Defeating macOS Malware Anti-Analysis Tricks with Radare2

sentinelone.com/labs/defeating-macos-malware-anti-analysis-tricks-with-radare2/

Phil Stokes

In this second post in our series on intermediate to advanced macOS malware reversing, we start our journey into tackling common challenges when dealing with macOS malware samples. Last time out, we took a look at how to use radare2 for rapid triage, and we'll continue using r2 as we move through these various challenges. Along the way, we'll pick up tips on both how to beat obstacles put in place by malware authors and how to use r2 more productively.

Although we can achieve a lot from static analysis, sometimes it can be more efficient to execute the malware in a controlled environment and conduct dynamic analysis. Malware authors, however, may have other ideas and can set up various roadblocks to stop us doing exactly that. Consequently, one of the first challenges we often have to overcome is working around these attempts to prevent execution in our safe environment.

In this post, we'll look at how to circumvent the malware author's control flow to avoid executing unwanted parts of their code, learning along the way how to take advantage of some nice features of the r2 debugger! We'll be looking at a sample of EvilQuest (password: infect3d), so fire up your VM and download it before reading on.

A note for the unwary: if you're using Safari in your VM to download the file and you see "decompression failed", go to Safari Preferences and turn off the 'Open "safe" files after downloading' option in the General tab and try the download again.

Getting Started With the radare2 Debugger

Our sample hit the headlines in July 2020, largely because at first glance it appeared to be a rare example of macOS ransomware. SentinelLabs quickly analyzed it and produced a decryptor to help any potential victims, but it turned out the malware was not very effective in the wild.

It may well have been a PoC, or a project still in early development stages, as the code and functionality have the look and feel of someone experimenting with how to achieve various attacker objectives. However, that's all good news for us, as EvilQuest implements several anti-analysis features that will serve us as good practice.

The first thing you will want to do is remove any extended attributes and codesigning if the sample has a revoked signature. In this case, the sample isn't signed at all, but if it were we could use:

```
% sudo codesign --remove-signature <path to bundle or file>
```

If we need the sample to be codesigned for execution, we can also sign it (remember your VM needs to have installed the Xcode command line tools via `xcode-select --install`) with:

```
% sudo codesign -fs - <path to bundle or file> --deep
```

We'll remove the extended attributes to bypass Gatekeeper and Notarization checks with

```
% xattr -rc <path to bundle or file>
```

And we'll attempt to attach to the radare2 debugger by adding the `-d` switch to our initialization command:

```
% r2 -AA -d patch
```

Unfortunately, our first attempt doesn't go well. We already removed the extended attributes and codesigning isn't the issue here, but the radare2 debugger fails to attach.

```
auser@reversing-lab-10 ~ % cd ~/Downloads/EvilQuest
auser@reversing-lab-10 EvilQuest % ls -al
total 21440
drwxr-xr-x@  5 auser  staff       160 30 Jun  2020 .
drwx------@ 20 auser  staff       640 20 Sep 15:02 ..
-rw-r--r--@  1 auser  staff  10880309 30 Jun  2020 Mixed In Key 8.dmg
-rwxr-xr-x@  1 auser  admin     87920 27 Jun  2020 patch
-rw-r--r--@  1 auser  staff       208 30 Jun  2020 readme.txt
auser@reversing-lab-10 EvilQuest % shasum patch
efbb681a61967e6f5a811f8649ec26efe16f50ae  patch
auser@reversing-lab-10 EvilQuest % r2 -AA -d patch
Child killed
unknown error in debug_attach
Child killed
ptrace: Cannot attach: Invalid argument
Possibly unsigned r2. Please see doc/macos.md
ERRNO: 22 (EINVAL)
[w] Cannot open 'dbg://./patch' for writing.
auser@reversing-lab-10 EvilQuest %
```

Failing to attach the debugger.

That `ptrace: Cannot Attach: Invalid argument` looks ominous, but actually the error message is misleading. The problem is that we need elevated privileges to debug, so a simple `sudo` should get us past our current obstacle.

```
auser@reversing-lab-10 EvilQuest % sudo r2 -AA -d patch
Password:
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references (aao)
[x] Finding and parsing C++ vtables (avrr)
[x] Skipping type matching analysis in debugger mode (aaft)
[x] Propagate noreturn information (aanr)
[x] Finding function preludes
[x] Enable constraint types analysis for variables
 -- Helping siol merge? No way, that would be like.. way too much not lazy. - vi
fino
[0x112ae0000]>
```

The debugger needs elevated privileges

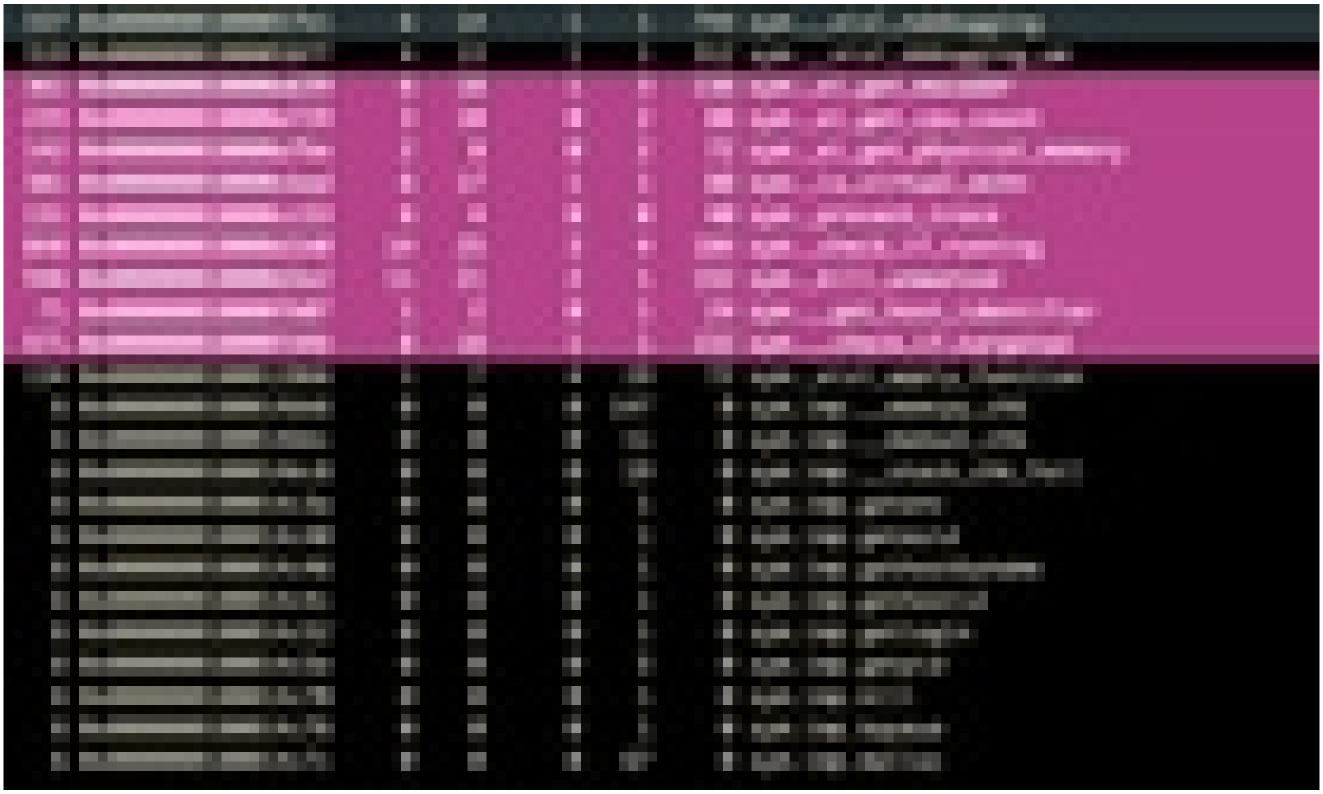
Yay, attach success! Let's take a look around before we start diving further into the debugger.

A Faster Way of Finding XREFS and Interesting Code

Let's run `afll` as we did when analyzing OSX.Calisto previously, but this time we'll output the function list to file so that we can sort it and search it more conveniently without having to keep running the command or scrolling up in the Terminal window.

```
> afll > functions.txt
```

Looking through our text file, we can see there are a number of function names that could be related to some kind of anti-analysis.



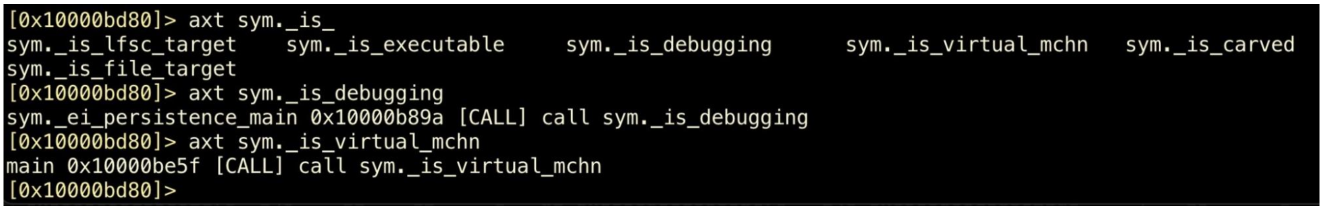
Some of EvilQuest's suspected anti-analysis functions

We can see that some of these only have a single cross-reference, and if we dig into these using the `axt` commmand, we see the cross-reference (XREF) for the `is_virtual_mchn` function happens to be `main()`, so that looks a good place to start.

```
:> axt?
Usage: axt[?gq*]  find data/code references to this address
| axtj [addr]  find data/code references to this address and print in json format
| axtg [addr]  display commands to generate graphs according to the xrefs
| axtq [addr]  find and list the data/code references in quiet mode
| axt* [addr]  same as axt, but prints as r2 commands
:>
```

Getting help on radare2's `axt` command

```
> axt sym._is_debugging
main 0x10000be5f [CALL] sys._is_virtual_mchn
```

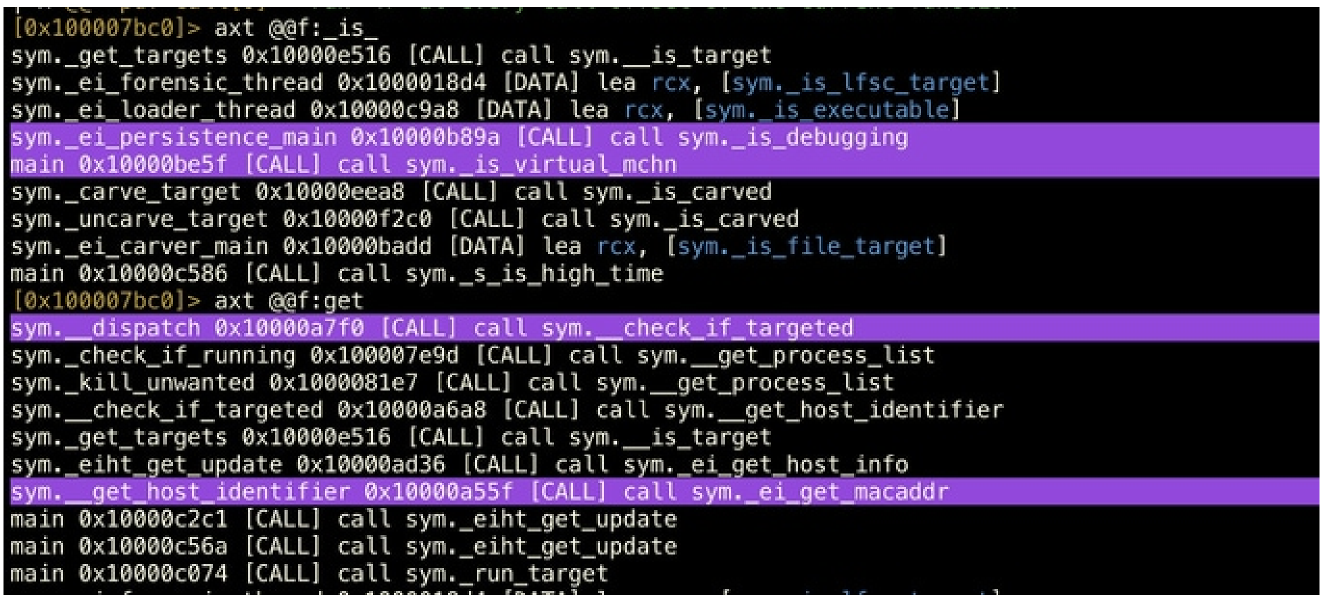

```
[0x10000bd80]> axt sym._is_
sym._is_lfsc_target    sym._is_executable     sym._is_debugging      sym._is_virtual_mchn    sym._is_carved
sym._is_file_target
[0x10000bd80]> axt sym._is_debugging
sym._ei_persistence_main 0x10000b89a [CALL] call sym._is_debugging
[0x10000bd80]> axt sym._is_virtual_mchn
main 0x10000be5f [CALL] call sym._is_virtual_mchn
[0x10000bd80]>
```
Many commands in r2 support tab expansion

Here's a useful powertrick for those already comfortable with r2. You can run any command on a for-each loop using `@@`. For example, with

```
axt @@f:<search term>
```

we can get the XREFS to any function containing the search term in one go.

In this case I tell r2 to give me the XREFS for every function that contains "_is_". Then I do the same with "get". Try `@@?` to see more examples of what you can do with `@@`.


```
[0x100007bc0]> axt @@f:_is_
sym._get_targets 0x10000e516 [CALL] call sym.__is_target
sym._ei_forensic_thread 0x1000018d4 [DATA] lea rcx, [sym._is_lfsc_target]
sym._ei_loader_thread 0x10000c9a8 [DATA] lea rcx, [sym._is_executable]
sym._ei_persistence_main 0x10000b89a [CALL] call sym._is_debugging
main 0x10000be5f [CALL] call sym._is_virtual_mchn
sym._carve_target 0x10000eea8 [CALL] call sym._is_carved
sym._uncarve_target 0x10000f2c0 [CALL] call sym._is_carved
sym._ei_carver_main 0x10000badd [DATA] lea rcx, [sym._is_file_target]
main 0x10000c586 [CALL] call sym._s_is_high_time
[0x100007bc0]> axt @@f:get
sym.__dispatch 0x10000a7f0 [CALL] call sym.__check_if_targeted
sym._check_if_running 0x100007e9d [CALL] call sym.__get_process_list
sym._kill_unwanted 0x1000081e7 [CALL] call sym.__get_process_list
sym.__check_if_targeted 0x10000a6a8 [CALL] call sym.__get_host_identifier
sym._get_targets 0x10000e516 [CALL] call sym.__is_target
sym._eiht_get_update 0x10000ad36 [CALL] call sym._ei_get_host_info
sym.__get_host_identifier 0x10000a55f [CALL] call sym._ei_get_macaddr
main 0x10000c2c1 [CALL] call sym._eiht_get_update
main 0x10000c56a [CALL] call sym._eiht_get_update
main 0x10000c074 [CALL] call sym._run_target
```
Using a for-each in radare2

Since we see that `is_virtual_mchn` is called in `main`, we should start by disassembling the entire `main` function to see what's going on, but first I'm going to change the r2 color theme to something a bit more reader-friendly with the `eco` command (try `eco` and hit the `tab` key to see a list of available themes).

```
eco focus
pdf @ main
```

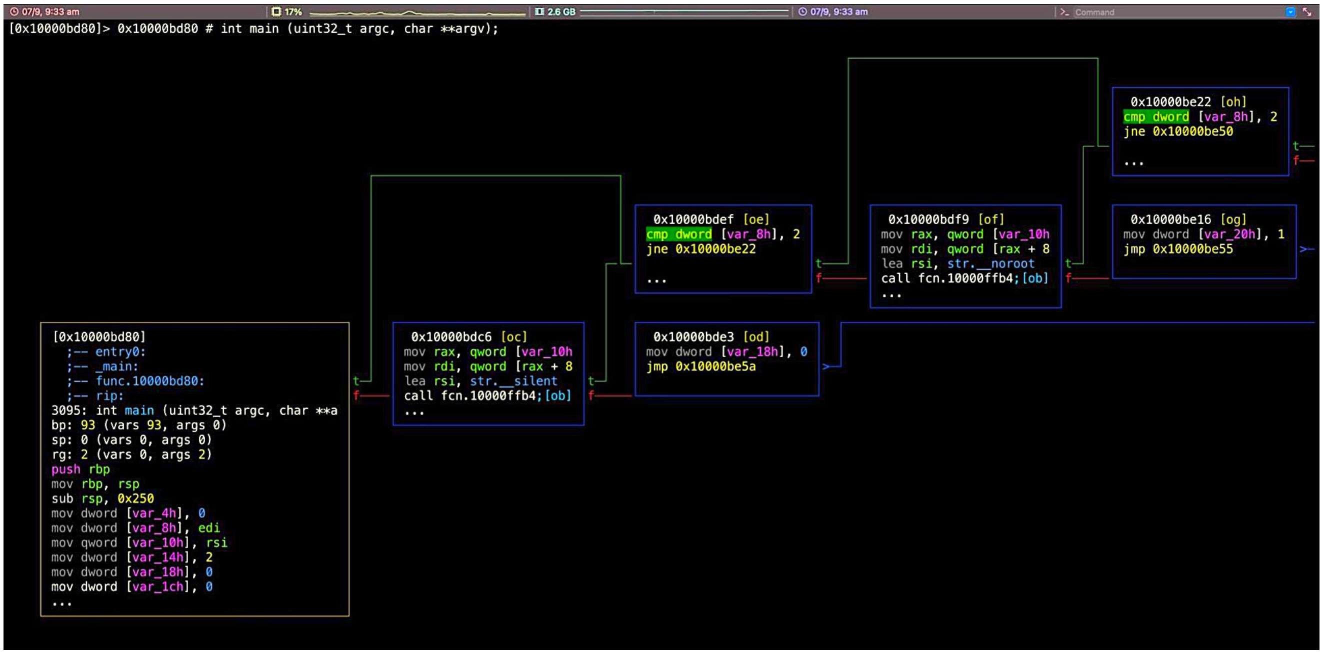
Visual Graph Mode and Renaming Functions with Radare2

```
0x10000bdc0    0f8529000000    jne 0x10000bdef
0x10000bdc6    488b45f0        mov rax, qword [var_10h]
0x10000bdca    488b7808        mov rdi, qword [rax + 8]
0x10000bdce    488d35774b00.   lea rsi, str.__silent       ; 0x10001094c ; "--silent"
0x10000bdd5    e8da410000      call fcn.10000ffb4
0x10000bdda    83f800          cmp eax, 0
0x10000bddd    0f850c000000    jne 0x10000bdef
0x10000bde3    c745e8000000.   mov dword [var_18h], 0
0x10000bdea    e96b000000      jmp 0x10000be5a
; CODE XREFS from main @ 0x10000bdc0, 0x10000bddd
0x10000bdef    837df802        cmp dword [var_8h], 2
0x10000bdf3    0f8529000000    jne 0x10000be22
0x10000bdf9    488b45f0        mov rax, qword [var_10h]
0x10000bdfd    488b7808        mov rdi, qword [rax + 8]
0x10000be01    488d35b25c00.   lea rsi, str.__noroot       ; 0x100011aba ; "--noroot"
0x10000be08    e8a7410000      call fcn.10000ffb4
0x10000be0d    83f800          cmp eax, 0
0x10000be10    0f850c000000    jne 0x10000be22
0x10000be16    c745e0010000.   mov dword [var_20h], 1
0x10000be1d    e933000000      jmp 0x10000be55
; CODE XREFS from main @ 0x10000bdf3, 0x10000be10
0x10000be22    837df802        cmp dword [var_8h], 2
0x10000be26    0f8524000000    jne 0x10000be50
0x10000be2c    488b45f0        mov rax, qword [var_10h]
0x10000be30    488b7808        mov rdi, qword [rax + 8]
0x10000be34    488d35885c00.   lea rsi, str.__ignrp        ; 0x100011ac3 ; "--ignrp"
0x10000be3b    e874410000      call fcn.10000ffb4
0x10000be40    83f800          cmp eax, 0
0x10000be43    0f8507000000    jne 0x10000be50
0x10000be49    c745dc010000.   mov dword [var_24h], 1
; CODE XREFS from main @ 0x10000be26, 0x10000be43
0x10000be50    e900000000      jmp 0x10000be55
; CODE XREFS from main @ 0x10000be1d, 0x10000be50
0x10000be55    e900000000      jmp 0x10000be5a
; CODE XREFS from main @ 0x10000bdea, 0x10000be55
0x10000be5a    bf02000000      mov edi, 2                  ; int64_t arg1
0x10000be5f    e85cbdffff      call sym._is_virtual_mchn
0x10000be64    83f800          cmp eax, 0
0x10000be67    0f840a000000    je 0x10000be77
0x10000be6d    bfffffffff      mov edi, 0xffffffff         ; -1
0x10000be72    e83b400000      call fcn.10000feb2
```

As we scroll back up to the beginning of the function, we can see the disassembly provides pretty interesting reading. At the beginning of `main`, we can see some unnamed functions are called. We're going to jump into Visual Graph mode and start renaming code as this will give us a good idea of the malware's execution flow and indicate what we need to do to beat the anti-analysis.

Hit `VV` to enter Visual Graph mode. I will try to walk you through the commands, but if you get lost at any point, don't feel bad. It happens to us all and is part of the r2 learning curve! You can just quit out and start again if needs be (part of the beauty of r2's speed; you can also save your project: type uppercase `P?` to see project options).

I prefer to view the graph as a horizontal, left-to-right flow; you can toggle between horizontal and vertical by pressing the `@` key.



Viewing the sample's visual graph horizontally

Here's a quick summary of some useful commands (there are many more as you'll see if you play around):

- hjkl(arrow keys) – move the graph around
- -/+0 – reduce, enlarge, return to default size
- ' – toggle graph comments
- tab/shift-tab – move to next/previous function
- dr – rename function
- q – back to visual mode
- t/f – follow the true/false execution chain
- u – go back
- ? – help/available options

Hit `'` once or twice make sure graph comments are on.

Use the tab key to move to the first function after `main()` (the border will be highlighted), where we can see an unnamed function and a reference in square brackets that begins with the letter 'o' (for example, `[ob]`, though it may be different in your sample). Type the letters (without the square brackets) to go to that function. Type `p` to rotate between different display modes till you see something similar to the next image.

```
[0x10000ffb4]
6: fcn.10000ffb4 ();
bp: 0 (vars 0, args 0)
sp: 0 (vars 0, args 0)
rg: 0 (vars 0, args 0)
0x10000ffb4 ff25de320000   jmp qword [reloc.strcmp]
```

As we can see, this function call is actually a call to the standard C library function `strcmp()`, so let's rename it.

Type `dr` and at the prompt type in the name you want to use and hit 'enter'. Unsurprisingly, I'm going to call it `strcmp`.

```
[0x10000ffb4]
  ;-- strcmp__:
6: int strcmp (const char *s1, const char *s2);
bp: 0 (vars 0, args 0)
sp: 0 (vars 0, args 0)
rg: 0 (vars 0, args 0)
0x10000ffb4 ff25de320000   jmp qword [reloc.strcmp]
```

To return to the main graph, type `u` and you should see that all references to that previously unnamed function now show `strcmp`, making things much clearer.

If you scroll through the graph (hjkl, remember) you will see many other unnamed functions that, once you explore them in the same way, are just relocations of standard C library calls such as `exit`, `time`, `sleep`, `printf`, `malloc`, `srandom` and more. I suggest you repeat the above exercise and rename as many as you can. This will both make the malware's behaviour easier to understand and build up some valuable muscle-memory for working in r2!

Beating Anti-Analysis Without Patching

There are two approaches you can take to interrupt a program's designed logic. One is to identify functions you want to avoid and patch the binary statically. This is fairly easy to do in r2 and there's quite a few tutorials on how to patch binaries already out there. We're not going to look at patching today because our entire objective is to run the sample dynamically, so we might as well interact with the program dynamically as well. Patching is really only worth considering if you need to create a sample for repeated use that avoids some kind of unwanted behaviour.

We basically have two easy options in terms of affecting control flow dynamically. We can either execute the function but manipulate the returned value (like put 0 in rax instead of 1) or skip execution of the function altogether.

We'll see just how easy it is to do each of these, but we should first think about the different consequences of each choice based on the malware we're dealing with.

If we NOP a function or skip over it, we're going to lose any behaviour or memory states invoked by that function. If the function doesn't do anything that affects the state of our program later on, this can be a good choice.

By the same token, if we execute the function but manipulate the value it returns, we may be allowing execution of code buried in that function that might trip us up. For example, if our function contains jumps to subroutines that do further anti-analysis tests, then we might get blocked before the parent function even returns, so this strategy wouldn't help us. Clearly then, we need to take a look around the code to figure out which is the best strategy in each particular case.

Let's take a look inside the `_is_virtual_mchn` function to see what it would do and work out our strategy.

If you're still in Visual Graph mode, hit `q` to get back to the r2 prompt. Regardless of where you are, you can disassemble a function with `pdf` and the `@` symbol and provide a flag or address. Remember, you can also use tab expansion to get a list of possible symbols.

```
[0x100007bc0]> pdf @ sym._is_
sym._is_lfsc_target    sym._is_executable     sym._is_debugging      sym._is_virtual_mchn   sym._is_carved
sym._is_file_target
[0x100007bc0]> pdf @ sym._is_virtual_mchn
            ;-- func.100007bc0:
            ; CALL XREF from main @ 0x10000be5f
┌ 83: sym._is_virtual_mchn (int64_t arg1);
│           ; var int64_t var_1ch @ rbp-0x1c
│           ; var int64_t var_18h @ rbp-0x18
│           ; var int64_t var_10h @ rbp-0x10
│           ; var int64_t var_4h @ rbp-0x4
│           ; arg int64_t arg1 @ rdi
│           0x100007bc0      55             push rbp
│           0x100007bc1      4889e5         mov rbp, rsp
│           0x100007bc4      4883ec20       sub rsp, 0x20
│           0x100007bc8      31c0           xor eax, eax
│           0x100007bca      89c1           mov ecx, eax
│           0x100007bcc      897dfc         mov dword [var_4h], edi    ; arg1
│           0x100007bcf      4889cf         mov rdi, rcx
│           0x100007bd2      e807840000     call time()
│           0x100007bd7      488945f0       mov qword [var_10h], rax
│           0x100007bdb      8b7dfc         mov edi, dword [var_4h]
│           0x100007bde      e8b3830000     call sleep                 ; int sleep(int s)
│           0x100007be3      31ff           xor edi, edi
│           0x100007be5      8945e4         mov dword [var_1ch], eax
│           0x100007be8      e8f1830000     call time()
│           0x100007bed      31d2           xor edx, edx
│           0x100007bef      488945e8       mov qword [var_18h], rax
│           0x100007bf3      488b45e8       mov rax, qword [var_18h]
│           0x100007bf7      482b45f0       sub rax, qword [var_10h]
│           0x100007bfb      8b75fc         mov esi, dword [var_4h]
│           0x100007bfe      89f1           mov ecx, esi
│           0x100007c00      4839c8         cmp rax, rcx
│           0x100007c03      be01000000     mov esi, 1
│           0x100007c08      0f4cd6         cmovl edx, esi
│           0x100007c0b      89d0           mov eax, edx
│           0x100007c0d      4883c420       add rsp, 0x20
│           0x100007c11      5d             pop rbp
└           0x100007c12      c3             ret
[0x100007bc0]>
```

It seems this function subtracts the sleep interval from the second timestamp, then compares it against the first timestamp. Jumping back out to how this result is consumed in `main`, it seems that if the result is not ‘0’, the malware calls `exit()` with ‘-1’.

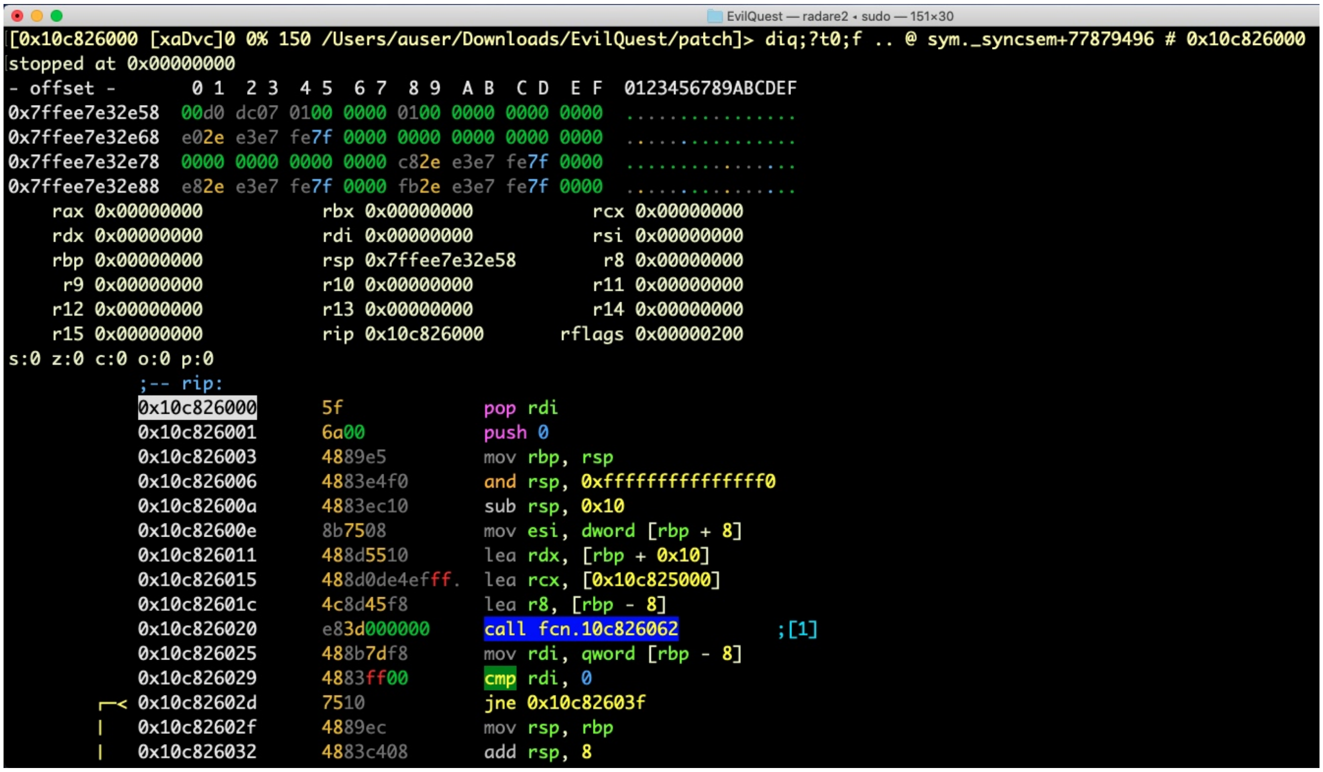
```
│      └─└─> 0x10000be5a      bf02000000     mov edi, 2                 ; int64_t arg1
│            0x10000be5f      e85cbdffff     call sym._is_virtual_mchn
│            0x10000be64      83f800         cmp eax, 0
│      ┌─< 0x10000be67      0f840a000000   je 0x10000be77
│      │     0x10000be6d      bfffffffff     mov edi, 0xffffffff        ; -1
│      │     0x10000be72      e83b400000     call exit()
│      │     ; CODE XREF from main @ 0x10000be67
│      └─> 0x10000be77      48c745d00000.  mov qword [var_30h], 0
│            0x10000be7f      c745cc000000.  mov dword [var_34h], 0
│            0x10000be86      c745c8000000.  mov dword [var_38h], 0
│            0x10000be8d      488d7dd0       lea rdi, [var_30h]         ; int64_t arg1
│            0x10000be91      488d75cc       lea rsi, [var_34h]         ; int64_t arg2
│            0x10000be95      e8866dffff     call sym._user_info
│            0x10000be9a      83f800         cmp eax, 0
```

The is_virtual_mchn function causes the malware to exit unless it returns ‘0’

The function appears to be somewhat misnamed as we don’t see the kind of tests that we would normally expect for VM detection. In fact, it looks like an attempt to evade automated sandboxes that patch the sleep function, and we’re not likely to fall foul of it just by executing in our VM. However, we can also see that the next function, `user_info`, also exits if it doesn't return the expected value, so let's practice both the techniques discussed above so that we can learn how to use the debugger whichever one we need to use.

Manipulating Execution with the radare2 Debugger

If you are at the command prompt, type `Vp` to go into radare2 visual mode (yup, this is another mode, and not the last!).



```
EvilQuest — radare2 • sudo — 151×30
[0x10c826000 [xaDvc]0 0% 150 /Users/auser/Downloads/EvilQuest/patch]> diq;?t0;f .. @ sym._syncsem+77879496 # 0x10c826000
stopped at 0x00000000
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffee7e32e58  00d0 dc07 0100 0000 0100 0000 0000 0000  ................
0x7ffee7e32e68  e02e e3e7 fe7f 0000 0000 0000 0000 0000  ................
0x7ffee7e32e78  0000 0000 0000 0000 c82e e3e7 fe7f 0000  ................
0x7ffee7e32e88  e82e e3e7 fe7f 0000 fb2e e3e7 fe7f 0000  ................
    rax 0x00000000          rbx 0x00000000          rcx 0x00000000
    rdx 0x00000000          rdi 0x00000000          rsi 0x00000000
    rbp 0x00000000          rsp 0x7ffee7e32e58       r8 0x00000000
     r9 0x00000000          r10 0x00000000          r11 0x00000000
    r12 0x00000000          r13 0x00000000          r14 0x00000000
    r15 0x00000000          rip 0x10c826000      rflags 0x00000200
s:0 z:0 c:0 o:0 p:0
            ;-- rip:
            0x10c826000      5f             pop rdi
            0x10c826001      6a00           push 0
            0x10c826003      4889e5         mov rbp, rsp
            0x10c826006      4883e4f0       and rsp, 0xfffffffffffffff0
            0x10c82600a      4883ec10       sub rsp, 0x10
            0x10c82600e      8b7508         mov esi, dword [rbp + 8]
            0x10c826011      488d5510       lea rdx, [rbp + 0x10]
            0x10c826015      488d0de4efff.  lea rcx, [0x10c825000]
            0x10c82601c      4c8d45f8       lea r8, [rbp - 8]
            0x10c826020      e83d000000     call fcn.10c826062          ;[1]
            0x10c826025      488b7df8       mov rdi, qword [rbp - 8]
            0x10c826029      4883ff00       cmp rdi, 0
        ┌─< 0x10c82602d      7510           jne 0x10c82603f
        │   0x10c82602f      4889ec         mov rsp, rbp
        │   0x10c826032      4883c408       add rsp, 8
```

The Visual Debugger in radare2

Ooh, this is nice! We get registers at the top, and source code underneath. The current line where we're stopped in the debugger is highlighted. If you don't see that, hit uppercase `S` once (i.e., `shift-s`), which steps over one source line, and – in case you lose your way – also brings you back to the debugger view.

Let's step smartly through the source with repeated uppercase `S` commands (by the way, in visual mode, lowercase 's' steps in, whereas uppercase 'S' steps over). After a dozen or so rapid step overs, you should find yourself inside this familiar code, which is `main()`.

```
[0x107dd8d84 [xaDvc]0 0% 170 /Users/auser/Downloads/EvilQuest/patch]> diq;?t0;f .. @ main+4 # 0x107dd8d84
step at 0x107dd8d81
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffee7e32e40  582e e3e7 fe7f 0000 c97c ef70 ff7f 0000  X.......|.p....
0x7ffee7e32e50  c97c ef70 ff7f 0000 0000 0000 0000 0000  .|.p...........
0x7ffee7e32e60  0100 0000 0000 0000 e02e e3e7 fe7f 0000  ...............
0x7ffee7e32e70  0000 0000 0000 0000 0000 0000 0000 0000  ...............
    rax 0x107dd8d80         rbx 0x00000000          rcx 0x7ffee7e32e80
    rdx 0x7ffee7e32e78      rdi 0x00000001          rsi 0x7ffee7e32e68
    rbp 0x7ffee7e32e40      rsp 0x7ffee7e32e40       r8 0x00000000
     r9 0x00000000          r10 0x00000000          r11 0x00000000
    r12 0x00000000          r13 0x00000000          r14 0x00000000
    r15 0x00000000          rip 0x107dd8d84       rflags 0x00000346
s:0 z:1 c:0 o:0 p:1
|           ;-- rip:
|           0x107dd8d84      4881ec50020000.   sub rsp, 0x250
|           0x107dd8d8b      c745fc00000000.   mov dword [var_4h], 0
|           0x107dd8d92      897df8            mov dword [var_8h], edi     ; argc
|           0x107dd8d95      488975f0          mov qword [var_10h], rsi     ; argv
|           0x107dd8d99      c745ec02000000.   mov dword [var_14h], 2
|           0x107dd8da0      c745e800000000.   mov dword [var_18h], 0
|           0x107dd8da7      c745e400000000.   mov dword [var_1ch], 0
|           0x107dd8dae      c745e000000000.   mov dword [var_20h], 0
|           0x107dd8db5      c745dc00000000.   mov dword [var_24h], 0
|           0x107dd8dbc      837df802          cmp dword [var_8h], 2
|       ┌─< 0x107dd8dc0      0f8529000000      jne 0x107dd8def
|       │   0x107dd8dc6      488b45f0          mov rax, qword [var_10h]
|       │   0x107dd8dca      488b7808          mov rdi, qword [rax + 8]
|       │   0x107dd8dce      488d35774b00.     lea rsi, str.__silent       ; 0x107ddd94c ; "--silent"
|       │   0x107dd8dd5      e8da410000        call fcn.107ddcfb4          ;[1]
|       │   0x107dd8dda      83f800            cmp eax, 0
|      ┌──< 0x107dd8ddd      0f850c000000      jne 0x107dd8def
|      ││   0x107dd8de3      c745e800000000.   mov dword [var_18h], 0
|     ┌───< 0x107dd8dea      e96b000000        jmp 0x107dd8e5a
```
main() in Visual Debugger mode

Note the highlighted dword, which is holding the value of `argc`. It should be ‘2’, but we can see from the register above that `rdi` is only 1. The code will jump over the next function call, which if you hit the ‘1’ key on the keyboard you can inspect (hit `u` to come back) and see this is a string comparison. Let’s continue stepping over and let the jump happen, as it doesn’t appear to block us. We’ll stop just short of the `is_virtual_mchn` function.

```
[0x101028e50 [xAdvc]0 0% 183 /Users/auser/Downloads/EvilQuest/patch]> pd $r @ main+208 # 0x101028e50
|      ┌─< 0x101028e50      e900000000      jmp 0x101028e55
|      |   ; CODE XREFS from main @ 0x101028e1d, 0x101028e50
|     ┌└─> 0x101028e55      e900000000      jmp 0x101028e5a
|     |    ;-- rip:
|     |    ; CODE XREFS from main @ 0x101028dea, 0x101028e55
|     └──> 0x101028e5a      bf02000000      mov edi, 2
|          0x101028e5f      e85cbdffff      call sym._is_virtual_mchn   ;[1]
|          0x101028e64      83f800          cmp eax, 0
|      ┌─< 0x101028e67      0f840a000000    je 0x101028e77
|      |   0x101028e6d      bfffffffff      mov edi, 0xffffffff         ; -1
|      |   0x101028e72      e83b400000      call 0x10102ceb2            ;[2]
|      └─> 0x101028e77      48c745d00000.   mov qword [var_30h], 0
|          0x101028e7f      c745cc000000.   mov dword [var_34h], 0
|          0x101028e86      c745c8000000.   mov dword [var_38h], 0
|          0x101028e8d      488d7dd0        lea rdi, [var_30h]
|          0x101028e91      488d75cc        lea rsi, [var_34h]
|          0x101028e95      e8866dffff      call sym._user_info         ;[3]
|          0x101028e9a      83f800          cmp eax, 0
|      ┌─< 0x101028e9d      0f840a000000    je 0x101028ead
|      |   0x101028ea3      bfffffffff      mov edi, 0xffffffff         ; -1
|      |   0x101028ea8      e805400000      call 0x10102ceb2            ;[2]
|      └─> 0x101028ead      48c745c00000.   mov qword [var_40h], 0
|          0x101028eb5      488b45f0        mov rax, qword [var_10h]
```

Seek and break locations are two different things!

We know from our earlier discussion what's going to happen here, so let's see how to take each of our options.

The first thing to note is that although the highlighted address is where the debugger is, that's not where you are if you enter an r2 command prompt, unless it's a debugger command. To see what I mean, hit the colon key to enter the command line.

From there, print out one line of disassembly with this command:

```
> pd 1
```

Note that the line printed out is r2's current seek position, shown at the top of the visual view. This is good. It means you can move around the program, seek to other functions and run other r2 commands without disturbing the debugger.

On the other hand, if you execute a debugger command on the command line it will operate on the source code where the debugger is currently parked, not on the current seek at the top of your view (unless they happen to be the same).

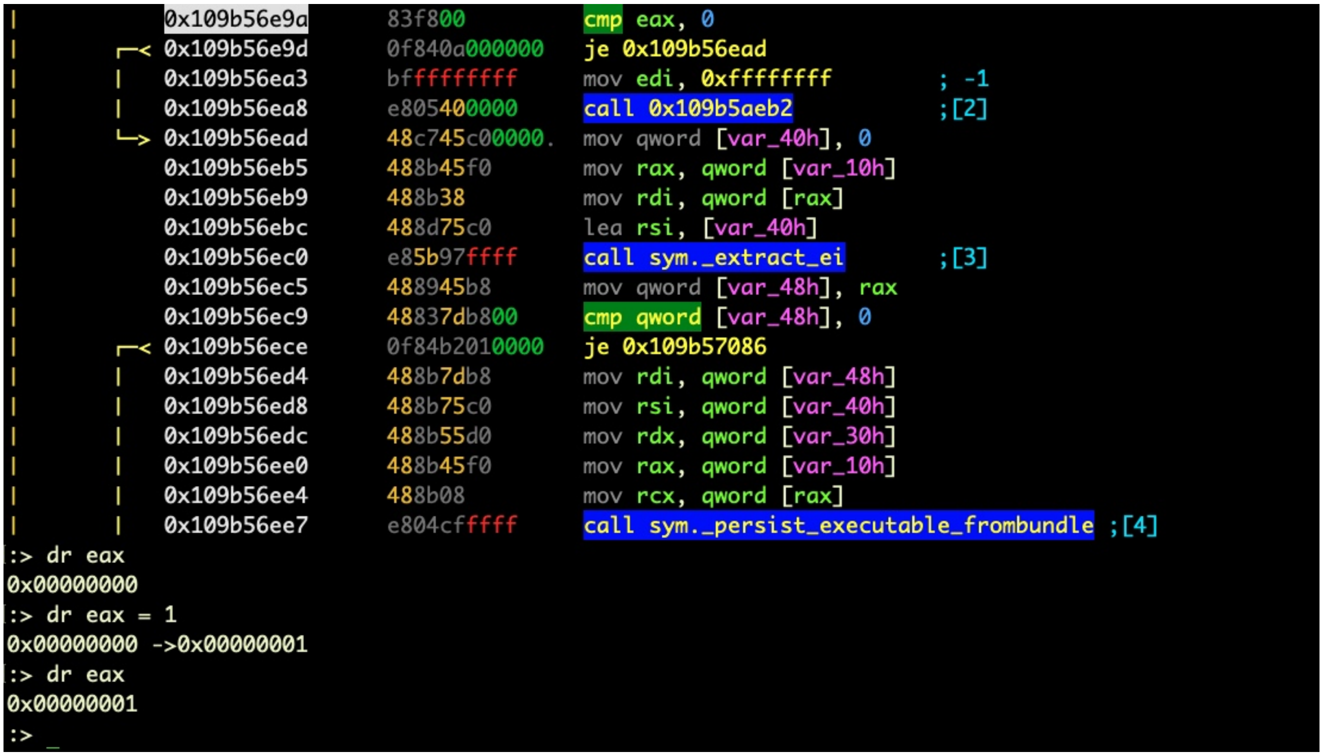
OK, let's entirely skip execution of the `_is_virtual_mchn` function by entering the command line with `:` and then:

```
> dss 2
```

Hit 'return' twice. As you can see, the `dss` command skips the number of source lines specified by the integer you gave it, making it a very easy way to bypass unwanted code execution!

Alternatively, if we want to execute the function then manipulate the register, stop the debugger on the line where the register is compared, and enter the command line again. This time, we can use `dr` to both inspect and write values to our chosen register.

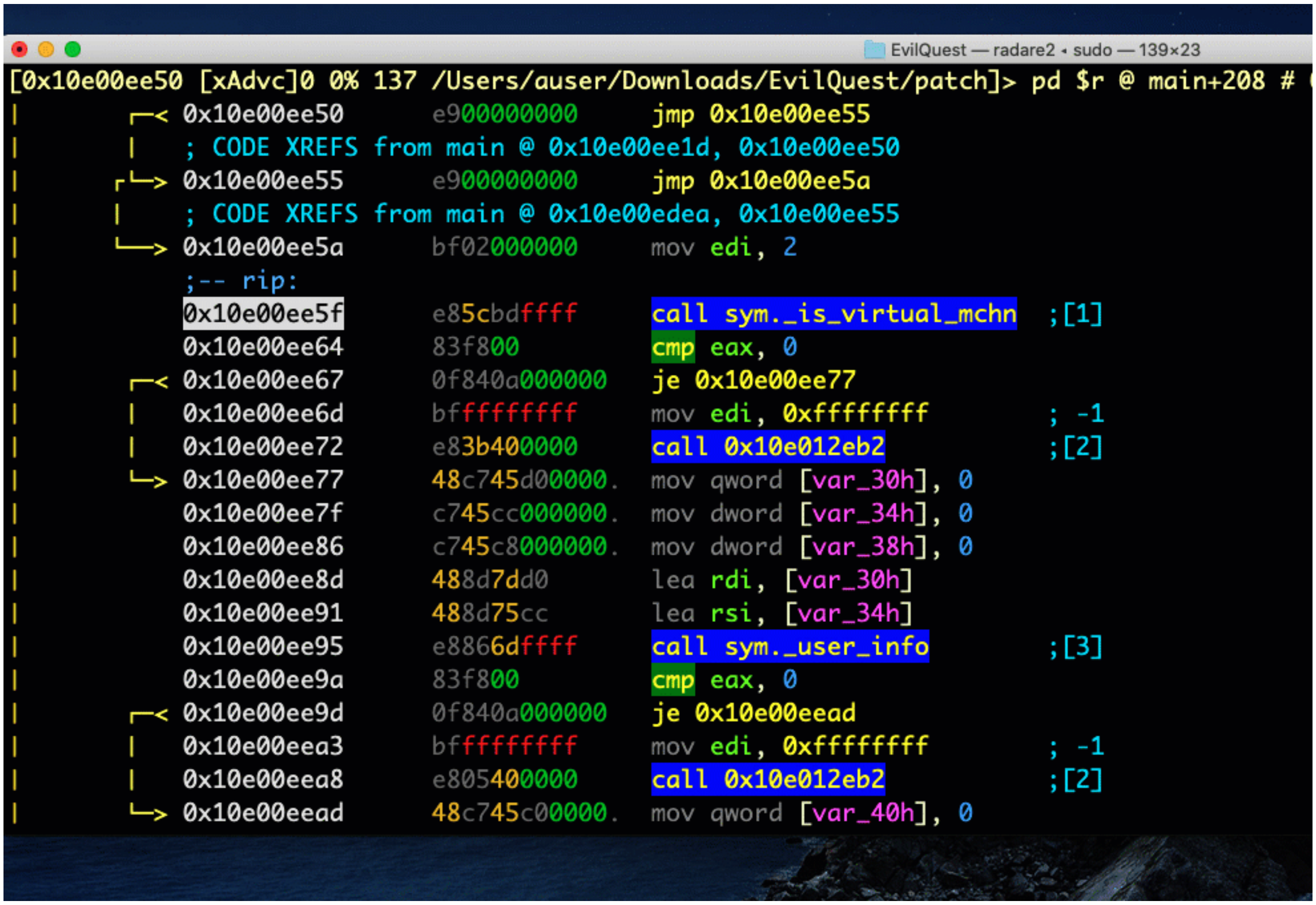
```
> dr eax // see eax's current value
> dr eax = 0 // set eax to 0
> drr // view all the registers
> dro // see the previous values of the registers
```



Viewing and changing register values

And that, pretty much, is all you need to defeat anti-analysis code in terms of manipulating execution. Of course, the fun part is finding the code you need to manipulate, which is why we spent some time learning how to move around in radare2 in both visual graph mode and visual mode. Remember that in either mode you can get back to the regular command prompt by hitting `q`. As a bonus, you might play around with hitting `p` and `tab` when in the visual modes.

At this point, what I suggest you do is go back to the list of functions we identified at the beginning of the post and see what they do, and whether it's best to skip them or modify their return values (or whether either option will do). You might want to look up the built-in help for listing and setting breakpoints (from a command prompt, try `db?`) to move quickly through the code. By the time you've done this a few times, you'll be feeling pretty comfortable about tackling other samples in radare2's debugger.



Conclusion

If you're starting to see the potential power of r2, I strongly suggest you read the free online radare2 book, which will be well worth investing the time in. By now you should be starting to get the feel of r2 and exploring more on your own with the help of the `?` and other resources. As we go into further challenges, we'll be spending less time going over the r2 basics and digging more into the actual malware code.

In the next part of our series, we're going to start looking at one of the major challenges in reversing macOS malware that you are bound to face on a regular basis: dealing with encrypted and obfuscated strings. I hope you'll join us there and practice your r2 skills in the meantime!