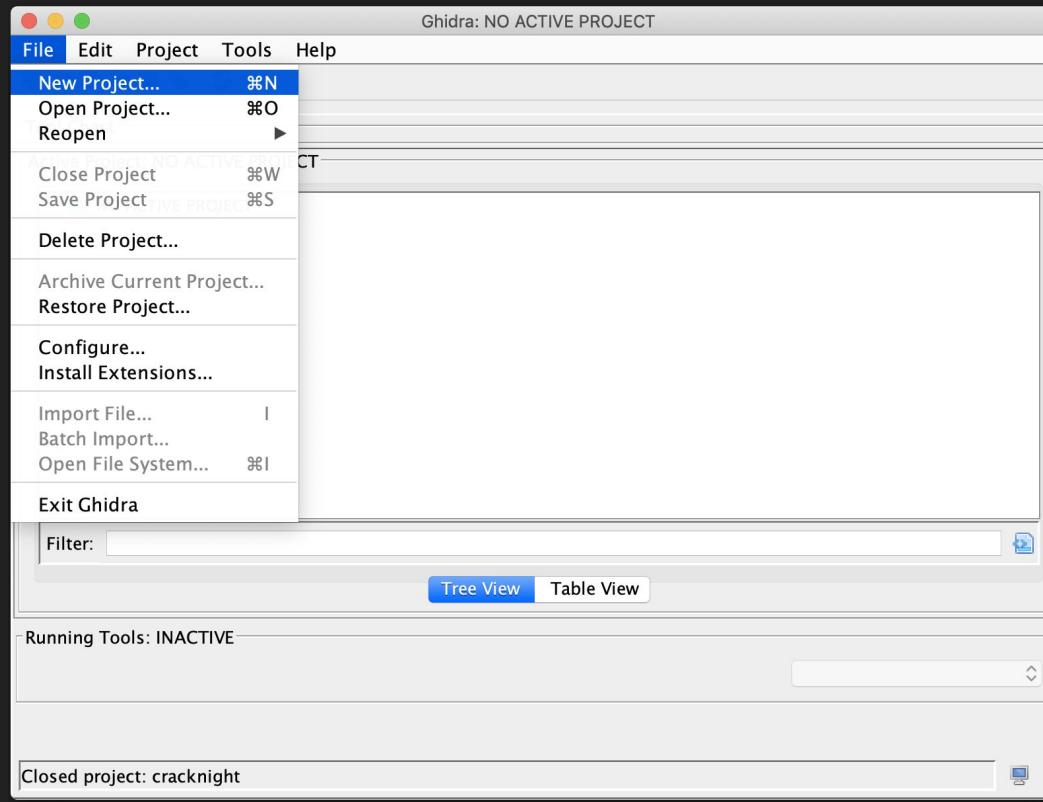
- 64-bit Linux VM (I chose Ubuntu 19)
 - Install GDB on it (apt-get install gdb) - This will be our debugger
- Ghidra (This is our disassembler && decompiler)
 - Either on the host or on the VM itself
 - If Ghidra is on the host, then make sure you've got a way to easily transfer files between the host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

Ghidra Setup

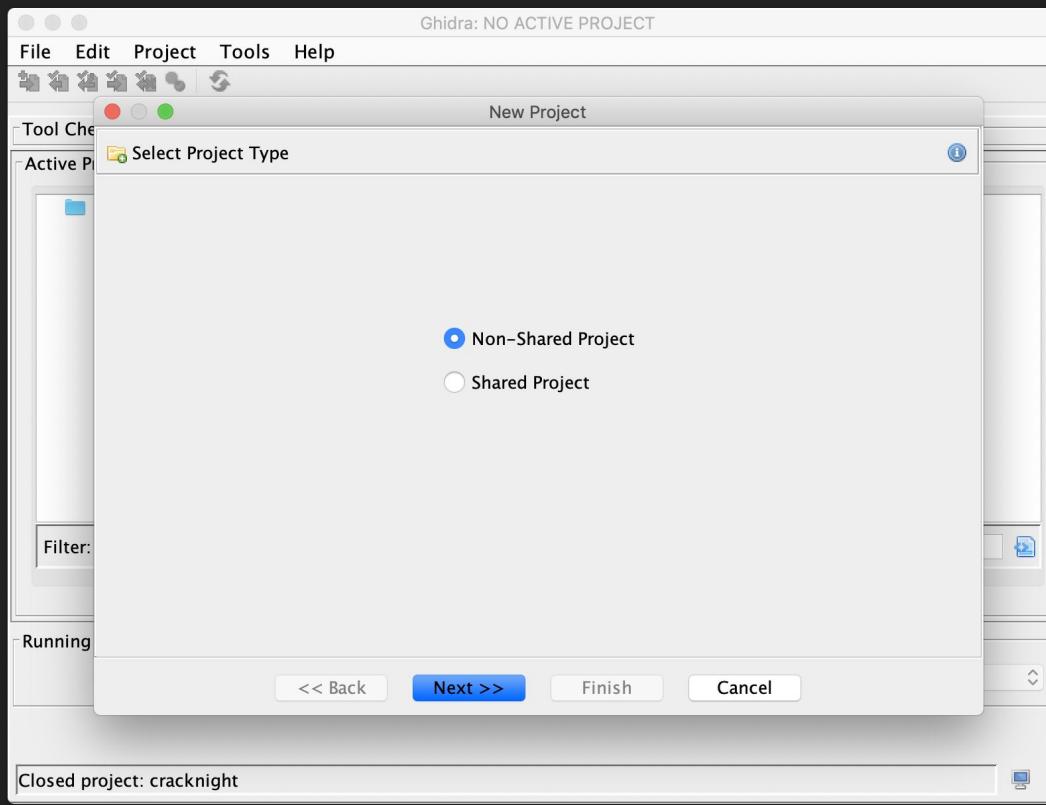
1. Open Ghidra



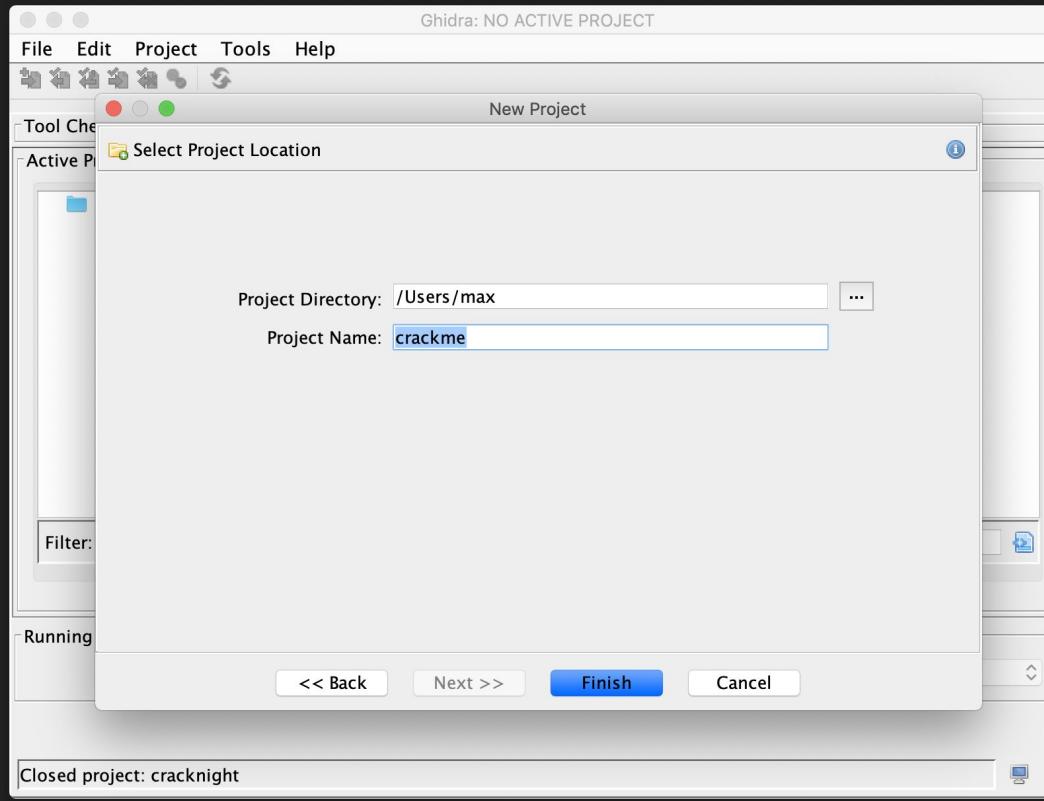
2. Create a new project



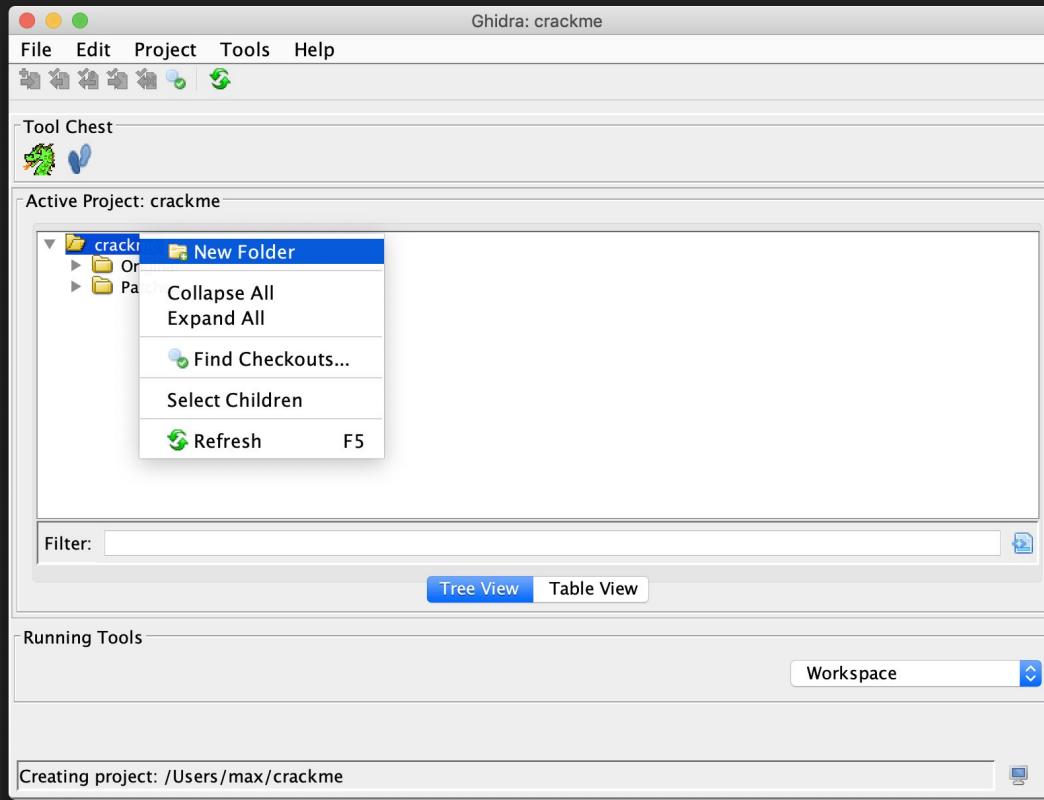
3. Non-Shared



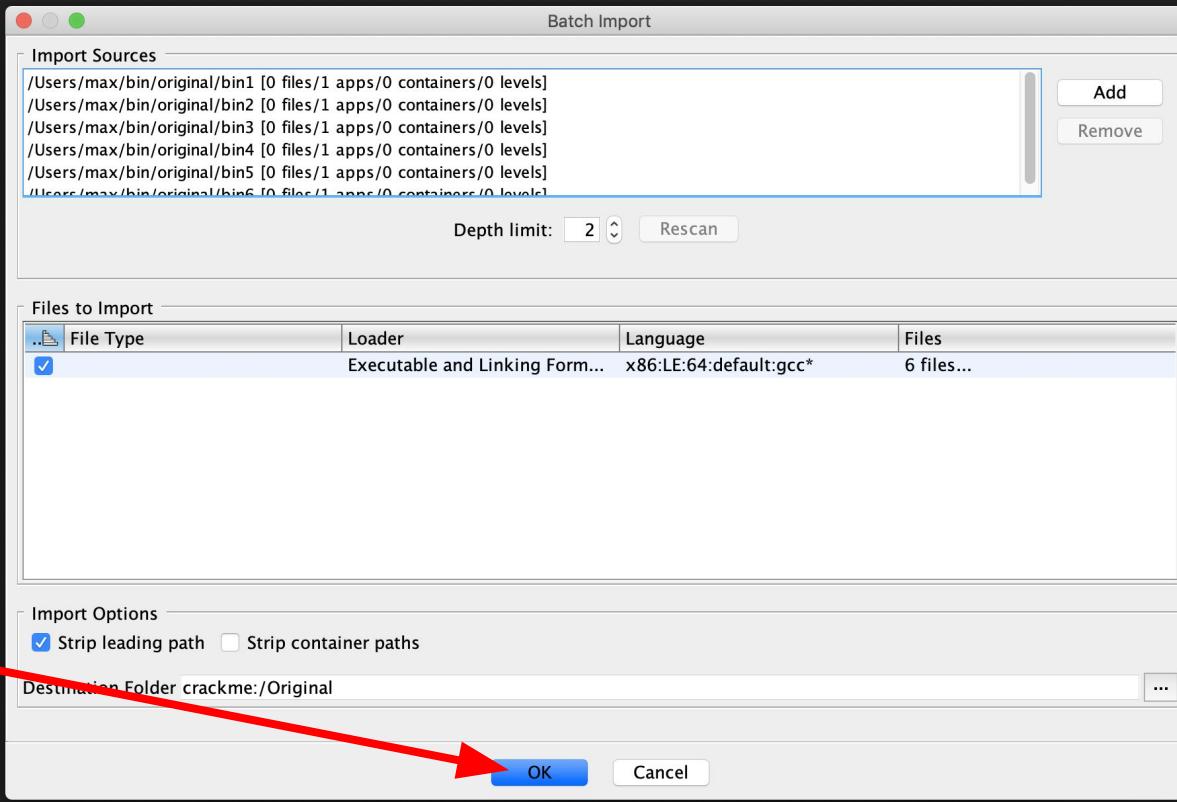
4. Pick a project name



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



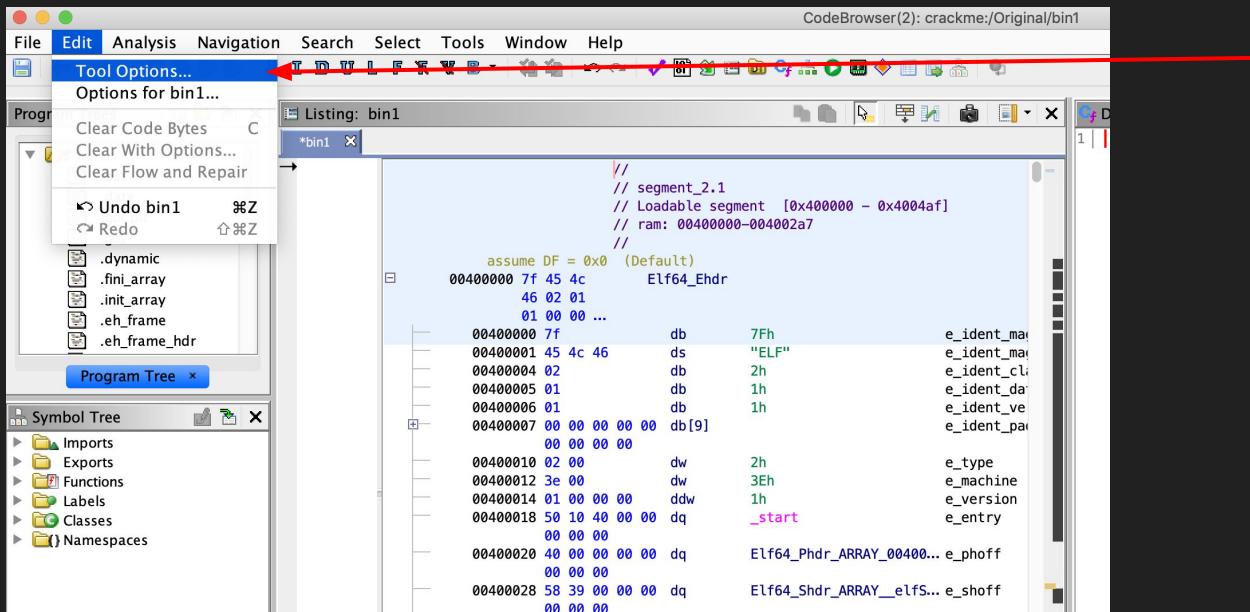
6. Drag 'n Drop the binaries into the Original folder



8. Setup Highlighting in Tool Options

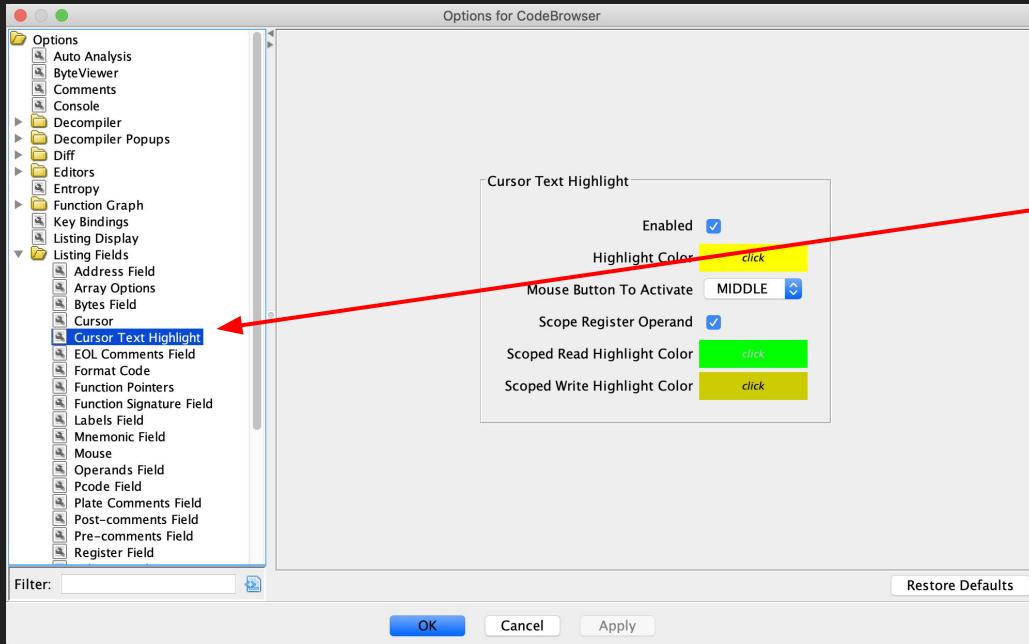
- Highlighting is where you click on a variable and it also highlights other uses of the variable
- This is incredibly useful for when you have large functions and want to see whether or where a variable is used without having to strain your eyes searching!

8. Setup Highlighting in Tool Options



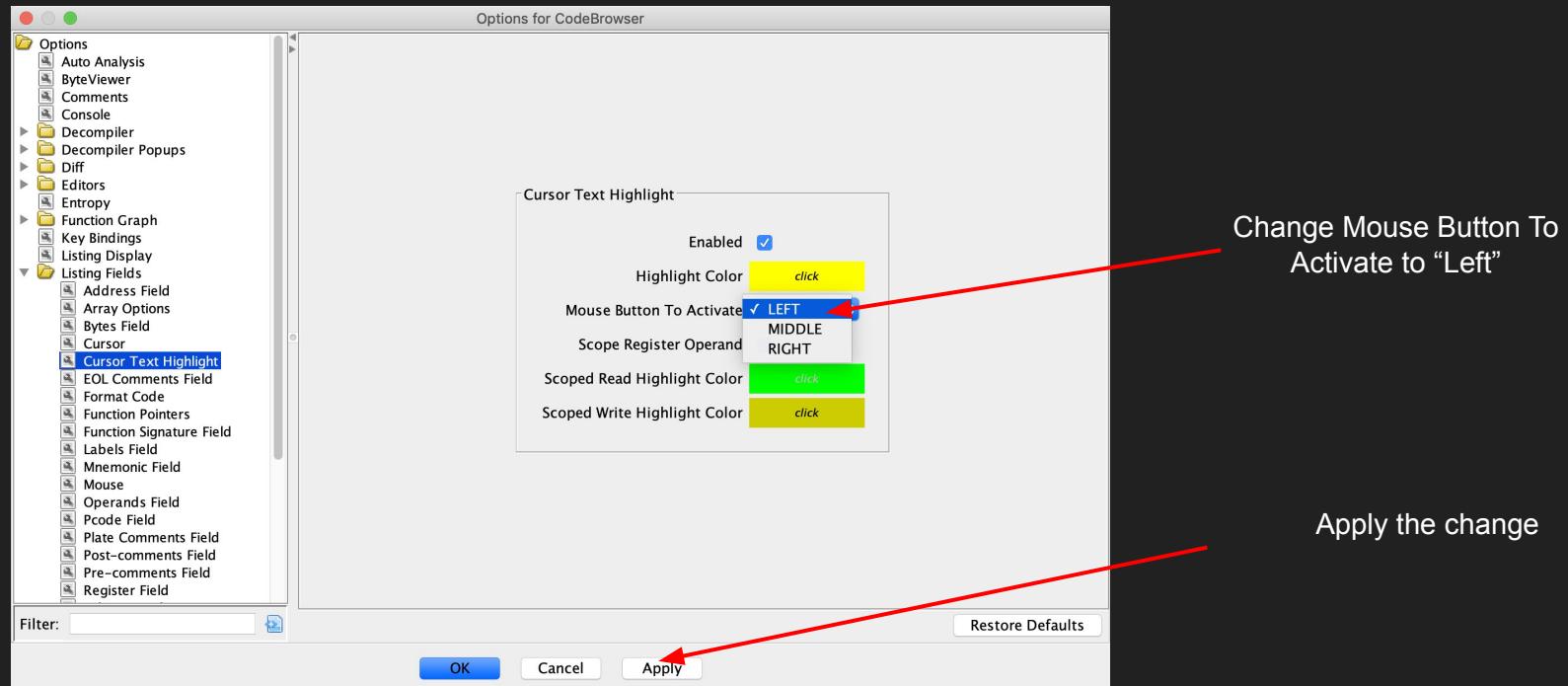
Open up one of the binaries and then click on Tool Options

8. Setup Highlighting in Tool Options



Go to Listing Fields -> Cursor
Text Highlight

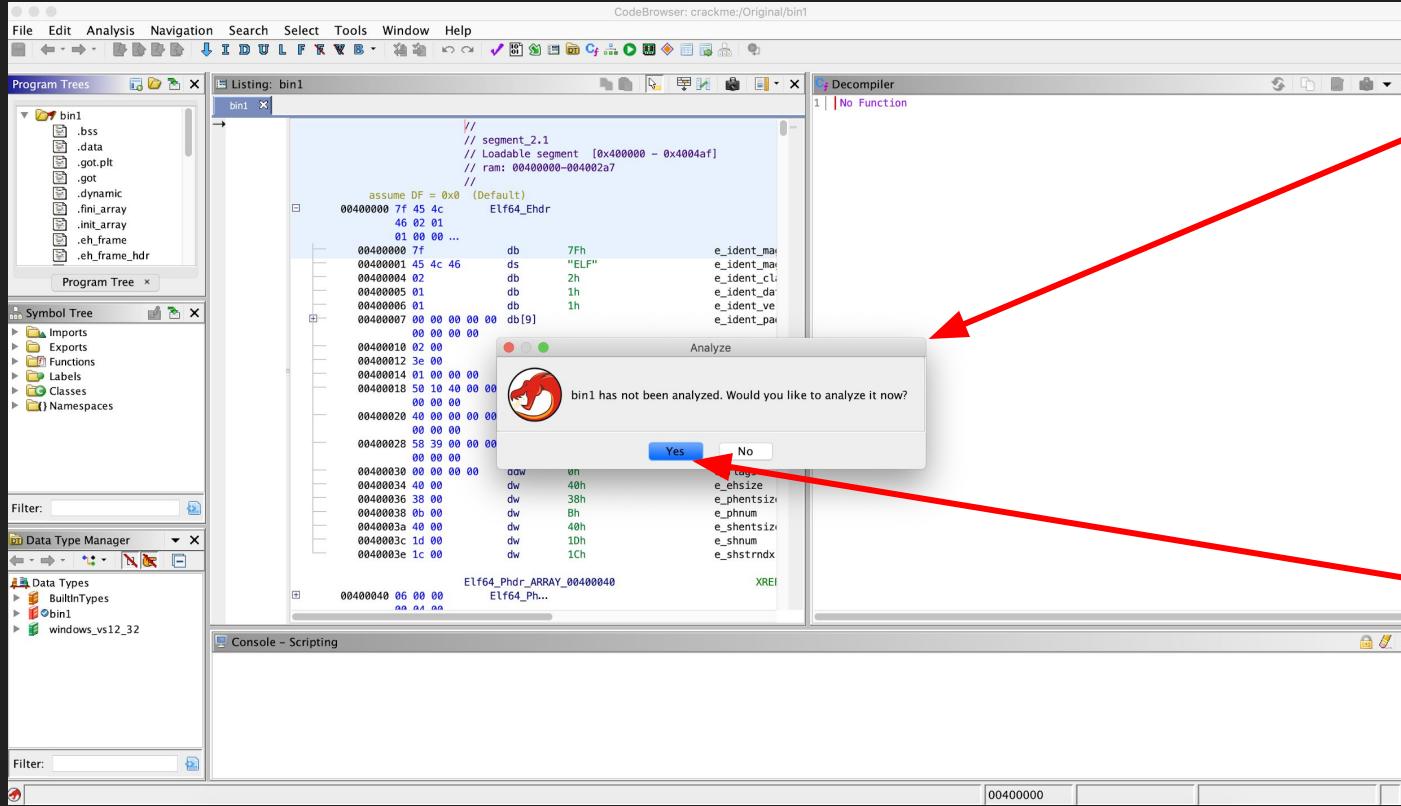
8. Setup Highlighting in Tool Options



bin1

So let's get started!

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

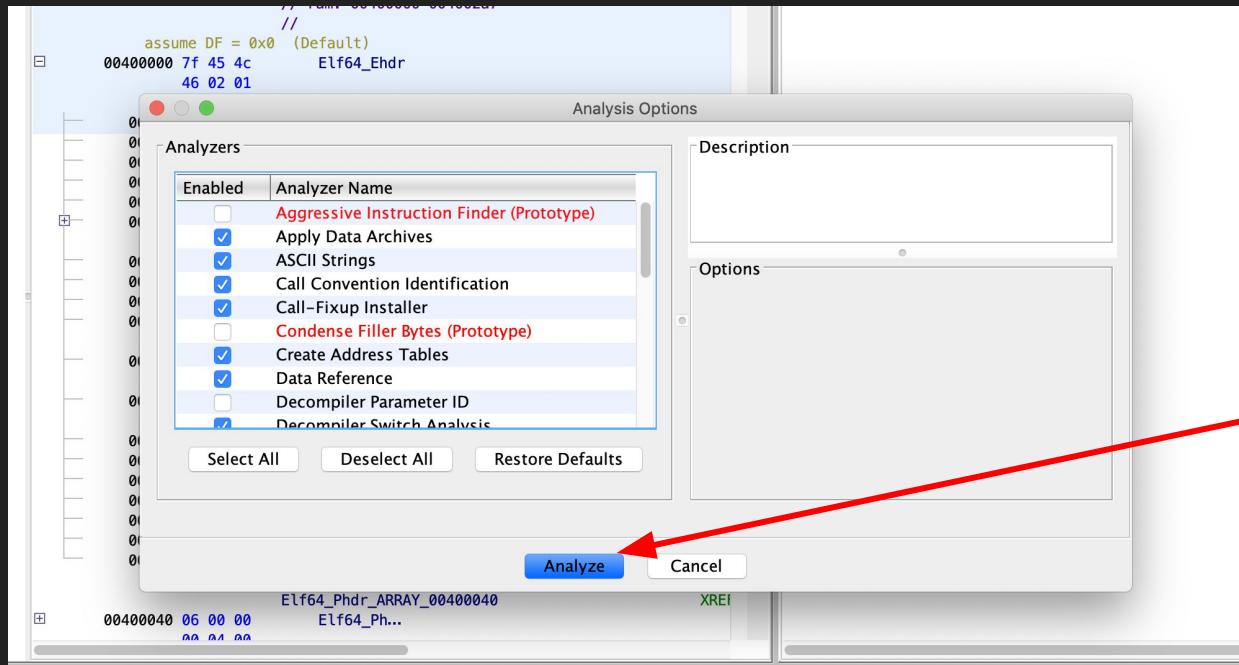


You will be prompted to analyze the file

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?

Listing: bin1

*bin1 X

```
//  
// segment_2.1  
// Loadable segment [0x400000 - 0x4004af]  
// ram: 00400000-004002a7  
//  
assume DF = 0x0 (Default)  
00400000 7f 45 4c Elf64_Ehdr  
    46 02 01  
    01 00 00 ...  
00400000 7f db 7Fh e_ident_mag  
00400001 45 4c 46 ds "ELF" e_ident_mag  
00400004 02 db 2h e_ident_cl  
00400005 01 db 1h e_ident_da  
00400006 01 db 1h e_ident_ve  
00400007 00 00 00 00 00 db[9] e_ident_pa  
    00 00 00 00  
00400010 02 00 dw 2h e_type  
00400012 3e 00 dw 3Eh e_machine  
00400014 01 00 00 00 ddw 1h e_version  
00400018 50 10 40 00 00 dq _start e_entry  
    00 00 00  
00400020 40 00 00 00 00 dq Elf64_Phdr_ARRAY_00400... e_phoff  
    00 00 00  
00400028 58 39 00 00 00 dq Elf64_Shdr_ARRAY_elfs... e_shoff  
    00 00 00  
00400030 00 00 00 00 ddw 0h e_flags  
00400034 40 00 dw 40h e_ehsize  
00400036 38 00 dw 38h e_phentsize  
00400038 0b 00 dw Bh e_phnum  
0040003a 40 00 dw 40h e_shentsize  
0040003c 1d 00 dw 1Dh e_shnum  
0040003e 1c 00 dw 1Ch e_shstrndx  
Elf64_Phdr_ARRAY_00400040  
00400040 06 00 00 Elf64_Ph...
```

Here it is!

3. Double click on `_start`

The screenshot shows the Immunity Debugger interface with two main panes. The left pane displays the assembly listing for the `_start` function, which contains standard C startup code. The right pane shows the corresponding decompiled C pseudocode. Red arrows point from the assembly code in the left pane to the decompiled code in the right pane, indicating the relationship between the two.

Disassembly - The assembly instructions in the program

Decompiler - C pseudocode that is generated based on the assembly

```
void _start(undefined8 param_1,undefined8 param_2,undefined8 param_3)
{
    undefined8 in_stack_00000000;
    undefined auStack [8];
    __libc_start_main(main,in_stack_00000000,&stackIdx0x00000008,__libc_csu_init,__libc_csu_fini,param_3,austack8);
    do {
        /* WARNING: Do nothing block with infinite loop */
    } while( true );
}
```

```
00401040 ff 25 da          XOR    EBP,EBP
00401046 ff 00 00          MOV    R0,00
0040104d e9 ff ff          JMP    FIN_00401020
00401050 ff ff             XOR    EBP,EBP
00401052 48 89 d1          MOV    R0,RDX
00401055 5e                POP    RSI
00401058 48 89 e2          MOV    RDX,RSP
0040105b 48 83 e4 f0        AND    RSP,-0x10
0040105e 48 c7 c9          PUSHD RSP
0040105f 49 c7 c8          MOV    R0,local_10
00401060 eb 12 48 00        JNE    00401060
00401063 48 c7 c7          MOV    R0,__libc_csu_fini,__libc_csu_fini
00401066 eb 12 48 00        JNE    00401066
0040106d 48 c7 c7          MOV    R0,main.main
```

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*

The screenshot shows the Immunity Debugger interface with the following windows:

- Program Tree:** Shows the project structure with a file named `bin1`.
- Symbol Tree:** Shows imports, exports, functions, labels, classes, and namespaces.
- Listing: bin1:** Displays the assembly code for the `bin1` file. The assembly code includes instructions like `MOV EAX, 0x0`, `POP RBP`, and `RET`. It also shows the definition of the `main` function.
- Decompile: main - (bin1):** Displays the C-like pseudocode for the `main` function. The code reads a serial number from `stdin`, calls `check_serial`, and prints a message based on the result.
- Console - Scripting:** An empty console window.

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

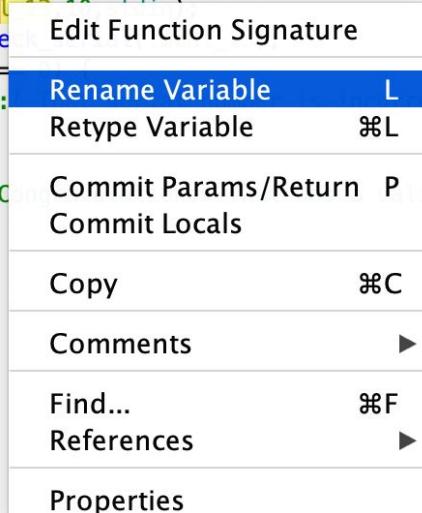
Using what we've just learned, we know that:
fgets(local_13,10,stdin) == "Read 9 bytes from the *input* into buffer *local_13*"

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*

```
1 undefined8 main(void)
2
3 {
4     int iVar1;
5     char local_13 [11];
6
7     fgets(local_13, 11, stdin);
8     iVar1 = checkSerialNumber(local_13);
9     if (iVar1 == 1) {
10         printf("Success\n");
11     }
12     else {
13         printf("Failure\n");
14     }
15     return 0;
16 }
```



A screenshot of a code editor showing a context menu for the variable 'local_13'. The menu options are:

- Edit Function Signature
- Rename Variable (highlighted)
- Retype Variable
- Commit Params/Return
- Commit Locals
- Copy
- Comments
- Find...
- References
- Properties

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

NOPE.

What does *check_serial* do?

```
1 ulong check_serial(void) ← Incorrect number of arguments
2 {
3     uint local_10;
4     uint local_c;
5
6     local_c = 0x7fafaf31;
7     local_10 = 0;
8     while ((int)local_10 < 10) {
9         if ((local_10 & 1) != 0) {
10            local_c = (int)local_c % (local_10 * local_10 + local_10); ← Crazy modulus magic
11        }
12        if (3 < (int)local_10 % 0xe) ←
13            local_c = local_c + (2 - local_10 * local_10);
14        }
15        if ((int)local_10 % 9 == 0) { ← N.O.P.E.
16            local_c = (int)local_10 % 0x4a + 0xfaU ^ local_10;
17        }
18        local_10 = local_10 + 1;
19    }
20    return (ulong)(local_c == 0x662);
21 }
22 }
```

Why are the arguments messed up?

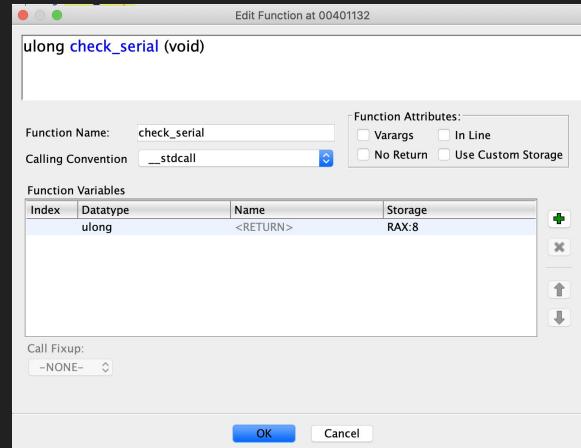
- Decompilers aren't always correct.
- They work by making lots of guesses and assumptions and detecting patterns.
 - “Ah, while this bit of assembly is repeating, this variable is decrementing. This must be a for-loop!”
- Although decompilers are incredibly helpful for quickly understanding functions, 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?

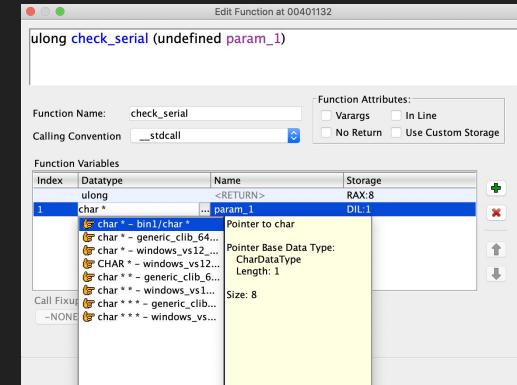
1) Edit Function Signature

```
1 ulong check_serial();
2 {
3     uint local_c;
4     uint local_10;
5     local_c = 0;
6     local_10 = 0;
7     while ((int)local_c < 10) {
8         if (((local_c * 9) + local_10) % 10 == 0) {
9             local_c = (int)local_10 % 0x4a + 0xfaU ^ local_10;
10        }
11        local_10 = local_10 + 1;
12    }
13    if (3 < (int)local_c) {
14        local_c = local_c - 1;
15    }
16    Properties
17    if ((int)local_10 % 9 == 0) {
18        local_c = (int)local_10 % 0x4a + 0xfaU ^ local_10;
19    }
20 }
```

2) Click + to add an argument

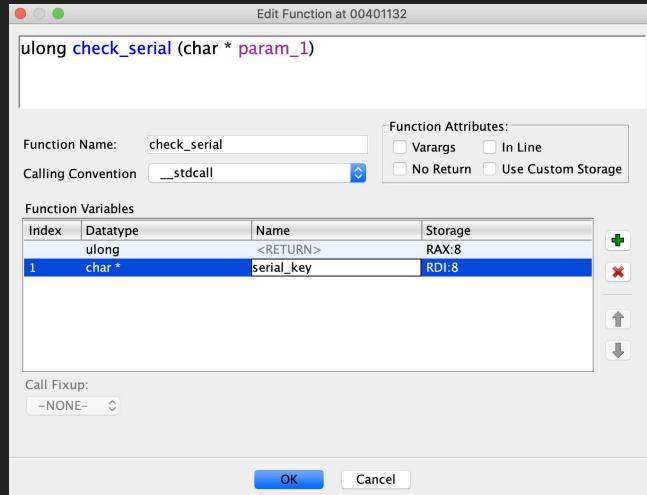


3) Change the Datatype to `char *`



Can we fix the arguments?

4) Rename the parameter



So... What happened?

```
1  ulong check_serial(char *serial_key)
2
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

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

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

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

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

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

To jump, or not to jump?

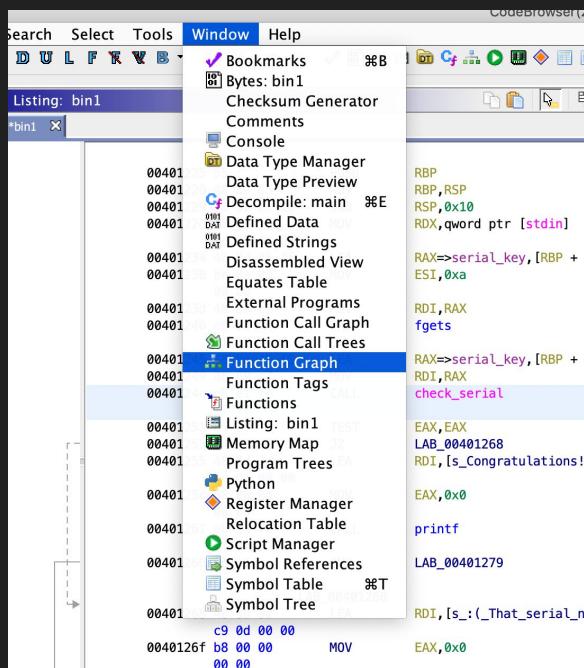
```
00401251 85 c0      TEST    EAX,EAX
00401253 74 13      JZ      LAB_00401268
00401255 48 8d 3d    LEA     RDI,[s_Congratulations!_That_was_a_v
                      ac 0d 00 00
0040125c b8 00 00    MOV     EAX,0x0
                      00 00
00401261 e8 ca fd    CALL    printf
                      ff ff
00401266 eb 11      JMP     LAB_00401279
                      LAB_00401268
00401268 48 8d 3d    LEA     RDI,[s_:(_That_serial_number_is_inco
                      c9 0d 00 00
0040126f b8 00 00    MOV     EAX,0x0
                      00 00
00401274 e8 b7 fd    CALL    printf
                      ff ff
                      LAB_00401279
```

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 :(

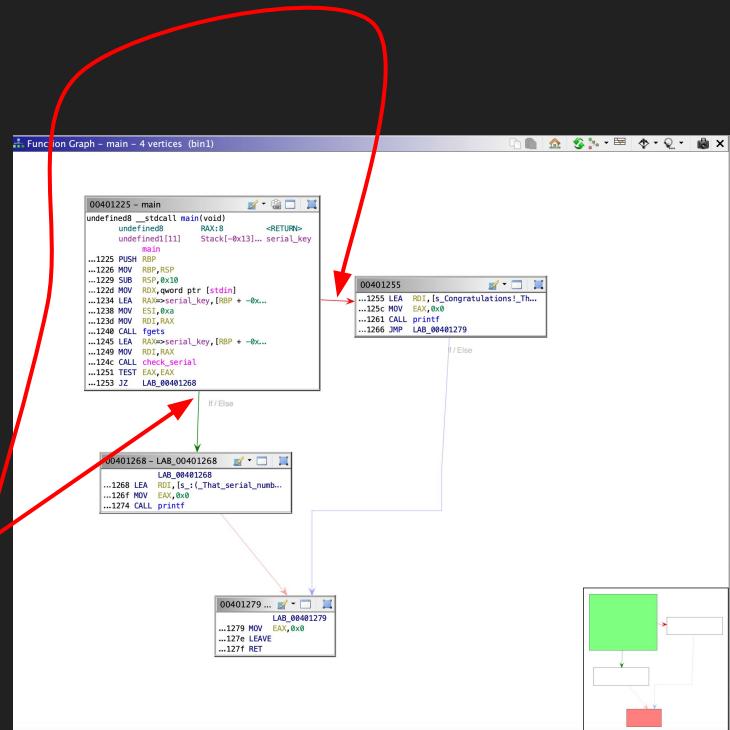
So **nope**, we do NOT want to jump. We wanna skip right over this jump

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



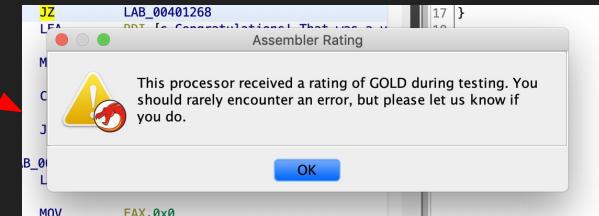
5. Shall we jump? NOP

A screenshot of a debugger interface showing assembly code. The assembly window displays several instructions, including TEST, JZ, MOV, and CALL. A context menu is open over the instruction at address 00401251, which is a TEST instruction. The menu options include: Bookmark..., Clear Code Bytes, Clear With Options..., Clear Flow and Repair..., Copy, Copy Special..., Paste, Comments, Instruction Info..., Modify Instruction Flow..., Patch Instruction, Processor Manual..., and Processor Options... The 'Patch Instruction' option is highlighted with a red arrow pointing from the text above.

```
00401251 85 c0
00401253 74 13
00401255 48 8d 3d
          ac 0d 00 00
0040125c b8 00 00
          00 00
00401261 e8 ca fd
          ff ff
00401266 eb 11
LAB_00401268 48 8d 3d
          c9 0d 00 00
0040126f b8 00 00
          00 00
00401274 e8 b7 fd
          ff ff
00401279 h8 00 00
```

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

```
00401251 85 c0      TEST    EAX, EAX
00401253 74 13      JZ     offset 0040126c
00401255 48 8d 3d    MOV    ECX, ECX
ac 0d 00 00
0040125c b8 00 00    MOV    EAX, 0
00        00 00
00401261 e8 ca fd    JZ     offset 00401253
ff ff
00401266 eb 11      RET
```

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 screenshot shows the Ghidra assembly editor interface. On the left is the assembly view, displaying the following code:

```
00401251 85 c0          TEST    EAX, EAX
00401253 74 13          JNE     loc_0040126D
00401255 48 8d 3d        MOV    ECX, ECX
                           ac 0d 00 00
0040125c b8 00 00        MOV    EAX, 0
                           00 00
00401261 e8 ca fd        JMP    loc_00401253
                           ff ff
00401266 eb 11          RET
```

The instruction at address 00401253, which is a `JNE` (Jump if Not Equal) instruction, has its target address highlighted in red. The assembly editor's status bar at the bottom shows the current assembly line and the assembly offset.

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

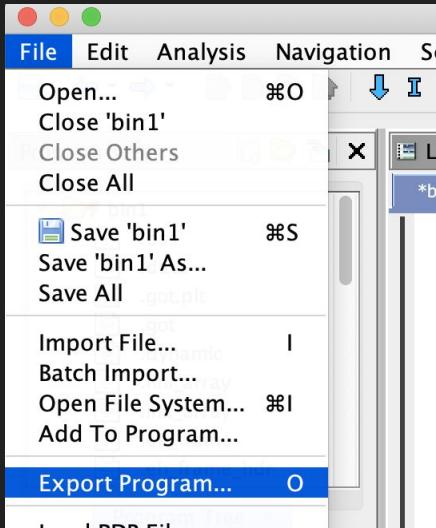
Luckily, Ghidra has given us some choices!

Nowhere to go but success!

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

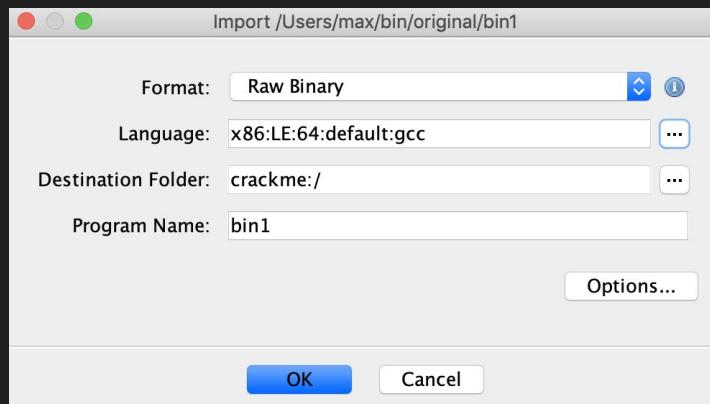
Notice that the decompiled output has now been updated.
The *check_serial* result is no longer checked.

how2export?



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

how2export?



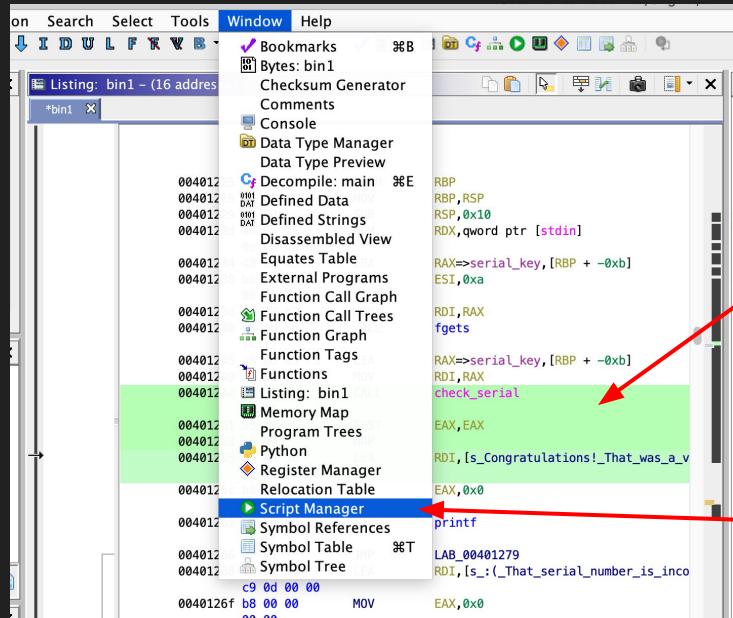
One solution is to not import the file as an ELF but as a raw binary instead.

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

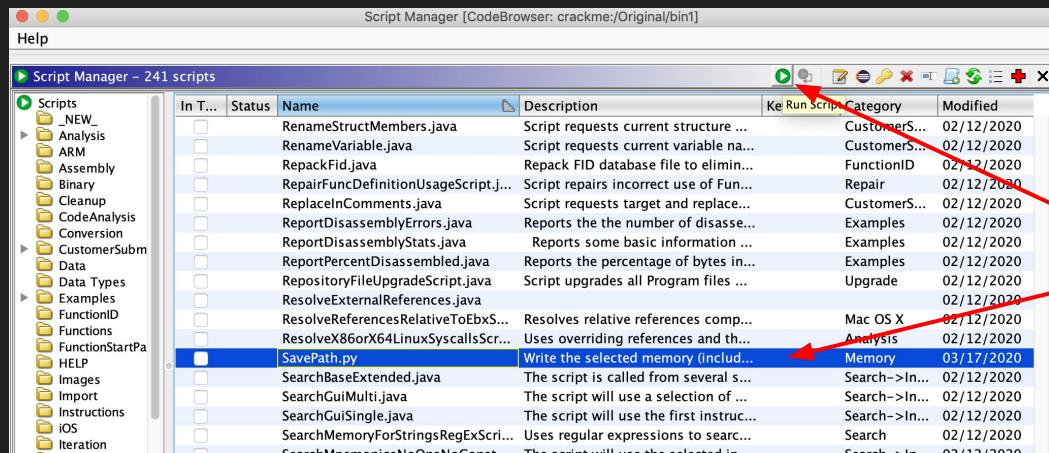
6. how2savepath



Highlight the changed byte area

Go to Window -> Script Manager

6. how2savepath

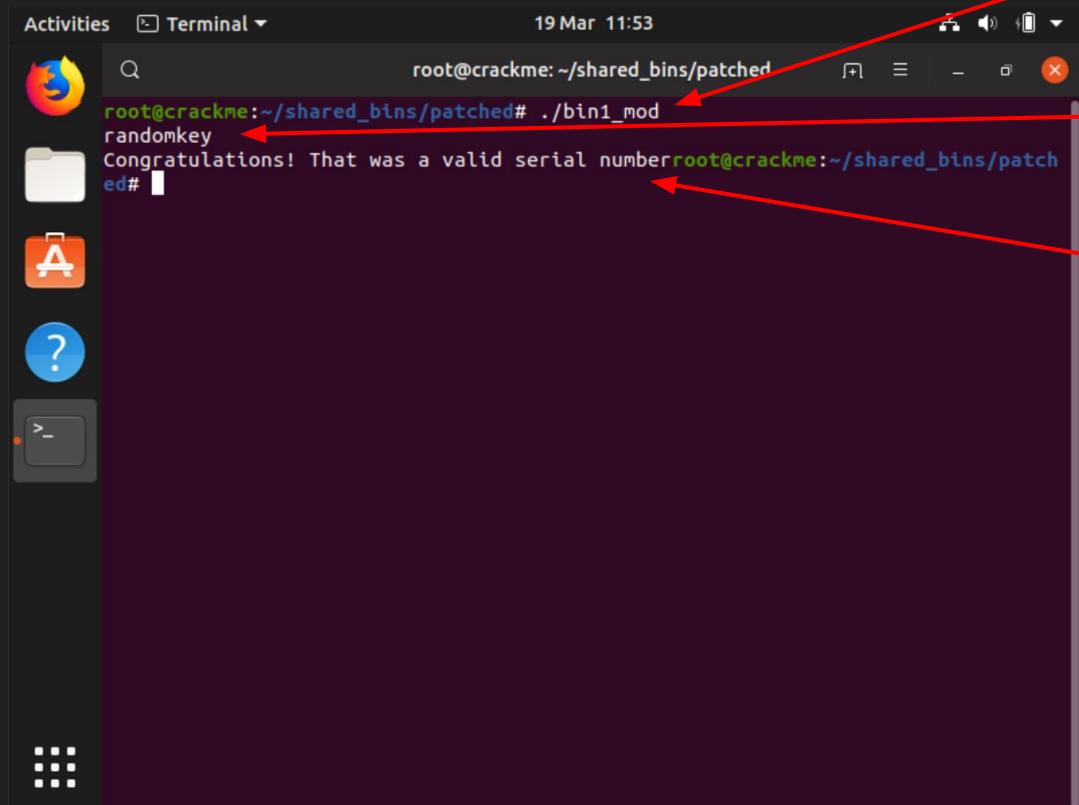


Click the script, click "Run Script"

Save the new binary in the "Patched" folder

7. `winner=$(whoami)"`

Run the modified binary on the testing VM



A screenshot of a Linux desktop environment showing a terminal window. The terminal title is "root@crackme: ~/shared_bins/patched". The terminal content shows:

```
root@crackme:~/shared_bins/patched# ./bin1_mod
randomkey
Congratulations! That was a valid serial numberroot@crackme:~/shared_bins/patch
ed#
```

The terminal window has a dark background and light-colored text. Red arrows point from the text "randomkey" and "Congratulations! That was a valid serial number" to the right, with the following annotations:

- "randomkey" is annotated with "Provide any random key"
- "Congratulations! That was a valid serial number" is annotated with "Insert evil laugh here"

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 screenshot shows the CodeBrowser application interface. On the left, there's a 'Program Trees' panel with a tree view of the binary file structure, including sections like .bss, .data, .got.plt, .got, .dynamic, .fini_array, .init_array, and .eh_frame. Below it is a 'Symbol Tree' panel showing imports, exports, functions, labels, classes, and namespaces. In the center, there are two main windows: 'Listing: bin2' which displays assembly code for the main function, and 'Decompile: main - (bin2)' which shows the corresponding C-like pseudocode. Red arrows point from the text 'It calls fgets, like before' to the fgets call in the pseudocode, and from the text 'It uses the serial key in a function called exec' to the exec call in the pseudocode.

CodeBrowser: crackme:/Original/bin2

File Edit Analysis Navigation Search Select Tools Window Help

Program Trees

bin2

.bss
.data
.got.plt
.got
.dynamic
.fini_array
.init_array
.eh_frame

Symbol Tree

Imports
Exports
Functions
Labels
Classes
Namespaces

Filter:

Data Type Manager

Data Types

BuiltinTypes
bin2
generic_clib_64
windows_vs12_32

Filter:

Listing: bin2

00401380 ff d0 CALL HAX
00401382 48 98 CDQE
LAB_00401384
00401384 c9 LEAVE
00401385 c3 RET

***** FUNCTION *****

undefined undefined main()

AL:1 <RETURN>
Stack[-0x13]:1 local_13

main

00401306 55 PUSH RBP
00401307 48 89 e5 MOV RBP,RSP
00401308 48 83 ec 10 SUB RSP,0x10
00401309 48 b8 15 MOV RDX,qword ptr [stdin]
3b 2e 00 00
00401315 48 8d 45 f5 LEA RAX,local_13,[RBP + -0x00]
00401319 be 00 00 MOV ESI,0xa
00
0040131e 48 89 c7 MOV RDI,RAX
00401321 e8 1a fd CALL fgets
ff ff
00401326 48 8d 45 f5 LEA RAX,local_13,[RBP + -0x00]
0040132a 48 89 c7 MOV RDI,RAX
0040132d e8 00 fe CALL exec
ff ff
00401332 b8 00 00 MOV EAX,0x0
00
00401337 c9 LEAVE

Decompile: main - (bin2)

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

Console – Scripting

00401306 main PUSH RBP

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 long exec(undefined8 param_1)
2 {
3     undefined8 uVar1;
4     int iVar2;
5     undefined8 *puVar3;
6     long lVar4;
7
8     puVar3 = (undefined8 *)mmap((void *)0x0, 0xec, 7, 0x22, -1, 0);
9     uVar1 = shellc0de._8_8;
10    if (puVar3 == (undefined8 *)0xffffffffffffffffff) {
11        lVar4 = 0;
12    }
13    else {
14        *puVar3 = shellc0de._0_8_;
15        puVar3[1] = uVar1;
16        uVar1 = shellc0de._24_8_;
17        puVar3[2] = shellc0de._16_8_;
18        puVar3[3] = uVar1;
19        uVar1 = shellc0de._40_8_;
20        puVar3[4] = shellc0de._32_8_;
21        puVar3[5] = uVar1;
22        uVar1 = shellc0de._56_8_;
23        puVar3[6] = shellc0de._48_8_;
24        puVar3[7] = uVar1;
25        uVar1 = shellc0de._72_8_;
26        puVar3[8] = shellc0de._64_8_;
27        puVar3[9] = uVar1;
28        uVar1 = shellc0de._88_8_;
29        puVar3[10] = shellc0de._80_8_;
30        puVar3[0xb] = uVar1;
31        uVar1 = shellc0de._104_8_;
32        puVar3[0xc] = shellc0de._96_8_;
33        puVar3[0xd] = uVar1;
34        uVar1 = shellc0de._120_8_;
35        puVar3[0xe] = shellc0de._112_8_;
36        puVar3[0xf] = uVar1;
37        uVar1 = shellc0de._136_8_;
38        puVar3[0x10] = shellc0de._128_8_;
39        puVar3[0x11] = uVar1;
40        uVar1 = shellc0de._152_8_;
41        puVar3[0x12] = shellc0de._144_8_;
42        puVar3[0x13] = uVar1;
43        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    iVar2 = (long)iVar2;
61 }
62 return lVar4;
63 }
```

3. Rename variables to make the function clearer

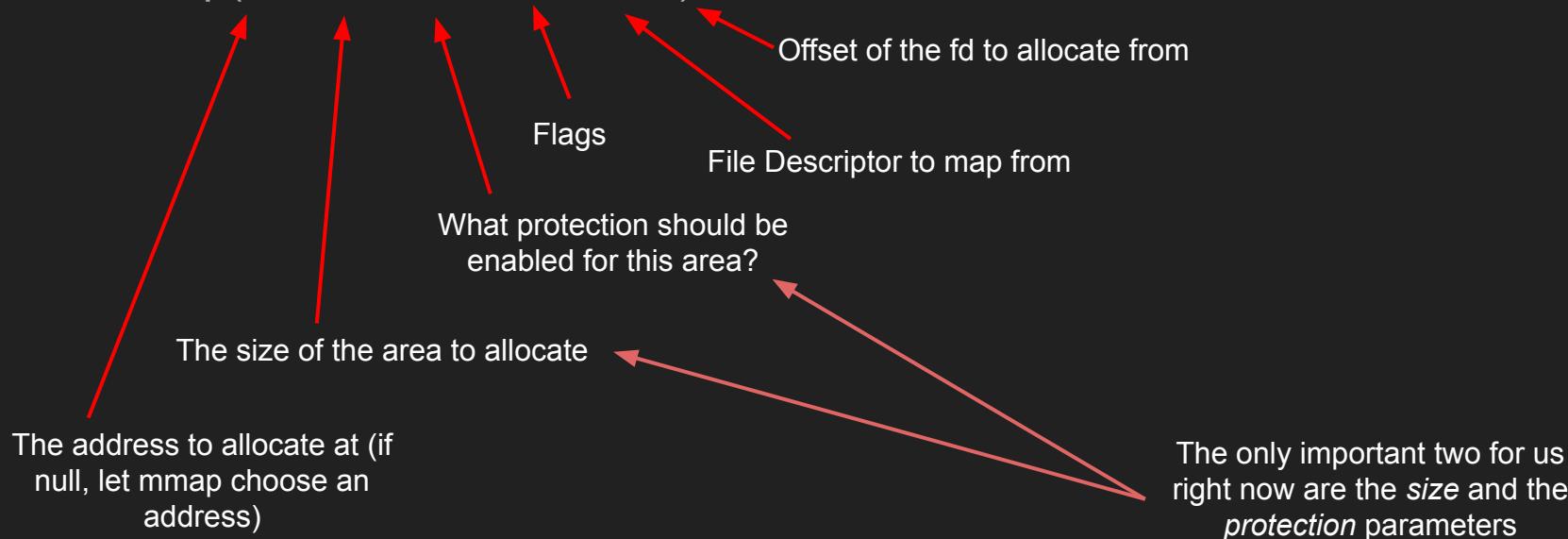
```
function_pointer[0x1b] = uVar1;
function_pointer[0x1c] = shellc0de._224_8_;
*(undefined4 *) (function_pointer + 0x1d) = shellc0de._232_4_;
iVar2 = (*(code *)function_pointer)(serial_key);
lVar3 = (long)iVar2;
}
return lVar3;
```

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.

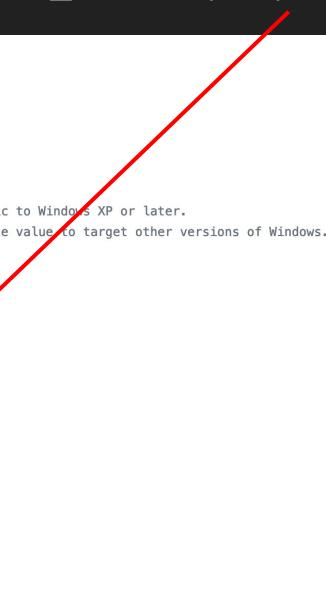
Okay, but what're those parameters?

- `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)



```
1  /*
2   * sys/mman.h
3   * mmman-win32
4   */
5
6  #ifndef _SYS_MMAN_H_
7  #define _SYS_MMAN_H_
8
9  #ifndef _WIN32_WINNT           // Allow use of features specific to Windows XP or later.
10 #define _WIN32_WINNT 0x0501    // Change this to the appropriate value to target other versions of Windows.
11 #endif
12
13 /* All the headers include this file. */
14 #ifndef __MSC_VER
15 #include <_mingw.h>
16 #endif
17
18 #include <sys/types.h>
19
20 #ifdef __cplusplus
21 extern "C" {
22 #endif
23
24 #define PROT_NONE 0
25 #define PROT_READ 1
26 #define PROT_WRITE 2
27 #define PROT_EXEC 4
28
```

So what does it mean overall?

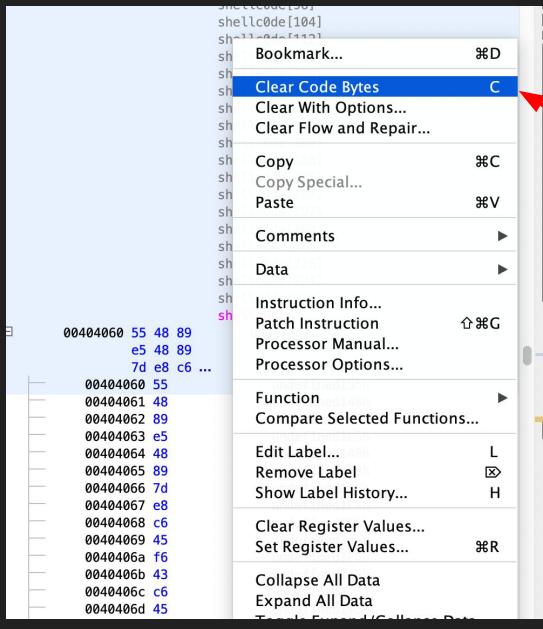
- The mmap function call here is allocating 0xec bytes of data that is readable, writeable, and executable (One super dangerous combination!)
- The contents of *shellcOde* (which we can now assume is 0xec bytes in size) is copied into this allocated area
- Finally, this is executed as a function

4. Double click on the *shellc0de* variable

```
shellc0de[224]
shellc0de[232]
shellc0de
00404060 55 48 89      undefined...
00404060 e5 48 89
00404060 7d e8 c6 ...
00404060 55      undefined155h
00404061 48      undefined148h
00404062 89      undefined189h
00404063 e5      undefined1E5h
00404064 48      undefined148h
00404065 89      undefined189h
00404066 7d      undefined170h
00404067 e8      undefined1E8h
00404068 c6      undefined1C6h
00404069 45      undefined145h
0040406a f6      undefined1F6h
0040406b 43      undefined1A3h
0040406c c6      undefined1C6h
0040406d 45      undefined145h
0040406e f7      undefined1F7h
0040406f 6f      undefined1F6h
00404070 c6      undefined1C6h
00404071 45      undefined145h
00404072 f8      undefined1F8h
00404073 6e      undefined16Eh
00404074 c6      undefined1C6h
00404075 45      undefined145h
00404076 f9      undefined1F9h
00404077 67      undefined167h
00404078 c6      undefined1C6h
00404079 45      undefined145h
```

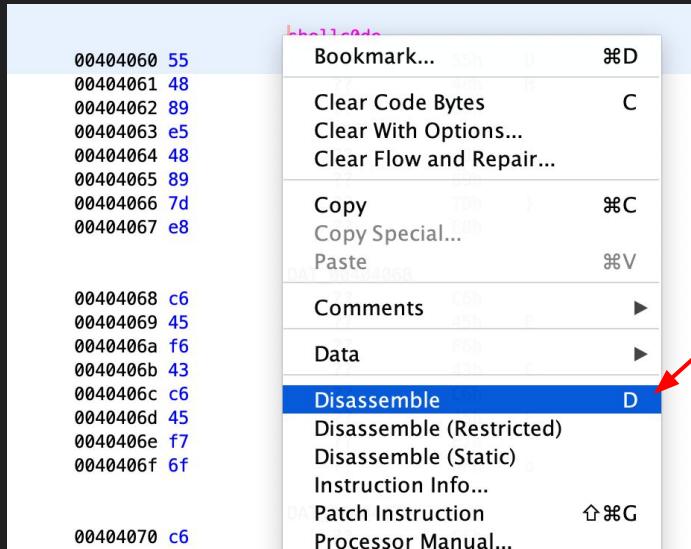
Since this will later be executed, we know that these bytes must be actual x86_64 instructions!

5. Clear code/data bytes



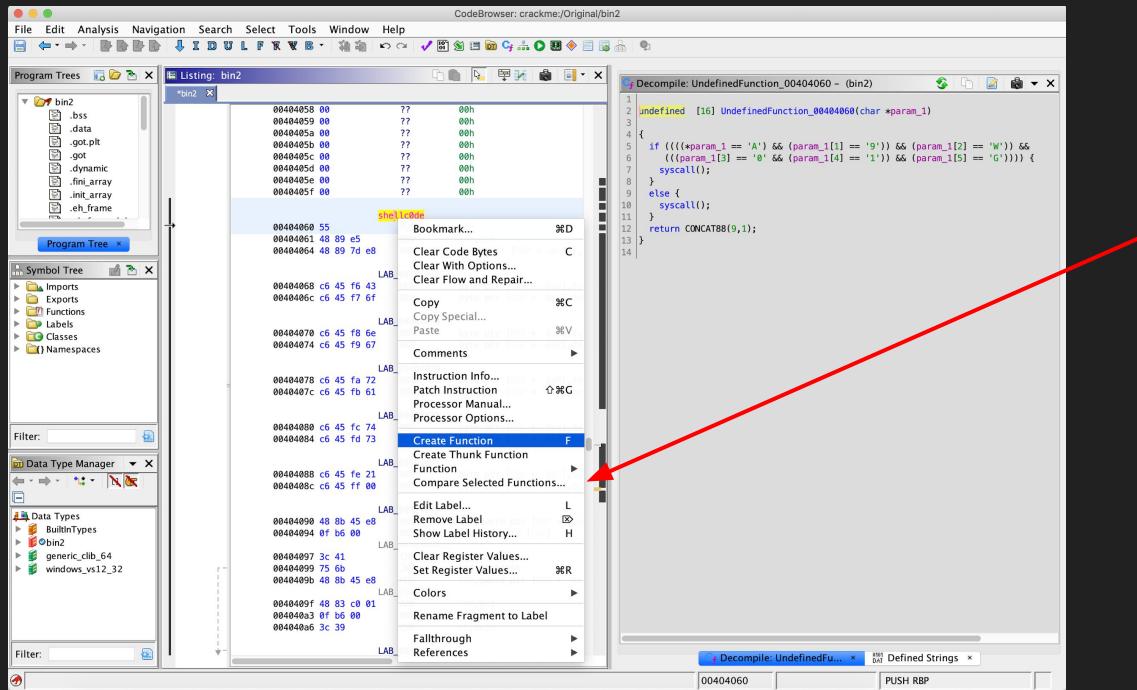
First we need to make these bytes undefined (neither code nor data). Right click *shellc0de* and click “Clear Code Bytes”)

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 undefined [16] shellcode(char *param_1)
2
3 {
4     if ((((*param_1 == 'A') && (param_1[1] == '9')) && (param_1[2] == 'W')) &&
5         (((param_1[3] == '0' && (param_1[4] == '1')) && (param_1[5] == 'G')))) {
6         syscall();
7     }
8     else {
9         syscall();
10    }
11    return CONCAT38(9,1);
12 }
13 }
```

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

```
LAB_004040e6+2
004040e6 48 8d 55 f6      LEA      RDX=>local_12, [RBP + -0xa]
                            LAB_004040ea+6
004040ea 48 c7 c0          MOV      RAX, 0x1
                            01 00 00 00
004040f1 48 c7 c7          MOV      RDI, 0x0
                            00 00 00 00

LAB_004040f8
004040f8 48 89 d6          MOV      RSI, RDX
                            LAB_004040fb+5
004040fb 48 c7 c2          MOV      RDX, 0x9
                            09 00 00 00
00404102 0f 05              SYSCALL
```

```
0040412a 48 8d 55 f6      LEA      RDX=>local_12, [RBP + -0xa]
                            LAB_0040412e+2
0040412e 48 c7 c0          MOV      RAX, 0x1
                            01 00 00 00
00404135 48 c7 c7          MOV      RDI, 0x0
                            00 00 00 00
0040413c 48 89 d6          MOV      RSI, RDX
                            LAB_0040413f+1
0040413f 48 c7 c2          MOV      RDX, 0x9
                            09 00 00 00
00404146 0f 05              SYSCALL
```

Syscall?

```
004040e6 48 8d 55 f6    LEA    RDX=>local_12, [RBP + -0xa]
004040ea 48 c7 c0        MOV    RAX, 0x1
004040f1 48 c7 c7        MOV    RDI, 0x0
004040f8 48 89 d6        MOV    RSI, local_12
004040fb 48 c7 c2        MOV    RDX, 0x9
00404102 0f 05          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!

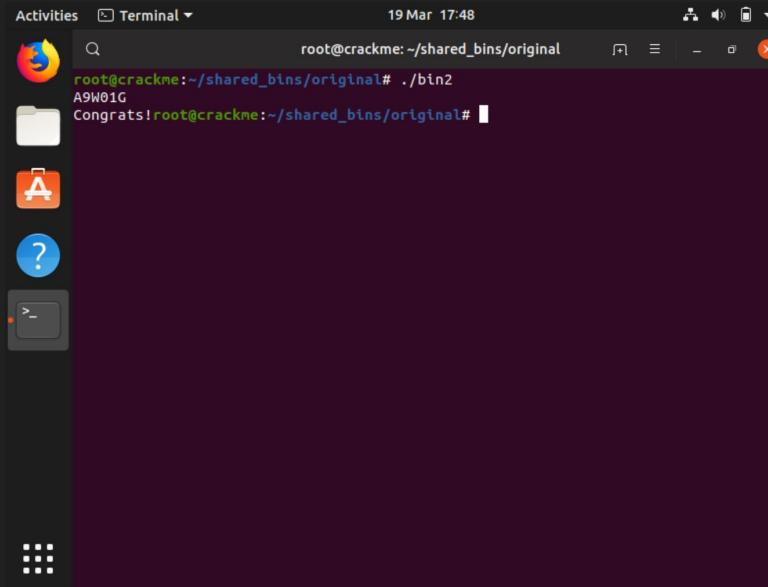
8. Make an assumption!

```
1 undefined [16] shellc0de(char *param_1)
2
3 {
4     if ((((*param_1 == 'A') && (param_1[1] == '9')) && (param_1[2] == 'W')) &&
5         (((param_1[3] == '0' && (param_1[4] == '1')) && (param_1[5] == 'G')))) {
6         syscall();
7     }
8     else {
9         syscall();
10    }
11    return CONCAT88(9,1);
12}
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 undefined8 main(void)
2 {
3     char local_13 [11];
4
5     fgets(local_13,10,stdin);
6     exec(local_13); ←
7     return 0;
8 }
9
10}
11
```

Ah, it's our old friend exec()

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

```
1 long exec(undefined8 param_1)
2 {
3     undefined8 uVar1;
4     int iVar2;
5     undefined8 *puVar3;
6     long lVar4;
7     uint local_c;
8
9     puVar3 = (undefined8 *)mmap((void *)0x0, 0xec, 7, 0x22, -1, 0);
10    if (puVar3 == (undefined8 *)0xffffffffffffffffff) {
11        iVar4 = 0;
12    }
13    else {
14        local_c = 0;
15        while (iVar1 = shellc0de._8_8, local_c < 0xec) {
16            shellc0de[(int)local_c] = shellc0de[(int)local_c] ^ 0xab;
17            local_c = local_c + 1;
18        }
19        *puVar3 = shellc0de._0_8_;
20        puVar3[1] = uVar1;
21        uVar1 = shellc0de._24_8_;
22        puVar3[2] = shellc0de._16_8_;
23        puVar3[3] = uVar1;
24        uVar1 = shellc0de._40_8_;
25        puVar3[4] = shellc0de._32_8_;
26        puVar3[5] = uVar1;
27        uVar1 = shellc0de._56_8_;
28        puVar3[6] = shellc0de._48_8_;
29        puVar3[7] = uVar1;
30        uVar1 = shellc0de._72_8_;
31        puVar3[8] = shellc0de._64_8_;
32        puVar3[9] = uVar1;
33        uVar1 = shellc0de._88_8_;
34        puVar3[10] = shellc0de._80_8_;
35        puVar3[0xb] = uVar1;
36        uVar1 = shellc0de._104_8_;
37        puVar3[0xc] = shellc0de._96_8_;
38        puVar3[0xd] = uVar1;
39        uVar1 = shellc0de._120_8_;
40        puVar3[0xe] = shellc0de._112_8_;
41        puVar3[0xf] = uVar1;
42        uVar1 = shellc0de._136_8_;
43        puVar3[0x10] = shellc0de._128_8_;
44
45 }
```

mmap again

Shellc0de... again...

Function pointer...

```
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 }
```

BUT WAIT!

```

1 long exec(undefined8 param_1)
2 {
3
4     undefined8 uVar1;
5     int iVar2;
6     undefined8 *puVar3;
7     long lVar4;
8     uint local_c;
9
10    puVar3 = (undefined8 *)mmap((void *)0x0,0xec,7,0x22,-1,0);
11    if (puVar3 == (undefined8 *)0xffffffffffff) {
12        iVar4 = 0;
13    }
14    else {
15        local_c = 0;
16        while ((iVar1 = shellc0de._8_8_, local_c < 0xec) {
17            shellc0de[(int)local_c] = shellc0de[(int)local_c] ^ 0xab;
18            local_c = local_c + 1;
19        }
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 (iVar1 = shellc0de._8_8_, local_c < 0xec) {
    shellc0de[(int)local_c] = shellc0de[(int)local_c] ^ 0xab;
}
```

local_c is iterating

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

What is (itself \wedge 0xab) ?

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

$$1 \wedge 0 = 1$$

$$1 \wedge 1 = 0$$

$$0 \wedge 1 = 1$$

$$0 \wedge 0 = 0$$

$$1101 \wedge 1010 = 0111$$

Is the same as $0xd \wedge 0xa = 0x7$

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: $\text{Plaintext} \wedge \text{Key} = \text{Ciphertext}$
- One-Time Pad Decryption: $\text{Ciphertext} \wedge \text{Key} = \text{Plaintext}$
- Figure out the key: $\text{Ciphertext} \wedge \text{Plaintext} = \text{Key}$

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

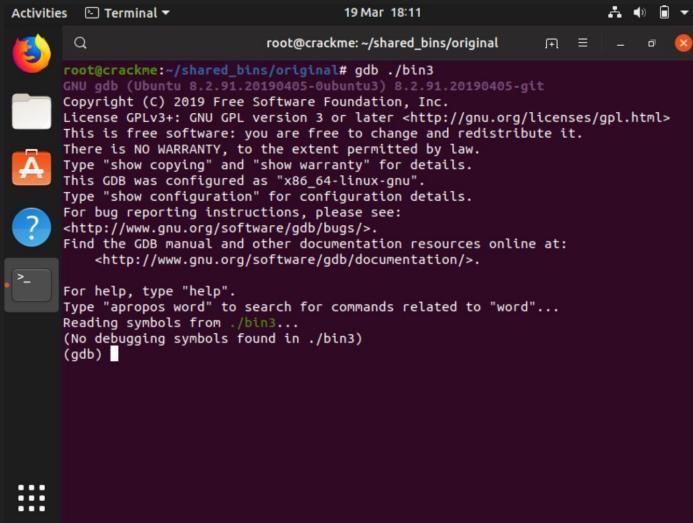
Two options:

- 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



A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window. The terminal window title is "root@crackme:~/shared_bins/original". The terminal content shows the output of running GDB on a binary named "bin3". The output includes the GDB version (8.2.91.20190405-0ubuntu3), copyright information from the Free Software Foundation, and a notice about the GNU GPL license. It also mentions that there is no warranty and provides details about the configuration (x86_64-linux-gnu). The terminal then prompts for help, searching for symbols in the binary, and finally shows the prompt "(gdb)".

```
root@crackme:~/shared_bins/original# gdb ./bin3
GNU gdb (Ubuntu 8.2.91.20190405-0ubuntu3) 8.2.91.20190405-git
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bin3...
(No debugging symbols found in ./bin3)
(gdb)
```

5. Disassemble the exec function

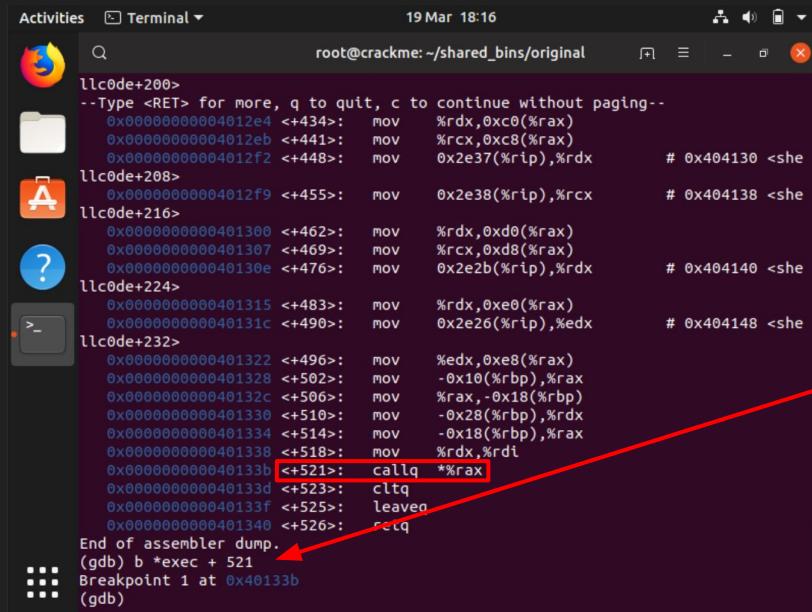
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"....
Reading symbols from ./bin3...
(No debugging symbols found in ./bin3)
(gdb) disas exec
Dump of assembler code for function exec:
0x000000000401132 <0>: push %rbp
0x000000000401133 <+1>: mov %rsp,%rbp
0x000000000401136 <+4>: sub \$0x30,%rsp
0x00000000040113a <+8>: mov %rdi,%rsp
0x00000000040113c <+12>: mov \$0x0,%r9d
0x000000000401140 <+18>: mov \$0xffffffff,%r8d
0x000000000401140 <+24>: mov \$0x22,%ecx
0x000000000401140 <+28>: mov \$0x7,%edx
0x000000000401154 <+34>: mov %ecx,%esi
0x000000000401159 <+39>: mov \$0x0,%edi
0x000000000401159 <+44>: callq *0x401030 < mmap@plt>
0x000000000401163 <+49>: mov %rax,-0x10(%rbp)
0x000000000401167 <+53>: cmpq \$0xfffffffffffffff,-0x10(%rbp)
0x000000000401167 <+58>: jne 0x401178 <exec+70>
0x000000000401167 <+60>: mov \$0x0,%eax
0x000000000401173 <+65>: jmpq 0x4013f3 <exec+525>

Type *disas exec* to disassemble the function

llc0de+192>
0x0000000004012dd <+427>: mov 0x2e44(%rip),%rcx # 0x404128 <she
llc0de+200>
-- Type <RET> for more, q to quit, c to continue without paging--
0x0000000004012e4 <+434>: mov %rdx,0xc0(%rax)
0x0000000004012eb <+441>: mov %rcx,0xc8(%rax)
0x0000000004012f2 <+448>: mov 0x2e37(%rip),%rdx # 0x404130 <she
llc0de+208>
0x0000000004012f9 <+455>: mov 0x2e38(%rip),%rcx # 0x404138 <she
llc0de+216>
0x000000000401300 <+462>: mov %rdx,0xd0(%rax)
0x000000000401307 <+469>: mov %rcx,0xd8(%rax)
0x00000000040130e <+476>: mov 0x2e2b(%rip),%rdx # 0x404140 <she
llc0de+224>
0x000000000401315 <+483>: mov %rdx,0xe0(%rax)
0x00000000040131c <+490>: mov 0x2e26(%rip),%rdx # 0x404148 <she
llc0de+232>
0x000000000401322 <+496>: mov %rdx,0xe8(%rax)
0x000000000401328 <+502>: mov -0x10(%rbp),%rax
0x00000000040132c <+506>: mov %rax,-0x18(%rbp)
0x000000000401330 <+510>: mov -0x28(%rbp),%rdx
0x000000000401334 <+514>: mov %rdx,0x10(%rax)
0x000000000401338 <+518>: mov %rdx,%rdl
0x00000000040133b <+521>: callq *%rax
0x00000000040133d <+523>: cltq
0x00000000040133f <+525>: leaveq
0x000000000401340 <+526>: retq
End of assembler dump.

Press enter until you read the end of the function

6. Break on the call

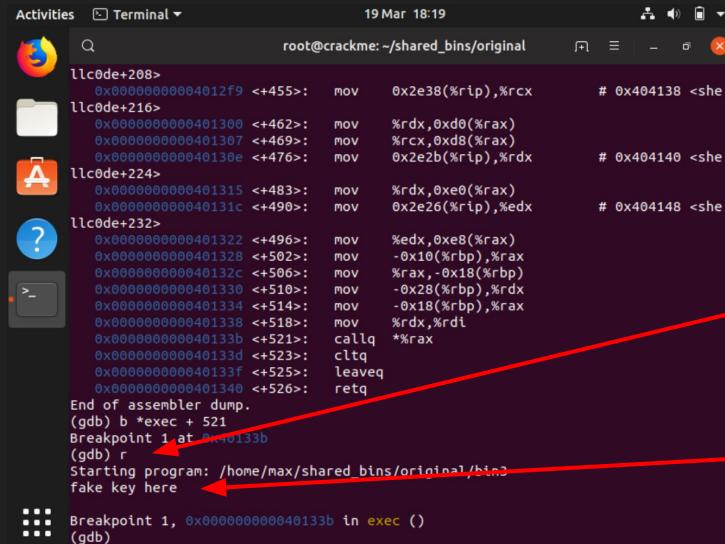


```
Activities Terminal 19 Mar 18:16
root@crackme:~/shared_bins/original
Q
llc0de+200>
--Type <RET> for more, q to quit, c to continue without paging--
0x00000000004012e4 <+434>: mov    %rdx,%c0(%rax)
0x00000000004012eb <+441>: mov    %rcx,%c8(%rax)
0x00000000004012f2 <+448>: mov    0x2e37(%rip),%rdx      # 0x404130 <she
llc0de+208>
0x00000000004012f9 <+455>: mov    0x2e38(%rip),%rcx      # 0x404138 <she
llc0de+216>
0x0000000000401300 <+462>: mov    %rdx,%xd0(%rax)
0x0000000000401307 <+469>: mov    %rcx,%xd8(%rax)
0x000000000040130e <+476>: mov    0xze2b(%rip),%rdx      # 0x404140 <she
llc0de+224>
0x0000000000401315 <+483>: mov    %rdx,%xe0(%rax)
0x000000000040131c <+490>: mov    0x2e26(%rip),%edx      # 0x404148 <she
llc0de+232>
0x0000000000401322 <+496>: mov    %edx,%xe8(%rax)
0x0000000000401328 <+502>: mov    -0x10(%rbp),%rax
0x000000000040132c <+506>: mov    %rax,-0x18(%rbp)
0x0000000000401330 <+510>: mov    -0x28(%rbp),%rdx
0x0000000000401334 <+514>: mov    -0x18(%rbp),%rax
0x0000000000401338 <+518>: mov    %rdx,%rdi
0x000000000040133b <+521>: callq  *%rax
0x000000000040133d <+523>: cltq
0x000000000040133f <+525>: leaveq
0x0000000000401340 <+526>: retq
End of assembler dump.
(gdb) b *exec + 521
Breakpoint 1 at 0x40133b
(gdb)
```

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



A terminal window titled "root@crackme: ~/shared_bins/original" showing assembly code and a GDB session. The assembly code is a dump of the binary. The GDB session shows:

```
l1c0de+208>
0x00000000004012f9 <+455>: mov    0x2e38(%rip),%rcx      # 0x404138 <she
l1c0de+216>
0x0000000000401300 <+462>: mov    %rdx,0xd0(%rax)
0x0000000000401307 <+469>: mov    %rcx,0xd8(%rax)
0x000000000040130e <+476>: mov    0x2e2b(%rip),%rdx      # 0x404140 <she
l1c0de+224>
0x0000000000401315 <+483>: mov    %rdx,0xe0(%rax)
0x000000000040131c <+490>: mov    0x2e26(%rip),%edx      # 0x404148 <she
l1c0de+232>
0x0000000000401322 <+496>: mov    %edx,0xe8(%rax)
0x0000000000401328 <+502>: mov    -0x10(%rbp),%rax
0x000000000040132c <+506>: mov    %rax,-0x18(%rbp)
0x0000000000401330 <+510>: mov    -0x28(%rbp),%rdx
0x0000000000401334 <+514>: mov    -0x18(%rbp),%rax
0x0000000000401338 <+518>: mov    %rdx,%rdi
0x000000000040133b <+521>: callq  *%rax
0x000000000040133d <+523>: cltq
0x000000000040133f <+525>: leaveq
0x0000000000401340 <+526>: retq
End of assembler dump.
(gdb) b *exec + 521
Breakpoint 1 at 0x000000000040133b
(gdb) r
Starting program: /home/max/shared_bins/original/bin3
fake key here
```

Type *r* to run the program

Add a fake key here

8. Get the address of the mmap'd area

The terminal window shows the assembly code for a program, starting with `llc0de+216>`. It includes several `mov` instructions involving registers `%rdx`, `%rcx`, `%rdx`, `%rax`, and `%rdx`. The assembly ends with `End of assembler dump.`. A breakpoint is set at `Breakpoint 1 at 0x40133b`. The command `(gdb) r` is run to start the program. The output shows the program starts at `/home/max/shared_bins/original/bin3` and prompts for a "fake key here". Finally, the command `(gdb) i r rax` is issued, displaying the current value of `rax` as `0x7ffff7fc0d00`.

```
Activities Terminal 19 Mar 18:30
root@crackme:~/shared_bins/original

llc0de+216>
    0x0000000000401300 <+462>: mov    %rdx,%rdx(%rax)
    0x00000000401307 <+469>: mov    %rcx,%d8(%rax)
    0x0000000040130e <+476>: mov    0x2e2b(%rip),%rdx      # 0x404140 <she
llc0de+224>
    0x0000000000401315 <+483>: mov    %rdx,%xe0(%rax)
    0x000000000040131c <+490>: mov    0x2e26(%rip),%edx      # 0x404148 <she
llc0de+232>
    0x0000000000401322 <+496>: mov    %edx,%e8(%rax)
    0x0000000000401328 <+502>: mov    -0x10(%rbp),%rax
    0x000000000040132c <+506>: mov    %rax,-0x18(%rbp)
    0x0000000000401330 <+510>: mov    -0x28(%rbp),%rdx
    0x0000000000401334 <+514>: mov    -0x18(%rbp),%rax
    0x0000000000401338 <+518>: mov    %rdx,%rdi
    0x000000000040133b <+521>: callq  *%rax
    0x000000000040133d <+523>: cltq
    0x000000000040133f <+525>: leaveq
    0x0000000000401340 <+526>: retq

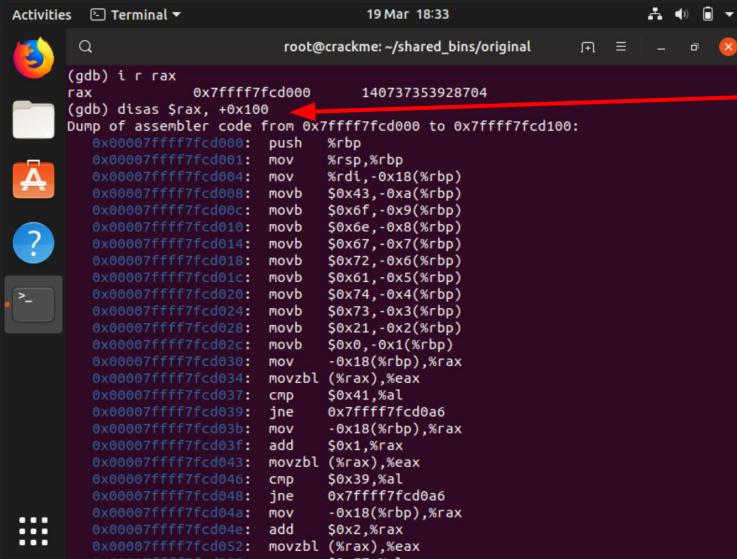
End of assembler dump.
(gdb) b *exec + 521
Breakpoint 1 at 0x40133b
(gdb) r
Starting program: /home/max/shared_bins/original/bin3
fake key here

Breakpoint 1, 0x0000000040133b in exec ()
(gdb) i r rax
rax            0x7ffff7fc0d00  140737353928704
```

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



```
(gdb) i r rax
rax            0x7ffff7fc000      140737353928704
(gdb) disas $rax, +0x100
Dump of assembler code from 0x7ffff7fc000 to 0x7ffff7fc0d00:
0x00007ffff7fc000: push  %rbp
0x00007ffff7fc001: mov   %rsp,%rbp
0x00007ffff7fc004: mov   %rdt,-0x18(%rbp)
0x00007ffff7fc008: movb  $0x43,-0xa(%rbp)
0x00007ffff7fc00c: movb  $0x6f,-0x9(%rbp)
0x00007ffff7fc010: movb  $0x6e,-0x8(%rbp)
0x00007ffff7fc014: movb  $0x67,-0x7(%rbp)
0x00007ffff7fc018: movb  $0x72,-0x6(%rbp)
0x00007ffff7fc01c: movb  $0x61,-0x5(%rbp)
0x00007ffff7fc020: movb  $0x74,-0x4(%rbp)
0x00007ffff7fc024: movb  $0x73,-0x3(%rbp)
0x00007ffff7fc028: movb  $0x21,-0x2(%rbp)
0x00007ffff7fc02c: movb  $0x0,-0x1(%rbp)
0x00007ffff7fc030: mov   -0x18(%rbp),%rax
0x00007ffff7fc034: movzbl (%rax),%eax
0x00007ffff7fc037: cmp   $0x41,%al
0x00007ffff7fc039: jne   0x7ffff7fc0a6
0x00007ffff7fc03b: mov   -0x18(%rbp),%rax
0x00007ffff7fc03f: add   $0x1,%rax
0x00007ffff7fc043: movzbl (%rax),%eax
0x00007ffff7fc046: cmp   $0x39,%al
0x00007ffff7fc048: jne   0x7ffff7fc0a6
0x00007ffff7fc04a: mov   -0x18(%rbp),%rax
0x00007ffff7fc04e: add   $0x2,%rax
0x00007ffff7fc052: movzbl (%rax),%eax
```

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

```
root@crackme:~/shared_bins/original
0x00007ffff7fc0b2: movb  $0x65,-0x7(%rbp)
0x00007ffff7fc0b6: movb  $0x20,-0x6(%rbp)
0x00007ffff7fc0ba: movb  $0x3a,-0x5(%rbp)
--Type <RET> for more, q to quit, c to continue without paging--
0x00007ffff7fc0be: movb  $0x28,-0x4(%rbp)
0x00007ffff7fc0c2: movb  $0xcd0,-0x3(%rbp)
0x00007ffff7fc0c6: movb  $0x20,-0x2(%rbp)
0x00007ffff7fc0ca: lea   -0xa(%rbp),%rdx
0x00007ffff7fc0ce: mov   $0x1,%rax
0x00007ffff7fc0d5: mov   $0x0,%rdi
0x00007ffff7fc0dc: mov   %rdx,%rsi
0x00007ffff7fc0df: mov   $0x9,%rdx
0x00007ffff7fc0e6: syscall
0x00007ffff7fc0e8: nop
0x00007ffff7fc0e9: pop   %rbp
0x00007ffff7fc0ea: retq 
0x00007ffff7fc0eb: stos  %eax,%es:(%rdi)
0x00007ffff7fc0ec: add   %al,(%rax)
0x00007ffff7fc0ee: add   %al,(%rax)
0x00007ffff7fc0f0: add   %al,(%rax)
0x00007ffff7fc0f2: add   %al,(%rax)
0x00007ffff7fc0f4: add   %al,(%rax)
0x00007ffff7fc0f6: add   %al,(%rax)
0x00007ffff7fc0f8: add   %al,(%rax)
0x00007ffff7fc0fa: add   %al,(%rax)
0x00007ffff7fc0fc: add   %al,(%rax)
0x00007ffff7fc0fe: add   %al,(%rax)

End of assembler dump.
(gdb)
```

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

```
root@crackme:~/shared_bins/original
0x00007ffff7fc0d030: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d034: movzbl (%rax),%eax
0x00007ffff7fc0d037: cmp    $0x41,%al
0x00007ffff7fc0d039: jne    0x7ffff7fc0d0a5
0x00007ffff7fc0d03b: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d03f: add    $0x1,%rax
0x00007ffff7fc0d043: movzbl (%rax),%eax
0x00007ffff7fc0d046: cmp    $0x39,%al
0x00007ffff7fc0d048: jne    0x7ffff7fc0d0a6
0x00007ffff7fc0d04a: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d04e: add    $0x2,%rax
0x00007ffff7fc0d052: movzbl (%rax),%eax
0x00007ffff7fc0d055: cmp    $0x57,%al
0x00007ffff7fc0d057: jne    0x7ffff7fc0d0a6
--Type <RET> for more, q to quit, c to continue without paging--
0x00007ffff7fc0d059: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d05d: add    $0x3,%rax
0x00007ffff7fc0d061: movzbl (%rax),%eax
0x00007ffff7fc0d064: cmp    $0x30,%al
0x00007ffff7fc0d066: jne    0x7ffff7fc0d0a6
0x00007ffff7fc0d068: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d06c: add    $0x4,%rax
0x00007ffff7fc0d070: movzbl (%rax),%eax
0x00007ffff7fc0d073: cmp    $0x31,%al
0x00007ffff7fc0d075: jne    0x7ffff7fc0d0a6
0x00007ffff7fc0d077: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d07b: add    $0x5,%rax
0x00007ffff7fc0d07f: movzbl (%rax),%eax
0x00007ffff7fc0d082: cmp    $0x47,%al
```

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

10. Look around

```
root@crackme:~/shared_bins/original
0x00007ffff7fc0d00: mov    -0x18(%rbp),%rax → Get serial[0]
0x00007ffff7fc0d04: movzbl (%rax),%eax → Compare with A
0x00007ffff7fc0d07: cmp    $0x41,%al
0x00007ffff7fc0d09: jne    0x7ffff7fc0d0a6
0x00007ffff7fc0d0b: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d0f: add    $0x1,%rax → Get serial[0 + 1]
0x00007ffff7fc0d43: movzbl (%rax),%eax → Compare with 9
0x00007ffff7fc0d46: cmp    $0x39,%al
0x00007ffff7fc0d48: jne    0x7ffff7fc0d0a6
0x00007ffff7fc0d4a: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d4e: add    $0x2,%rax → Get serial[0 + 2]
0x00007ffff7fc0d52: movzbl (%rax),%eax
0x00007ffff7fc0d55: cmp    $0x57,%al → Compare with W
0x00007ffff7fc0d57: jne    0x7ffff7fc0d0a6
--Type <RET> for more, q to quit, c to continue without paging--
0x00007ffff7fc0d59: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d5d: add    $0x3,%rax → Get serial[0 + 3]
0x00007ffff7fc0d61: movzbl (%rax),%eax
0x00007ffff7fc0d64: cmp    $0x30,%al → Compare with 0
0x00007ffff7fc0d66: jne    0x7ffff7fc0d0a6
0x00007ffff7fc0d68: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d6c: add    $0x4,%rax
0x00007ffff7fc0d70: movzbl (%rax),%eax
0x00007ffff7fc0d73: cmp    $0x31,%al
0x00007ffff7fc0d75: jne    0x7ffff7fc0d0a6
0x00007ffff7fc0d77: mov    -0x18(%rbp),%rax
0x00007ffff7fc0d7b: add    $0x5,%rax
0x00007ffff7fc0d7f: movzbl (%rax),%eax
0x00007ffff7fc0d82: cmp    $0x47,%al
Max-MBP:~ max$ 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

```
(gdb) x/wx 0x00007ffff7fcd0ec  
0x7ffff7fcd0ec: 0x00000000  
(gdb) r ←  
The program being debugged has been started already.  
Start it from the beginning? (y or n) y  
Starting program: /home/max/shared_bins/original/bin3  
A9W01G ←
```

```
Breakpoint 1, 0x000000000040133b in exec ()  
(gdb) c ←  
Continuing.  
Congrats![Inferior 1 (process 24228) exited normally]  
(gdb) █ ←
```

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

Your friendly neighbourhood exec()



2. Same-same but different!

```
1 long exec(undefined8 param_1)
2
3 {
4     undefined8 uVar1;
5     int iVar2;
6     undefined8 *puVar3;
7     long lVar4;
8     uint local_c;
9
10    puVar3 = (undefined8 *)mmap((void *)0x0,0xec,7,0x22,-1,0);
11    if (puVar3 == (undefined8 *)0xffffffffffff) {
12        lVar4 = 0;
13    }
14    else {
15        local_c = 0;
16        while ((uVar1 = shellc0de._8_8_, local_c < 0xec) {
17            shellc0de[(int)local_c] =
18                shellc0de[(int)local_c] ^ *(byte *)((long)&key + (ulong)(local_c & 7));
19            local_c = local_c + 1;
20        }
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 of 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*

The screenshot shows a debugger interface with two panes. The left pane displays assembly code, and the right pane displays C code. A red arrow points from the assembly reference to the C variable.

Assembly (Left Pane):

```
shellc0de[232]
shellc0de
00404060 ab e2 a9      undefined...
          e4 48 76
          5e 70 38 ...
0040414c 00      ??      00h
0040414d 00      ??      00h
0040414e 00      ??      00h
0040414f 00      ??      00h

key
00404150 fe aa 20      undefined8 9823FF000120AAFEh
          01 00 ff
          23 98

// .bss
// SHT_NOBITS [0x404160 - 0x40416f]
// ram: 00404160-0040416f
//
stdin@@GLIBC_2.2.5
```

C Code (Right Pane):

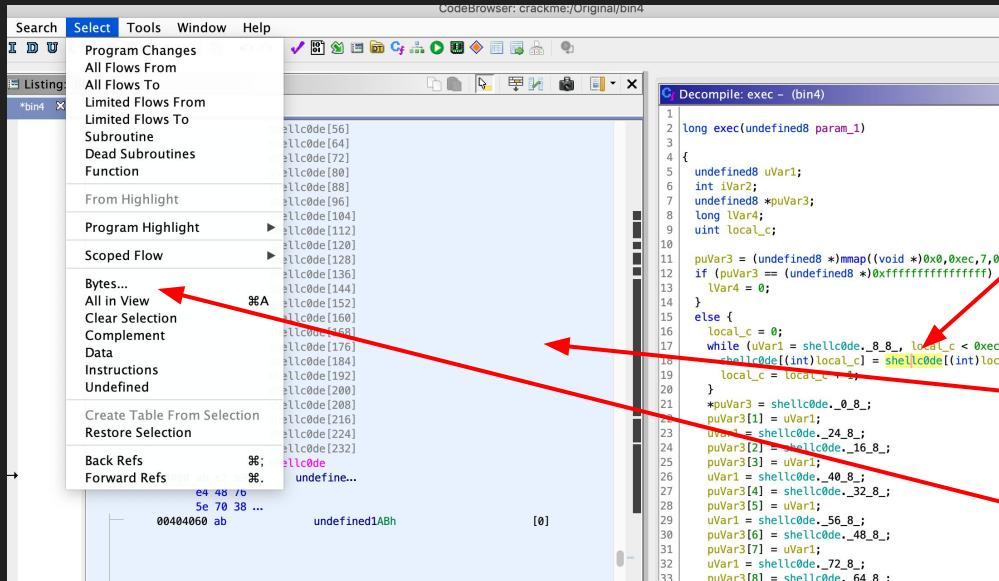
```
14 }
15 else {
16     local_c = 0;
17     while (uVar1 = shellc0de._8_8_, local_c < 0xc) {
18         shellc0de[(int)local_c] =
19             shellc0de[(int)local_c] ^ *(byte *)((long)&key + (ulong)(local_c * 4));
20         local_c = local_c + 1;
21     }
22     *puVar3 = shellc0de._0_8_;
23     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;
30     uVar1 = shellc0de._56_8_;
31     puVar3[6] = shellc0de._48_8_;
32     puVar3[7] = uVar1;
33     uVar1 = shellc0de._72_8_;
34     puVar3[8] = shellc0de._64_8_;
35     puVar3[9] = uVar1;
36     uVar1 = shellc0de._88_8_;
37 }
```

4. Start a Python script with the *key*

```
1  key = [0xfe, 0xaa, 0x20, 0x01, 0x00, 0xff, 0x23, 0x98]  
2  
3 |
```

5. Select the *shellc0de* bytes

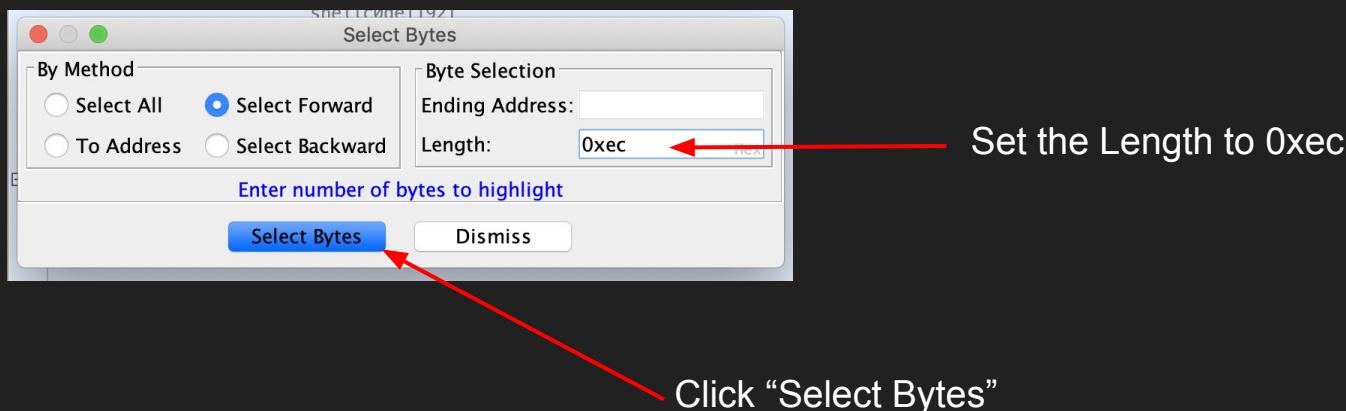
Double-click on *shellc0de*



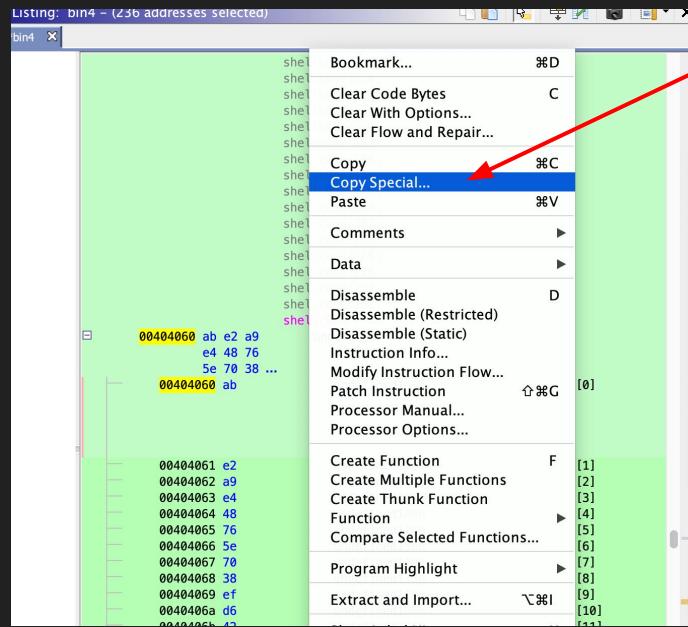
Select it in the disassembly window

Go to Select -> Bytes...

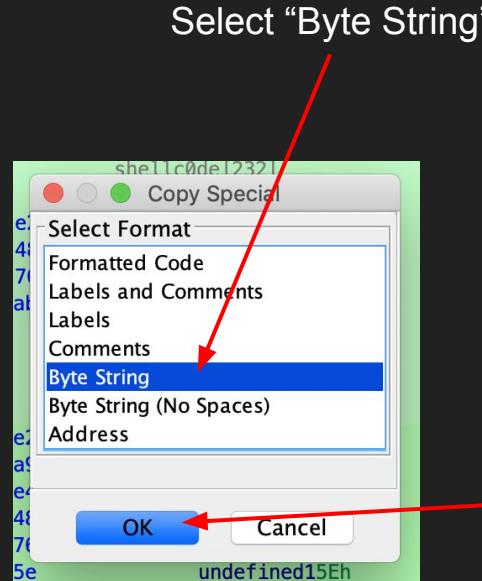
5. Select the *shellc0de* bytes



5. Select the shellc0de bytes



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



Click OK

6. Turn the copied bytes into a Python string

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

```
1 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 28 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:

```
1 \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\xbb\xf6\x68\x8a\x45\x17\x6b\x1b\x3e\x81\x2f\xb7  
\x00\xc3\x74\xed\xb3\xe2\xab\x44\xe8\xb7\x90\x58\xfd\xa5  
\x96\x01\x3c\xcf\x56\xa6\xb6\x21\x65\xe9\x48\x7c\xe3\x9c  
\xf1\xc1\x20\x3d\x31\xba\x0c\xd0\x75\xef\xc8\x91\x83\x3f  
\x26\x97\x48\xaa\x1c\x46\x75\xdf\x6b\x15\xab\x5c\x68\xc6  
\x04\xeb\xbd\xe5\xdd\x08\x4\x6\x4\x4\xf\x7\x9\x0\x5\xdd\x0\x6\x5\xf\x3\xab  
\x20\x01\x00\xb7\x4\x4\x5\xf\xfe\xaa\x20\x01\x4\x8\x7\x6\x5\xf\xd\x0\x3\x9\x8\x9\x0\x0\xf\xf\x2\x9\xd\x6\xe\x7\xe\x3\x0\x0
```

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

```
1 key = [0xfe, 0xaa, 0x20, 0x01, 0x00, 0xff, 0x22, 0x98]
2
3 data = "\xab\xe2\xab\x48\x76\x5e\x70\x38\xef\xd6\x42\xc6\xba\xd4\xf7\x38\xef\xd8\x6f\xc6\x
4
5 def decrypt(_data, _key):
6     output = ""
7     for i in range(len(_data)):
8         output += chr(ord(_data[i]) ^ _key[i % 8])
9     return output
10
11 decrypt(data, key)
12
```

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

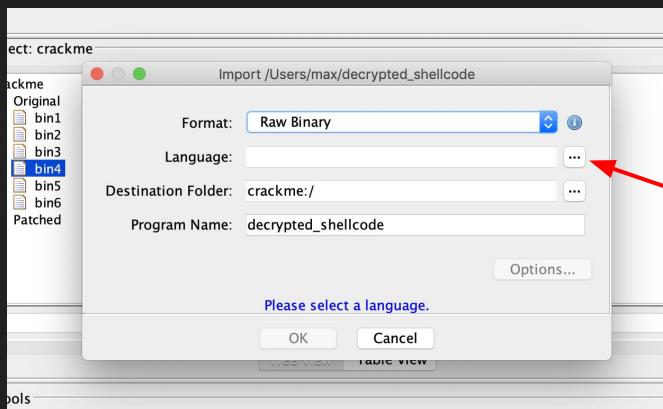
```
>>> decrypt(data, key)
'U\x89\xe8\x9b\x89}\x89}\xe8\x8c\x6E\xf6C\xc6E\xf70\xc6E\xf8n\xc6E\xf9g\xc6E\xfar\xc6E\xfb\x
a\xc6E\xfc\xc6E\xfd\xc6E\xfe!\x1c\x6E\xff\x00H\x8bE\xe8\x0f\xb6\x00<\u0A\x8bE\xe8\x
H\x83\xc0\x01\x0f\xb6\x00<\u0A\x8bE\xe8\x8H\x83\xc0\x02\x0f\xb6\x00<\u0W\x8bE\xe8\x
\x83\xc0\x03\x0f\xb6\x00<\u0A\x8bE\xe8\x8H\x83\xc0\x04\x0f\xb6\x00<\u1\x8bE\xe8\x
H\x83\xc0\x05\x0f\xb6\x00<\u0A\x8bE\xe8\x8H\x83\xc0\x06\x00<\u1\x8bE\xe8\x
\x80\x00H\x89\xd6H\xc7\xc2\xt\x00\x00\x0f\x05\xebB\xc6E\xf6N\xc6E\xf70\xc6E\xf8\x
\xc6E\xf9\xc6E\xfa\xc6E\xfb\xc6E\xfc\xc6E\xfd\xc6E\xfe\x1c\x6E\xff\x00H\x8bE\xe8\x
\x01\x00\x00H\xc7\xc7\x00\x00\x00H\x89\xd6H\xc7\xc2\xt\x00\x00\x0f\x05\xebB\xc6E\xf6N\xc6E\xf70\xc6E\xf8\x
\x05\x90}\x83\xc3\x01'
```

8. Write the decrypted data to a file

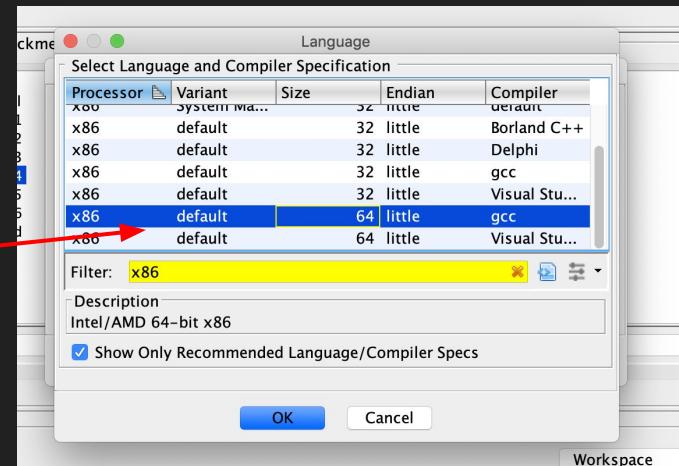
Write Bytes

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

9. Drop the decrypted shellcode into Ghidra



Select the Language
as x86_64



10. Analyze the shellcode file

The screenshot shows two windows from the Immunity Debugger interface. The left window is titled 'Listing: decrypted_shellcode' and displays assembly code for a function named FUN_00000000. The right window is titled 'Decompile: FUN_00000000 - (decrypted_shellcode)' and shows the corresponding Python-like pseudocode.

Assembly (Left):

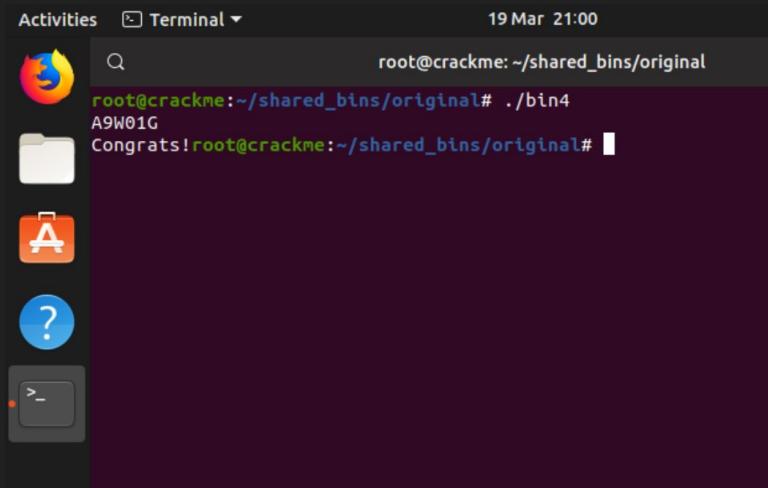
```
00000000 55      PUSH    RBP
00000001 48 89 e5  MOV     RBP, RSP
00000004 48 89 7d e8  MOV     qword ptr [RBP + local_2]
00000008 c6 45 f6 43  MOV     byte ptr [RBP + local_12]
0000000c c6 45 f7 6f  MOV     byte ptr [RBP + local_11]
00000010 c6 45 f8 6e  MOV     byte ptr [RBP + local_10]
00000014 c6 45 f9 67  MOV     byte ptr [RBP + local_f]
00000018 c6 45 fa 72  MOV     byte ptr [RBP + local_e]
0000001c c6 45 fb 61  MOV     byte ptr [RBP + local_d]
00000020 c6 45 fc 74  MOV     byte ptr [RBP + local_c]
00000024 c6 45 fd 73  MOV     byte ptr [RBP + local_b]
00000028 c6 45 ff 21  MOV     byte ptr [RBP + local_a]
0000002c c6 45 ff 00  MOV     byte ptr [RBP + local_g]
00000030 48 8b 45 e8  MOV     RAX, qword ptr [RBP + loc
00000034 0f b6 00  MOVZX  EA, byte ptr [RAX]
00000037 3c 41  CMP    AL, 0x41
00000039 75 6b  JNZ   LAB_000000a6
0000003b 48 8b 45 e8  MOV     RAX, qword ptr [RBP + loc
0000003f 48 83 c0 01  ADD    RAX, 0x1
```

Decompiled Python (Right):

```
1 undefined [16] FUN_00000000(char *param_1)
2
3 {
4     if ((((*param_1 == 'A') && (param_1[1] == '9')) && (param_1[2] == 'W')) &&
5         ((param_1[3] == '0') && (param_1[4] == '1') && (param_1[5] == 'G'))) {
6         syscall();
7     }
8     else {
9         syscall();
10    }
11
12    return CONCAT88(9,1);
13}
14
```

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

11. Enter the key



```
Activities Terminal 19 Mar 21:00
root@crackme:~/shared_bins/original# ./bin4
A9W01G
Congrats! root@crackme:~/shared_bins/original#
```

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

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

Decrypt the shellcode with
a key... again

A different way to copy?

Execute it... again

```
1 long exec(undefined8 param_1)
2 {
3     int iVar1;
4     undefined8 *puVar2;
5     long lVar3;
6     ulong uVar4;
7     undefined8 *puVar5;
8     undefined8 *puVar6;
9     byte bVar7;
10    uint local_c;
11
12    bVar7 = 0;
13    puVar2 = (undefined8 *)mmap((void *)0x0,0x188,7,0x22,-1,0);
14    if (puVar2 == (undefined8 *)0xfffffffffffffff) {
15        lVar3 = 0;
16    }
17    else {
18        local_c = 0;
19        while (local_c < 0x188) {
20            shellc0de[(int)local_c] =
21                shellc0de[(int)local_c] ^ *(byte *)((long)&key + (ulong)(local_c & 7));
22            local_c = local_c + 1;
23        }
24        *puVar2 = shellc0de._0_8_;
25        puVar2[0x30] = shellc0de._384_8_;
26        puVar5 = (undefined8 *)
27            ((long)puVar2 - (long)(undefined8 *)((ulong)(puVar2 + 1) & 0xfffffffffffff8));
28        uVar4 = (ulong)((int)puVar5 + 0x188U >> 3);
29        puVar5 = (undefined8 *)shellc0de + -(long)puVar5;
30        puVar6 = (undefined8 *)shellc0de + (long)(puVar2 + 1) & 0xfffffffffffff8;
31        while (uVar4 != 0) {
32            uVar4 = uVar4 - 1;
33            *puVar6 = *puVar5;
34            puVar5 = puVar5 + (ulong)bVar7 * 0x1fffffffffffffe + 1;
35            puVar6 = puVar6 + (ulong)bVar7 * 0x1fffffffffffffe + 1;
36        }
37        iVar1 = (*(code *)puVar2)(param_1,puVar5,param_1);
38        lVar3 = (long)iVar1;
39    }
40    return lVar3;
41 }
42
43 }
```

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.

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]
00101233	48 89 45 e8	MOV	qword ptr [RBP + local_20], RAX
00101237	48 8b 55 d8	MOV	RDX, qword ptr [RBP + local_30]
0010123b	48 8b 45 e8	MOV	RAX, qword ptr [RBP + local_20]
0010123f	48 89 d7	MOV	RDI, RDX

```
31 puVar5 = (undefined8 *)shellcode + -(long)puVar5;
32 puVar6 = (undefined8 *)((ulong)(puVar2 + 1) & 0xffffffffffff);
33 while (uVar4 != 0) {
34     uVar4 = uVar4 - 1;
35     *puVar6 = *puVar5;
36     puVar5 = puVar5 + (ulong)bVar7 * 0x1fffffffffffffe + 1;
37     puVar6 = puVar6 + (ulong)bVar7 * 0x1fffffffffffffe + 1;
38 }
39 iVar1 = (*(code *)puVar2)(param_1,puVar5,param_1);
40 lVar3 = (long)iVar1;
41 }
```

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

```
001011e7 b9 88 01      MOV    ECX,0x188
001011ec 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,[RSI]
001011f4 48 01 d6      ADD    RSI,RDX
001011f7 48 8d 7e 08      LEA    RDI,[RSI + 0x8]
001011fb 89 ce          MOV    ESI,[RSI]
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,0xffffffff8
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]
00101233 48 89 45 e8      MOV    qword ptr [RBP + local_20],RAX
```

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

```
1  key = [0xfe, 0xaa, 0x20, 0x01, 0x00, 0xff, 0x23, 0x98]
2
3  data = "\xab\xe2\xa9\xe4\x48\x76\x5e\x40\x38\xef\xce\x42\xc6\xba\xcc\x
4
5  def decrypt(_data, _key):
6      output = ""
7      for i in range(len(_data)):
8          output += chr(ord(_data[i]) ^ _key[i % 8])
9      return output
10
11 with open("decrypted_shellcode", 'wb') as f:
12     f.write(decrypt(data, key))
13
```

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  |.F.....F.....|
```

- 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()
AL:1 <RETURN>
Stack[-0xc]:4 local_c

undefined4 Stack[-0x10]:4 local_10

undefined1 Stack[-0x11]:1 local_11

2 undefined [16] FUN_00000000(void)
3
4 {
5     uint local_10;
6     int local_c;
7
8     local_c = 2;
9     local_10 = 0;
10    while ((int)local_10 < 10) {
11        if ((local_10 & 1) != 0) {
12            local_c = local_10 * 2 + local_c % local_10;
13        }
14        if (5 < (int)local_10 % 0xe) {
15            local_c = local_c + local_10 * local_10 + 0x7c;
16        }
17        if ((int)local_10 % 100 < 0x1f) {
18            local_c = local_c - (int)local_10 % 0x18;
19        }
20        local_10 = local_10 + 1;
21    }
22    if ((local_c * local_c) % 0x4e4 == 0x16) {
23        syscall(); // Red arrow points here
24    }
25    else {
26        syscall(); // Red arrow points here
27    }
28    return CONCAT88(9,1);
29 }
```

Another NOPE function

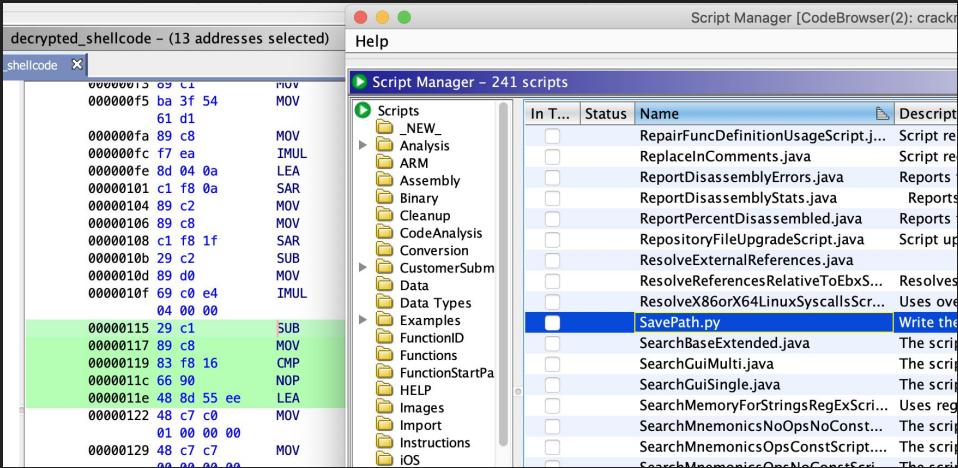
Two syscalls at the bottom

4. Patch the shellcode

000010f	69 c0 e4	IMUL	EAX,EAX, 0x4e4
	04 00 00		
0000115	29 c1	SUB	ECX,EAX
0000117	89 c8	MOV	EAX,ECX
0000119	83 f8 16	CMP	EAX, 0x16
000011c	75 20	NOP	
000011e	48 8d 55 ee		
0000122	48 c7 c0		
	01 00 00 00		
0000129	48 c7 c7	NOP	
	00 00 00 00		
0000130	48 89 d6		NOP/reserved

NOP that conditional jump

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

```
1  key = [0xfe, 0xaa, 0x20, 0x01, 0x00, 0xff, 0x23, 0x98]
2
3  with open("decrypted_shellcode", 'rb') as f:
4      data = f.read()
5
6  def encrypt(_data, _key):
7      output = ""
8      for i in range(len(_data)):
9          output += chr(ord(_data[i]) ^ _key[i % 8])
10     return output
11
12 with open("encrypted_shellcode", 'wb') as f:
13     f.write(encrypt(data, key))
14
```

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

The image shows two hex editors side-by-side. The left hex editor is titled "bin5" and displays the first 17232 bytes of an ELF executable. The right hex editor is titled "encrypted_shellcode" and displays 392 bytes of highly obfuscated shellcode.

bin5 (Left Hex Editor)

Address	Value	Label	Comment
0	7F454C46	ELF	>
24	02010100		
48	00000000		
72	40003800	@	-;
96	00000000	@ 8	@
120	40000000	@	@
144	68020000	h	h
168	03000000	@	@
192	A8020000	@	@
216	00000000	1C000000	1C000000
240	01000000	00000000	00000000
264	00000000	04000000	00000000
288	00000000	00100000	00000000
312	00000000	00000000	00000000
336	00000000	00000000	00000000
360	E83D0000	p	p
384	00000000	E83D0000	É-
408	00000000	00100000	É=
432	F83D0000	=	=
456	00000000	E0010000	‡
480	04000000	E0010000	‡
504	04000000	C4020000	f
528	04020000	C4020000	D

encrypted_shellcode (Right Hex Editor)

Address	Value	Label	Comment
0	ABE2A9E4	ELF	>
24	48765E40		
48	38EFCE42		
72	C6BACCF7		
96	38EFD06F		
120	C6BAD2FF		
144	38EFD273		
168	C6BAD0F9		
192	38EFD475		
216	C6BAD6EB		
240	38EFD620		
264	C6BAD498		
288	39EFDC03		
312	00FF235F		
336	B5B22001		
360	00FFCA07		
384	FEAA208A		

Signed Int le, dec (select some data) Signed Int le, dec (select some data)
0 out of 17232 bytes 124 out of 392 bytes

0x6 - H3X 3D171NG!

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

bin5

	Hex	Text	Find	Replace	Replace All	Replace	Replace & Find	Previous	Next
12384	ABE2A9E4 48765E40 38EFCE42 C6BACCF7 38EFD06F C6BAD2FF	,%Hv^@80€BΔj~80-oΔj“~	ABE2A9E4 48765E40 38EFCE42 C6BACCF7 38EFD06F C6BAD2FF						
12408	38EFD273 C6BAD0F9 38EFD475 C6BAD6EB 38EFD620 C6BAD498	80“sΔj~80‘uΔj÷180+ Δj‘ò	38EFD273 C6BAD0F9 38EFD475 C6BAD6EB 38EFD620 C6BAD498						
12423	39EFDC03 00FF235F BB522001 00FFCA07 FEAA208A 4507A078	90< ^#_“R ^ „™ AE tx	39EFDC03 00FF235F BB522001 00FFCA07 FEAA208A 4507A078						
12456	FF2FE075 11746664 675D5DF9 8BBADB99 3EABF088 4503A8D5	~/fu tfdgj]~äſeö>’äDE @’	FF2FE075 11746664 675D5DF9 8BBADB99 3EABF088 4503A8D5						
12480	0610B325 496DAA50 0940AD05 0A3EDB9B 7768A9C9 C1073CB1	%I”m”P @≠ >€öwh@...i <±	0610B325 496DAA50 0940AD05 0A3EDB9B 7768A9C9 C1073CB1						
12504	3C23F06A C0F10A59 7762A3F9 05812E13 BB522FAE 4507A058	<#äj;Ø Ywb£~ A. “R/ÆE tx	3C23F06A C0F10A59 7762A3F9 05812E13 BB522FAE 4507A058						
12528	82AB65FD 88B2DB22 E12FCB50 8937D472 3F502588 C83ED887	ç‘e“ä≤€”./ÄpA7‘r?%ä>>éá	82AB65FD 88B2DB22 E12FCB50 8937D472 3F502588 C83ED887						
12552	D768A9D1 6B3F47B1 3F23E882 F8E15CBE 75E7D8BB AB5589B2	øh@-k?G±?#Éç~ .\æuÁý“ Uâ≤	D768A9D1 6B3F47B1 3F23E882 F8E15CBE 75E7D8BB AB5589B2						
12576	7762D7EB C1052111 366BD81E 293DA448 FF6A21D1 C11F20B1	wbøÍj ! 6ky)=“H`j!-i ±	7762D7EB C1052111 366BD81E 293DA448 FF6A21D1 C11F20B1						
12600	3F23EA28 5503A0DD 06ABA37C F8F62C16 A955DFFE 8BBADF97	#Í(U tx ‘fI~, ØUf, äſflö	3F23EA28 5503A0DD 06ABA37C F8F62C16 A955DFFE 8BBADF97						
12624	51EFDC88 C1451CCC 9F7BA9C9 F715AE9C F46BD80B 893DAA50	Qô<ä;E Äü{Ø...~ ÄúÙký á=™P	51EFDC88 C1451CCC 9F7BA9C9 F715AE9C F46BD80B 893DAA50						
12648	3F523F28 C276F3F1 3E4E2401 00D6E211 3629D817 75DF6B15	?R?(-vÜÖ-N\$ +, 6)ý ufk	3F523F28 C276F3F1 3E4E2401 00D6E211 3629D817 75DF6B15						
12672	AB4468C6 C0FE2398 FEE2E7C6 00FF2398 B623F649 C73D2A98	‘DhΔz,.#ò,, Áð “#òð#“ I=>ò	AB4468C6 C0FE2398 FEE2E7C6 00FF2398 B623F649 C73D2A98						
12696	FEAA2F04 EBB9E5DD 10E4E644 EF90E5DD 0EDAE644 F19AE5DD	”/ ÍπÀ> %ÉDöéÀ, /ÉDööÀ,	FEAA2F04 EBB9E5DD 10E4E644 EF90E5DD 0EDAE644 F19AE5DD						
12720	0C8AE644 F3C5E5DD 0A82E644 F5DFE5DD 088AE644 F7FF6B15	äÉDÜ~À, ÇÉDíflÀ, äÉD~“k	0C8AE644 F3C5E5DD 0A82E644 F5DFE5DD 088AE644 F7FF6B15						
12744	AB4468C6 C0FE2398 FEE2E7C6 00FF2398 B623F649 C73D2A98	‘DhΔz,.#ò,, Áð “#òð#“ I=>ò	AB4468C6 C0FE2398 FEE2E7C6 00FF2398 B623F649 C73D2A98						
12768	FEAA2F04 90A2E000 FEAA2001 00FF2398 474343A 20284465	”/ è‡‡ „™ “#òGCC: (De	FEAA2F04 90A2E000 FEAA2001 00FF2398 474343A 20284465						

Signed Int (le, dec) (select less data)

392 bytes selected at offset 12384 out of 17232 bytes

encrypted_shellcode

	Signed Int	(le, dec)	(select some data)
0	ABE2A9E4 48765E40 38EFCE42 C6BACCF7 38EFD06F C6BAD2FF		
24	38EFD273 C6BAD0F9 38EFD475 C6BAD6EB 38EFD620 C6BAD498		
48	39EFDC03 00FF235F BB522001 00FFCA07 FEAA208A 4507A078		
72	FF2FE075 11746664 675D5DF9 8BBADB99 3EABF088 4503A8D5		
96	0610B325 496DAA50 0940AD05 0A3EDB9B 7768A9C9 C1073CB1		
120	3C23F06A C0F10A59 7762A3F9 05812E13 BB522FAE 4507A058		
144	82AB65FD 88B2DB22 E12FCB50 8937D472 3F502588 C83EDB87		
168	D768A9D1 6B3F47B1 3F23E882 F8E15CBE 75E7D8BB AB5589B2		
192	7762D7EB C1052111 366BD81E 293DA448 FF6A21D1 C11F20B1		
216	3F23EA28 5503A0DD 06ABA37C F8F62C16 A955DFFE 8BBADF97		
240	51EFDC88 C1451CCC 9F7BA9C9 F715AE9C F46BD80B 893DAA50		
264	3F523F28 C276F3F1 3E4E2401 00D6E211 3629D817 766F6B15		
288	AB4468C6 C0FE2398 FEE2E7C6 00FF2398 B623F649 C73D2A98		
312	FEAA2F04 EBB9E5DD 10E4E644 EF90E5DD 0EDAE644 F19AE5DD		
336	0C8AE644 F3C5E5DD 0A82E644 F5DFE5DD 088AE644 F7FF6B15		
360	AB4468C6 C0FE2398 FEE2E7C6 00FF2398 B623F649 C73D2A98		
384	FEAA2F04 90A2E000		

Signed Int (le, dec) (select some data)

124 out of 392 bytes

0x6 - H3X 3D171NG!

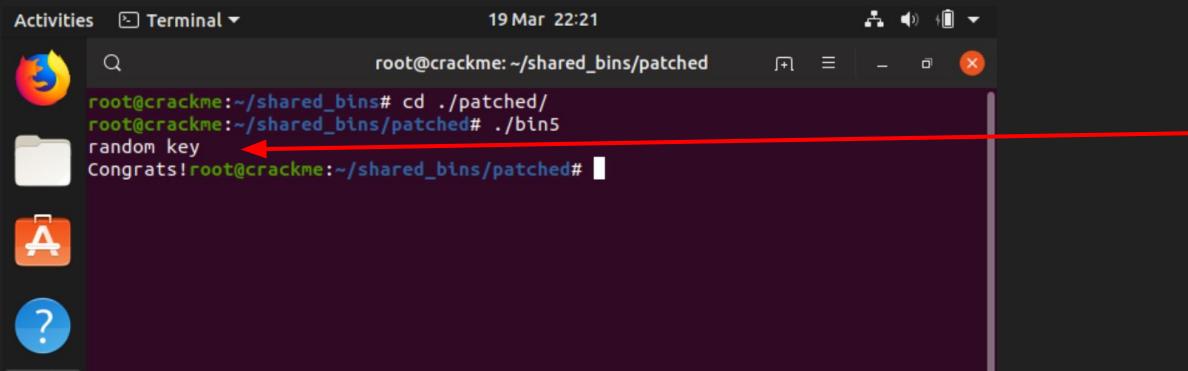
- Replace the shellcode in bin5 with the encrypted_shellcode file contents (and save)

The image shows two hex editors side-by-side. The left hex editor is titled "bin5" and has a search bar containing "AB 2E A9 E4". The right hex editor is titled "encrypted_shellcode" and also has a search bar containing "AB 2E A9 E4". Both editors are displaying the same assembly dump of the program, showing various memory addresses and assembly instructions. The assembly dump includes labels like ABE2A9E4, 38EFCE42, C6BACCF7, 38EF06F, C6BAD2FF, etc., and instructions involving registers like R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, and memory locations like 00FF235F, BB522001, 00FFCA07, FEAA208A, 4507A078, 0610B325, 496DA50, 0940AD05, 0A3EDB9B, 7768A9C9, C1073CB1, 3C23F06A, C0F10A59, 7762A3F9, 05812E13, BB522FAE, 4507A058, 82AB65FD, 8BB2DB22, E12FCB50, 8937D472, 3F502588, C83EDB87, D768A9D1, 6B3F47B1, 3F23E882, F8E15CBE, 75E7D8BB, AB5589B2, 7762D7EB, C1052111, 366BD81E, 293DAA48, FF6A21D1, C11F20B1, 3F23EA28, 5503A0DD, 06ABA37C, F8F62C16, A955DFFE, 88BADF97, 51EFDCC8, C1451CCC, 9F7BA9C9, F715AE9C, F46BD80B, 893DA050, 3F523F28, C276F3F1, 3E4E2401, 00D6E211, 3629D817, 666F6B15, AB4468C6, C0FE2398, FEE2E7C6, 00FF2398, B623F649, C73D2A98, FEAA2F04, EBB9E5DD, 10E4E644, EF90E5DD, 0EDEA644, F19AE5DD, 0C8AE644, F3C5E5DD, 0A82E644, F5DFE5DD, 088AE644, F7FF6B15, AB4468C6, C0FE2398, FEE2E7C6, 00FF2398, B623F649, C73D2A98, FEAA2F04, 90A2E000, FEAA2001, 00FF2398, 4743433A, 20284465, etc.

Signed Int le, dec (select some data) 12776 out of 17232 bytes

Signed Int le, dec (select less data) 392 bytes selected at offset 0 out of 392 bytes

7. Run the patched bin5 binary in the VM



A screenshot of a Linux desktop environment showing a terminal window. The terminal window title is "root@crackme: ~/shared_bins/patched". The terminal content shows the following command sequence:

```
root@crackme:~/shared_bins/patched# cd ./patched/  
root@crackme:~/shared_bins/patched# ./bin5  
random key  
Congrats!
```

A red arrow points from the word "random" in the terminal output to the text "All license keys are now accepted!" located to the right of the terminal window.

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...

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

mmap 0x4e bytes

Some decryption routine

...an strncmp?

...a memcpy?!

The infamous function call

```
1 long exec(long param_1)
2 {
3     int iVar1;
4     code * __dest;
5     long lVar2;
6     uint local_c;
7
8     _dest = (code *)mmap((void *)0x0,0x4e,7,0x22,-1,0);
9     if (_dest == (code *)0xfffffffffffffff) {
10         perror("mmap");
11         lVar2 = 0;
12     }
13     else {
14         local_c = 0;
15         while (local_c < 0x58) {
16             shellc0de[(int)local_c] = shellc0de[(int)local_c] ^ *(byte *) (param_1 + (int)local_c % 10);
17             local_c = local_c + 1;
18         }
19         iVar1 = strncmp(shellc0de,$DAT_00402009,10);
20         if (iVar1 == 0) {
21             memcpy(_dest,shellc0de + 10,0x4e);
22             iVar1 = (*_dest)(param_1);
23             lVar2 = (long) iVar1;
24         }
25     }
26     else {
27         puts("Failed.");
28         lVar2 = 0;
29     }
30 }
31 return lVar2;
32 }
```

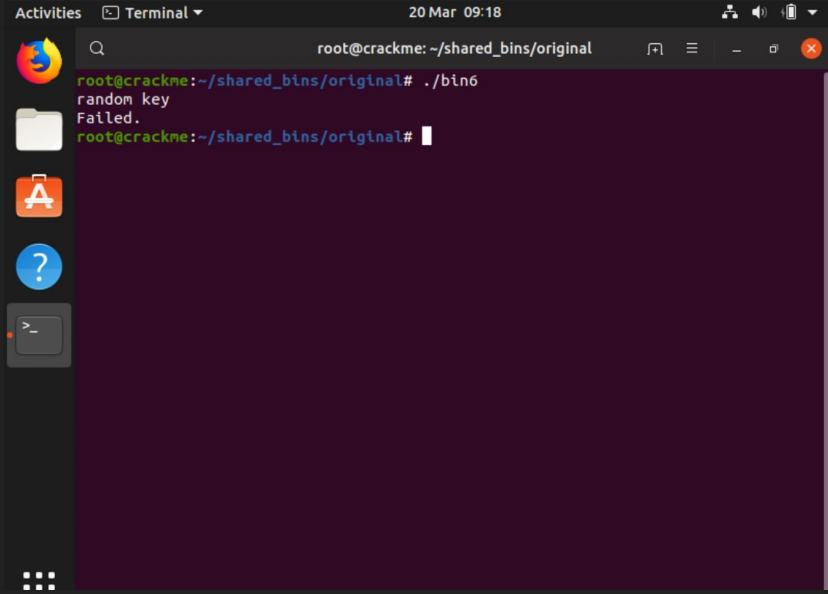
2. Magic Decryption

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?



A screenshot of a Linux desktop environment. On the left is a dock with icons for a browser (Firefox), file manager (Nautilus), text editor (gedit), help (man), and terminal (Konsole). The main area shows a terminal window titled "root@crackme: ~/shared_bins/original". The terminal output is:

```
root@crackme:~/shared_bins/original# ./bin6
random key
Failed.
root@crackme:~/shared_bins/original#
```

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

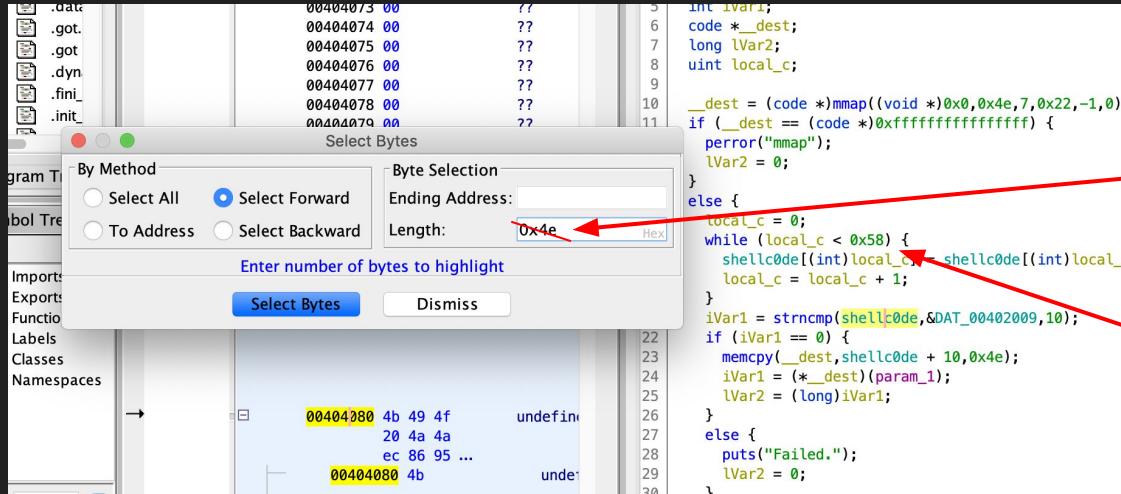
```
iVar1 = strncmp(shellc0de,&DAT_00402009,10);
if (iVar1 == 0) {
    memcpy(__dest,shellc0de + 10,0x4e);
    iVar1 = (*__dest)(param_1);
    lVar2 = (long)iVar1;
}
else {
    puts("Failed.");
    lVar2 = 0;
}
```

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

4. Decrypting *shellc0de*

- Back to those XOR properties!
- $\text{shellc0de}[0\ldots 9] \wedge \text{serial_key} = \text{magic_bytes}$
- Therefore: $\text{shellc0de}[0\ldots 9] \wedge \text{magic_bytes} = \text{serial_key}$

Careful!



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

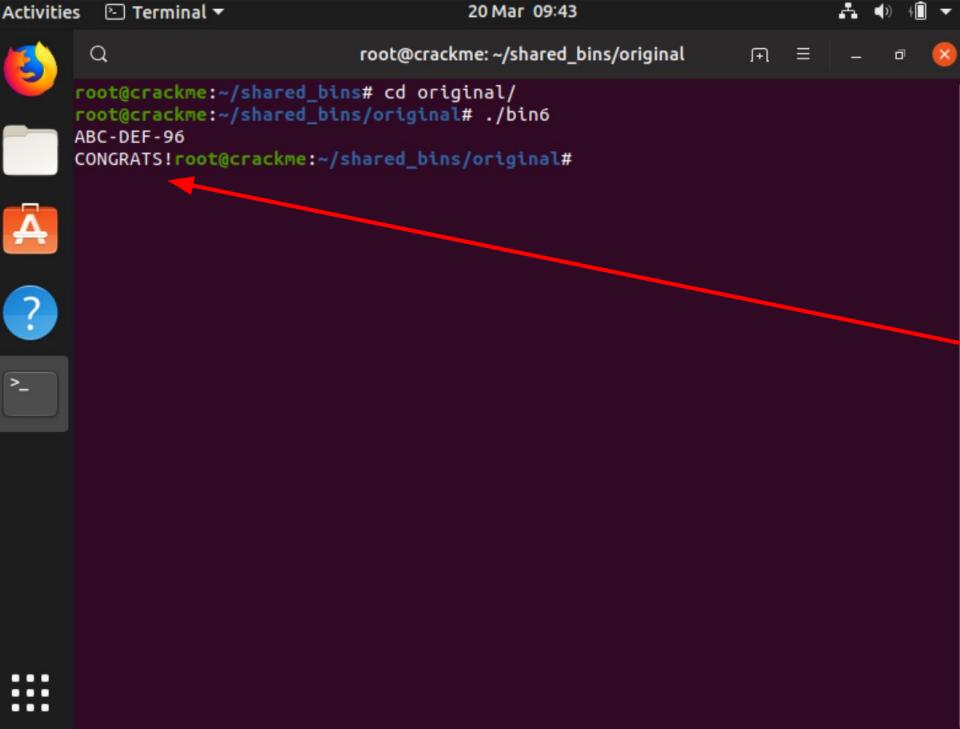
Don’t forget about the
magic bytes! It’s really 0x4e
+ 10 (0x58)

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\x
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]
12
```

```
>>> 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!



```
Activities Terminal 20 Mar 09:43
root@crackme:~/shared_bins/original
root@crackme:~/shared_bins/original# cd original/
root@crackme:~/shared_bins/original# ./bin6
ABC-DEF-96
CONGRATS! root@crackme:~/shared_bins/original#
```

A red arrow points from the text "A much more enthusiastic" in the adjacent block to the word "CONGRATS!" in the terminal output.

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!