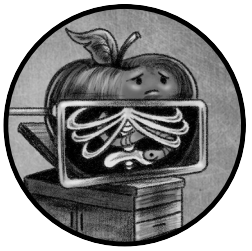
9

ANTI-ANALYSIS



In the previous chapters, we leveraged both static and dynamic analysis methods to uncover malware's persistence mechanisms, core capabilities, and 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 complicate analysis by writing anti-analysis logic or other protection schemes. In order to successfully analyze such malware, we must first identify these protections and then circumvent them.

In this chapter we'll discuss anti-analysis approaches common among macOS malware authors. Generally speaking, there are two kinds of anti-analysis measures: those that aim to thwart static analysis and those that seek to thwart dynamic analysis. Let's take a look at both.

Anti-Static-Analysis Approaches

Malware authors use several common approaches to complicate static analysis efforts:

- **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 its command and control server?" Malware that contains plaintext strings related to its persistence, like filepaths or the URL of its command and control server, makes analysis almost too easy. As such, malware authors often obfuscate or encrypt these sensitive strings.
- **Code obfuscation**: In order to complicate the static analysis of their code (and sometimes dynamic analysis as well), malware authors can obfuscate the code itself. Various obfuscator tools are available for nonbinary malware specimens like scripts. For Mach-O binaries, malware authors can use executable packers or encryptors to protect the binary's code.

Let's look at a few examples of anti-static-analysis methods and then discuss how to bypass them. As you'll see, it's often easier to overcome anti-static-analysis approaches with dynamic analysis techniques. In some cases, the opposite holds as well; static analysis techniques can reveal anti-dynamic-analysis tactics.

Sensitive Strings Disguised as Constants

One of the most basic string-based obfuscations involves splitting sensitive strings into chunks so that they are inlined directly into assembly instructions as constants. Depending on the chunk size, the `strings` command may miss these strings, while a disassembler, by default, will rather unhelpfully display the chunks as hexadecimal numbers. We find an example of this string obfuscation in Dacls (Listing 9-1):

```
main:
...
0x000000010000b5fa    movabs    rcx, 0x7473696c702e74
0x000000010000b604    mov       qword [rbp+rax+var_209], rcx
0x000000010000b60c    movabs    rcx, 0x746e6567612e706f
0x000000010000b616    mov       qword [rbp+rax+var_210], rcx
0x000000010000b61e    movabs    rcx, 0x6f6c2d7865612e6d
0x000000010000b628    mov       qword [rbp+rax+var_218], rcx
0x000000010000b630    movabs    rcx, 0x6f632f73746e6567
0x000000010000b63a    mov       qword [rbp+rax+var_220], rcx
0x000000010000b642    movabs    rcx, 0x4168636e75614c2f
0x000000010000b64c    mov       qword [rbp+rax+var_228], rcx
0x000000010000b654    movabs    rcx, 0x7972617262694c2f
0x000000010000b65e    mov       qword [rbp+rax+var_230], rcx
```

Listing 9-1: Basic string obfuscation (Dacls)

As you can see, six 64-bit values are moved first into the `RCX` register, then into adjacent stack-based variables. The astute reader will notice that each byte of these values falls within the range of printable ASCII

characters. We can overcome this basic obfuscation using a disassembler. Simply instruct the disassembler to decode the constants as characters instead of the default, hexadecimal. In the Hopper disassembler, you can simply CTRL-click the constant and select **Characters** to use the SHIFT-R keyboard shortcut (Listing 9-2):

```
main:
...
0x0000000010000b5fa    movabs     rcx, 't.plist'
0x0000000010000b604    mov        qword [rbp+rax+var_209], rcx
0x0000000010000b60c    movabs     rcx, 'op.agent'
0x0000000010000b616    mov        qword [rbp+rax+var_210], rcx
0x0000000010000b61e    movabs     rcx, 'm.aex-lo'
0x0000000010000b628    mov        qword [rbp+rax+var_218], rcx
0x0000000010000b630    movabs     rcx, 'gents/co'
0x0000000010000b63a    mov        qword [rbp+rax+var_220], rcx
0x0000000010000b642    movabs     rcx, '/LaunchA'
0x0000000010000b64c    mov        qword [rbp+rax+var_228], rcx
0x0000000010000b654    movabs     rcx, '/Library'
0x0000000010000b65e    mov        qword [rbp+rax+var_230], rcx
```

Listing 9-2: Deobfuscated strings (Dacls)

If we reconstitute the split string (noting the slight overlap of the first two string components), this deobfuscated disassembly now reveals the path of the malware's persistent launch item: */Library/LaunchAgents/com.aex-loop.agent.plist*.

Encrypted Strings

In previous chapters, we looked at several more complex examples of string-based obfuscations. For example, in Chapter 7 we noted that WindTail contains various embedded base64-encoded and AES-encrypted strings, including the address of its command and control server. The encryption key needed to decrypt the string is hardcoded within the malware, meaning it would be possible to manually decode and decrypt the server's address. However, this would involve some legwork, such as finding (or scripting up) an AES decryptor. Moreover, if the malware used a custom (or nonstandard) algorithm to encrypt the strings, even more work would be involved. Of course, at some point the malware will have to decode and decrypt the protected strings so that it can use them, such as to connect to a command and control server for tasking. As such, it's often far more efficient to simply allow the malware to run, which should trigger the decryption of its strings. If you're monitoring the execution of the malware, the decrypted strings can be easily recovered.

In Chapter 7, I showed one technique for doing this: using a network monitor, which allowed us to passively recover the (previously encrypted) address of the malware's command and control server as the malware beaconed out for tasking. We can accomplish the same thing using a debugger, as you'll see here. First, we locate WindTail's decryption logic, a method named `yoop:`. (In a subsequent section, I'll describe how to locate such methods.) Looking at cross-references to this method, we can see it's invoked any time the malware needs to decrypt one of its strings prior to use. For example,

Listing 9-3 shows a snippet of disassembly that invokes the `yoop:` method ❶ to decrypt the malware's primary command and control server.

```
0x0000000100001fe5    mov       r13, qword [objc_msgSend]
...
0x0000000100002034    mov       rsi, @selector(yoop:)
0x000000010000203b    lea       rdx, @"F5UrOCCFMOfWHjecxEqGLy...OLs="
0x0000000100002042    mov       rdi, self
❶ 0x0000000100002045    call      r13

❷ 0x0000000100002048    mov       rcx, rax
```

Listing 9-3: Decryption of a command and control server (WindTail)

We can set a debugger breakpoint at `0x100002048`, which is the address of the instruction immediately after the call to `yoop:` ❷. Because the `yoop:` method returns a plaintext string, we can print this string when we hit this breakpoint. (Recall that a method's return value can be found in the `RAX` register.) This reveals the malware's primary command and control server, *flux2key.com*, as shown in Listing 9-4:

```
% 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=%@
```

Listing 9-4: A decrypted command and control address (WindTail)

It's worth noting that you could also set a breakpoint on the return instruction (`retn`) within the decryption function. When the breakpoint is hit, you'll once again find the decrypted string in the `RAX` register. A benefit of this approach is that you only have to set a single breakpoint, instead of several at the locations from which the decryption method is invoked. This means that any time the malware decrypts, not just its command and control server but any string, you'll be able to recover the plaintext of that as well. However, it would become rather tedious to manually manage this breakpoint, as it will be invoked many times to decrypt each of the malware's strings. A more efficient approach would be to add additional debugger commands (via `breakpoint command add`) to the breakpoint. Then, once the breakpoint is hit, your breakpoint commands will be automatically executed and could just print out the register holding the decrypted string and then allow the process to automatically continue. If you're interested in the caller, perhaps to locate where a specific decrypted string is used, consider printing out the stack backtrace as well.

Note that this breakpoint-based 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 technique the malware is using to protect strings or data. If you're able to locate the deobfuscation or decryption routine during static analysis, all you'll need in order to read the string is a well-placed debugger breakpoint.

Locating Obfuscated Strings

Of course, this begs the question: How can you determine that malware has obfuscated sensitive strings and data? And how can you locate the routines within the malware responsible for returning their plaintext values?

While there are no foolproof methods for the latter, it's generally straightforward to ascertain if a malicious specimen has something to hide. Take, for example, the output of the `strings` command, which usually produces a significant number of extracted strings. 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. For example, if we run `strings` on WindTail, we'll find various plaintext strings alongside what appear to be obfuscated strings (Listing 9-5):

```
% strings - Final_Presentation.app/Contents/MacOS/usrnode

/bin/sh
open -a
Song.dat
KEY_PATH
oXOs4Qj3GiAzAnOmzGqjOA==
ie8DGq3HZ82UqV9N4cpuVw==
F5UrOCCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmHE6gRZGU7ZmXiW+/gzAouX
aagHdDG+YP9BEmHLCg9PVXOuIlMB12oTVPlb8CHvda6TWtptKmqJVvI4o63iQ36Shy9Y9hPtlh+kcrCLOuj+tQ==
```

Listing 9-5: Obfuscated strings (WindTail)

Of course, this method is not foolproof. For example, if the obfuscation method, such as an 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 many or large chunks of obfuscated or high entropy data that are cross-referenced elsewhere in the binary code. For example, malware called NetWire (which installs a malicious application named *Finder.app*) contains what appears to be a blob of encrypted data near the start of the `__data` section (Figure 9-1).

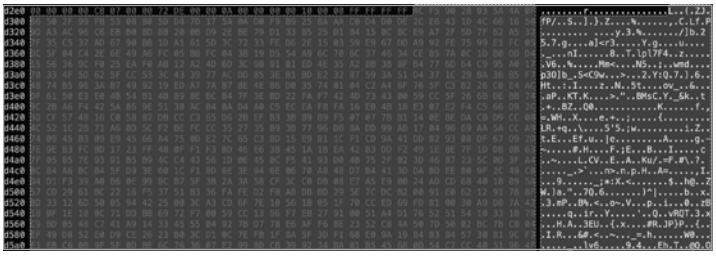


Figure 9-1: Embedded obfuscated data (NetWire)

A continued triage of the malware's `main` function reveals multiple calls to a function at `0x00009502`. 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
0x0000736b    call      sub_9502
...
0x00007380    push      0xe5d6
0x00007385    push      eax
0x00007386    call      sub_9502
...
0x000073fd    push      0xe6b6
0x00007402    push      edi
0x00007403    call      sub_9502
```

It seems reasonable to assume that this function is responsible for decrypting the contents of the blob of encrypted data. As noted previously, you can usually set a breakpoint after code that references the encrypted data and then dump the decrypted data. In the case of NetWire, we can set a breakpoint immediately after the final call to the decryption function, and then we can examine the decrypted data in memory. As it decrypts to a sequence of printable strings, we can display it via the `x/s` debugger command, as in Listing 9-6:

```
% 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 * thread #1, queue = 'com.apple.main-thread', stop reason
= breakpoint 1.1

(lldb) x/20s 0x0000e2f0
❶ 0x0000e2f8: "89.34.111.113:443;"
  0x0000e4f8: "Password"
  0x0000e52a: "HostId-%Rand%"
  0x0000e53b: "Default Group"
  0x0000e549: "NC"
  0x0000e54c: "-"
❷ 0x0000e555: "%home%/.defaults/Finder"
  0x0000e5d6: "com.mac.host"
  0x0000e607: "{0Q44F73L-1XD5-6N1H-53K4-I28DQ30QB8Q1}"
  ...
```

Listing 9-6: Dumping now-decrypted configuration parameters (NetWire)

The contents turn out to be configuration parameters that include the address of the malware's command and control server ❶, as well as its installation path ❷. Recovering these configuration parameters greatly expedites our analysis.

Finding the Deobfuscation Code

When we encounter obfuscated or encrypted data in a malicious sample, it's important to locate the code that deobfuscates or decrypts this data. Once we've done so, we can set a debugging breakpoint and recover the plaintext. This raises the question of how we can locate that code within the malware.

Usually, the best approach is to use a disassembler or decompiler to identify code that references the encrypted data. These references generally indicate either the code responsible for decryption or code that later references the data in a decrypted state.

For example, in the case of WindTail, we noted various strings that appeared to be obfuscated. If we select one such string (`"BouCfWujdfbAUfCos/iIOg=="`), we find it is referenced in the following disassembly (Listing 9-7):

```
0x0000000010000239f    mov       rsi, @selector(yoop:)
0x00000001000023a6     lea       rdx, @"BouCfWujdfbAUfCos/iIOg=="
0x00000001000023ad     mov       r15, qword [_objc_msgSend]
0x00000001000023b4     call      r15
```

Listing 9-7: Possible string deobfuscation (WindTail)

Recall that the `objc_msgSend` function is used to invoke Objective-C methods, that the `RSI` register will hold the name of the method being invoked, and that the `RDI` register will hold its first parameter. From the disassembly that references the obfuscated string, we can see that the malware is invoking the `yoop:` method with the obfuscated string as its parameter. Enumerating cross-references to the `yoop:` selector (found at `0x100015448`) reveals that the method is invoked once for each string that needs to be decoded and decrypted (Figure 9-2).

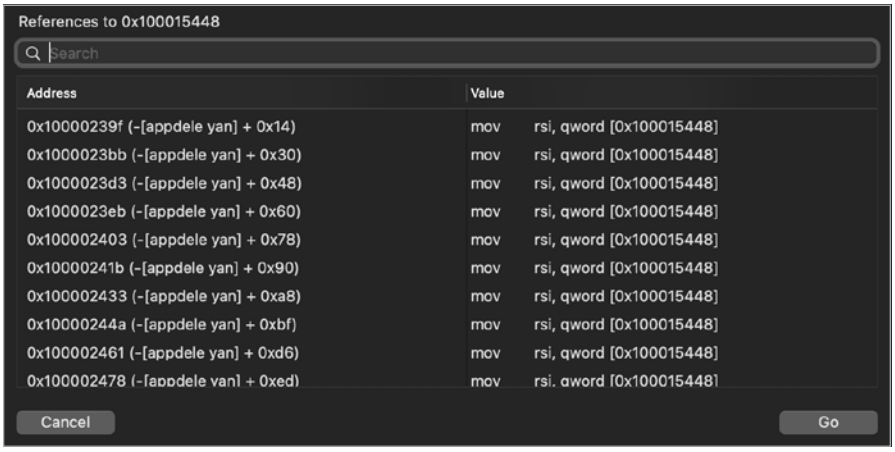


Figure 9-2: Cross-references to `@selector(yoop:)` (WindTail)

Taking a closer look at the actual `yoop:` method reveals calls to methods named `decode:` and `AESDecryptWithPassphrase:`, confirming it is indeed a decoding and decryption routine (Listing 9-8).

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

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

  return rax;
}
```

Listing 9-8: The yoop: method (WindTail)

Another approach to locating decryption routines is to peruse the disassembly for calls into system crypto routines (like `CCCrypt`) and well-known crypto constants (such as AES's `s-boxes`). In certain disassemblers, third-party plug-ins such as FindCrypt[1] can automate this crypto discovery process.

String Deobfuscation via a Hopper Script

The downside to the breakpoint-based approach is that it only allows you to recover specific decrypted strings. If an encrypted string is exclusively referenced in a block of code that isn't executed, you'll never encounter its decrypted value. A more comprehensive approach is to re-implement the malware's decryption routine and then pass in all the malware's encrypted strings to recover their plaintext values.

In Chapter 6, we introduced disassemblers, highlighting how they can be leveraged to statically analyze compiled binaries. Such disassemblers also generally support external third-party scripts or plug-ins that can directly interact with a binary's disassembly. This capability is extremely useful and can extend the functionality of a disassembler, especially in the context of overcoming malware's anti-static-analysis efforts. As an example of this, we'll create a Python-based Hopper script capable of decrypting all the embedded strings in a sophisticated malware sample.

DoubleFantasy is the notorious Equation APT Group's first-stage implant, capable of surveying an infected host and installing a persistent second-stage implant on systems of interest. The majority of its strings are encrypted, and many remain encrypted even while the malware is executed unless certain prerequisites, such as specific tasking, are met. However, as the embedded string decryption algorithm is fairly simple, we can re-implement it in a Hopper Python script to decrypt all of the malware's strings.

Looking at the disassembly of the DoubleFantasy malware, we can see what appears to be an encrypted string and its length (`0x38`) being stored to the stack prior to a call into an unnamed subroutine (Listing 9-9):

```
0x00007a93    mov       dword [esp+0x8], 0x38
0x00007a9b    lea       eax, dword [ebx+0x105a7]    ;"\xDA\xB3\...\x14"
0x00007aa1    mov       dword [esp+0x4], eax
0x00007aa5    call      sub_d900
```

Listing 9-9: An encrypted string, and a call to a possible string-decryption function (DoubleFantasy)

An examination of this subroutine reveals it decrypts a passed-in string by running it through a simple XOR algorithm. As shown in the following snippet of disassembly (Listing 9-10), the algorithm uses two keys:

```
0x0000d908    mov       eax, dword [ebp+arg_4]
❶ 0x0000d90b    movzx     edi, byte [eax]
...
0x0000d930    movzx     edx, byte [esi]
0x0000d933    inc       esi
0x0000d934    mov       byte [ebp+var_D], dl
0x0000d937    mov       eax, edx
0x0000d939    mov       edx, dword [ebp+arg_0]
0x0000d93c    xor       eax, edi 1
0x0000d93e    xor       eax, ecx
❷ 0x0000d940    xor       eax, 0x47
0x0000d943    mov       byte [edx+ecx-1], al
0x0000d947    movzx     eax, byte [ebp+var_D]
0x0000d94b    inc       ecx
0x0000d94c    add       edi, eax
0x0000d94e    cmp       ecx, dword [ebp+var_C]
0x0000d951    jne       loc_d930
```

Listing 9-10: A simple string-decryption algorithm (DoubleFantasy)

The first key is based on the values of the encrypted string itself ❶, while the second is hardcoded to `0x47` ❷. With this understanding of the malware's string decryption algorithm, we can trivially re-implement it in Python (Listing 9-11):

```
def decrypt(encryptedStr):
   ...
❶ key_1 = encryptedStr[0]
   key_2 = 0x47

   for i in range(1, len(encryptedStr)):
   ❷ byte = (encryptedStr[i] ^ key_1 ^ i ^ key_2) & 0xFF
     decryptedStr.append(chr(byte))

     key_1 = encryptedStr[i] + key_1

❸ return ''.join(decryptedStr)
```

Listing 9-11: A re-implementation of DoubleFantasy's string decryption algorithm in Python

In our Python re-implementation of the malware's decryption routine, we first initialize both XOR keys ❶. Then we simply iterate over each byte of the encrypted string, de-XORing each with both keys ❷. The decrypted string is then returned ❸.

With the malware's decryption algorithm re-implemented, we now need to invoke it on all of the malware's embedded encrypted strings. Luckily, Hopper makes this fairly straightforward. DoubleFantasy's encrypted strings are all stored in its `_cstring` segment. Using the Hopper APIs made available to any Hopper script, we programmatically iterate through this segment,

invoking the re-implemented decryption algorithm on each string. We add the logic in Listing 9-12 to our Python code to accomplish this.

```
#from start to end of cString segment
#extract/decrypt all strings
i = cSectionStart
while i < cSectionEnd:

   #skip if item is just a 0x0
   if 0 == cSegment.readByte(i):
      i += 1
      continue

   stringStart = i
   encryptedString = []
   while (0 != cSegment.readByte(i)): ❶
      encryptedString.append(cSegment.readByte(i))
      i += 1

      decryptedString = decryptStr(encryptedString) ❷
      if decryptedString.isascii(): ❸

         print(decryptedString)

         #add as inline comment and to all references ❹
         doc.getCurrentSegment().setInlineCommentAtAddress(stringStart, decryptedString)

         for reference in cSegment.getReferencesOfAddress(stringStart):
            doc.getCurrentSegment().setInlineCommentAtAddress(reference, decryptedString)
```

Listing 9-12: Leveraging the Hopper API to decrypt embedded strings (DoubleFantasy)

In this listing, we iterate through the `_cstring` segment and find any null-terminated items, which includes the malware's embedded encrypted strings ❶. For each of these items, we invoke our decryption function on it ❷. Finally, we check if the item decrypted to a printable ASCII string ❸. This check ensures we ignore other items found within the `_cstring` segment that are not encrypted strings. The decrypted string is then added as an inline comment directly into the disassembly, both at the location of the encrypted string and at any location where it is referenced in code to facilitate continuing analysis ❹.

After executing our decryption script in Hopper's Script menu, the strings are decrypted and the disassembly is annotated. For example, as you can see in Listing 9-13, the string `"\xDA\xB3\...\x14"` decrypts to */Library/Caches/com.apple.LaunchServices-02300.csstore*, which turns out to be the hard-coded path of the malware's configuration file.

```
0x00007a93    mov       dword [esp+0x8], 0x38
0x00007a9b    lea       eax, dword [ebx+0x105a7] ; "/Library/Caches/com.apple.LaunchServices
                                                     -02300.csstore, \xDA\xB3\...\x14"
0x00007aa1    mov       dword [esp+0x4], eax
0x00007aa5    call      sub_d900
```

Listing 9-13: Disassembly, now annotated with the decrypted string (DoubleFantasy)

Forcing the Malware to Execute Its Decryption Routine

Creating disassembly scripts to facilitate analysis is a powerful approach. However, in the context of string decryptions, it requires that you both fully understand the decryption algorithm and are capable of re-implementing it. This can often be a time-consuming endeavor. In this section we'll look at a potentially more efficient approach, especially for samples that implement complex decryption algorithms.

A malware specimen is almost certainly designed to decrypt all its strings; 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 directly invoke the malware's string decryption routine for all encrypted strings, all without having to understand the internals of the decryption algorithm. Let's walk through this process using the EvilQuest malware as our target.

First, we note that EvilQuest's binary, named *patch*, appears to contain many obfuscated strings (Listing 9-14):

```
% strings - EvilQuest/patch
Host: %s
ERROR: %s
1PnYzO1rdaiCOOOOO13
1MNsh21anlz9O6WugB2zwfjnOOOOO83
2Uy5DI3hMp7oOcq|T|14vHRzOOOOO13
3mTqdG3tFoV51KYxgy38orxyOOOOO83
OJVurl1WtxB53WxvoP18ouUM2Qo51c3v5dDiOOOOO83
2WVZmB2oRkhr1Y7s1D2asm{v1Al5AT33Xn3XOOOOO53
3iHMvKORFoOr3KGWvD28URSuO6OhV61tdkOt22nizO3nao1qOOOOO33
...
```

Listing 9-14: Obfuscated strings (EvilQuest)

Statically analyzing EvilQuest for a function that takes the obfuscated strings as input quickly reveals the malware's deobfuscation (decryption) logic, found in a function named `ei_str` (Listing 9-15):

```
lea     rdi, "OhC|h71FgtPJ32afft3EzOyU3xFA7qO{LBxN3vZ"...
call    ei_str
...
lea     rdi, "OhC|h71FgtPJ19|69cOm4GZL1xMqqS3kmZbz3FW"...
call    ei_str
```

Listing 9-15: Invocation of a deobfuscation function, `ei_str` (EvilQuest)

The `ei_str` function is rather long and complicated, so instead of trying to decrypt the strings solely via a static analysis approach, we'll opt for a dynamic approach. Moreover, as many of the strings are only deobfuscated at runtime under certain circumstances, such as when a specific command is received, we'll inject a custom library into the code instead of leveraging a debugger.

Our custom injectable library will perform two tasks. First, within a running instance of the malware, it will resolve the address of the deobfuscation function, `ei_str`. Then it will invoke the `ei_str` function for all encrypted strings found embedded within the malware's binary. Because we place this logic in the constructor of the dynamic library, it will be executed when the library is loaded, well before the malware's own code is run.

Listing 9-16 shows the code we'll write for the constructor of the injectable dynamic decryptor 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);
}
```

Listing 9-16: Our dynamic string deobfuscator library (EvilQuest)

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 it's compiled (`% clang decryptor.m -dynamiclib -framework Foundation -o decryptor.dylib`), we can coerce the malware to load our decryptor library via the `DYLD_INSERT_LIBRARIES` environment variable. In the terminal of a virtual machine, we can execute the following command:

```
% DYLD_INSERT_LIBRARIES=<path to dylib> <path to EvilQuest>
```

Once loaded, the library's code is automatically invoked and coerces the malware to decrypt all its strings (Listing 9-17):

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib EvilQuest/patch

decrypted string (0x10eb675ec): andrewka6.pythonanywhere.com

decrypted string (0x10eb67a95): *id_rsa*/i
decrypted string (0x10eb67c15): *key*.png/i
decrypted string (0x10eb67c35): *wallet*.png/i
decrypted string (0x10eb67c55): *key*.jpg/i

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>
```

Listing 9-17: Deobfuscated strings (EvilQuest)

The decrypted output (abridged) reveals informative strings that appear to show a potential command and control server, files of interest, and a template for launch item persistence.

If the malware is compiled with a hardened runtime, the dynamic loader will ignore the `DYLD_INSERT_LIBRARIES` variable and fail to load our deobfuscator. To bypass this protection, you can first disable System Integrity Protection (SIP) and then execute the following command to set the `amfi_get_out_of_my_way` boot argument and then reboot your analysis system (or virtual machine):

```
# nvram boot-args="amfi_get_out_of_my_way=0x1"
```

For more information on this topic, see "How to Inject Code into Mach-O Apps, Part II."[2]

Code-Level Obfuscations

To further protect their creations from analysis, malware authors may also turn toward broader code-level obfuscations. For malicious scripts, which are otherwise easy to analyze, as they are not compiled into binary code, this sort of obfuscation is quite common. As we discussed in Chapter 4, we can often leverage tools such as beautifiers to improve the readability of obfuscated scripts. Obfuscated Mach-O binaries are somewhat less common, but we'll look at several examples of this technique.

One such obfuscation method involves adding *spurious*, or *garbage*, instructions at compile time. These instructions are essentially non-operations (NOPs) and have no impact on the core functionality of the malware. However, when spread effectively throughout the binary, they can mask the malware's real instructions. The prolific Pirrit malware provides an example of such binary obfuscation. To hinder static analysis and hide other logic aimed at preventing dynamic analysis, its authors added large amounts of garbage instructions. In the case of Pirrit, these instructions make up either calls into system APIs (whose results are ignored), bogus control flow blocks, or inconsequential modifications to unused memory. The following is an example of the former, in which we see the `dlsym` API being invoked. This API is normally invoked to dynamically resolve the address of a function by name. In Listing 9-18, the decompiler has determined the results are unused:

```
dlsym(dlopen(0x0, 0xa), 0x100058a91);
dlsym(dlopen(0x0, 0xa), 0x100058a80);
dlsym(dlopen(0x0, 0xa), 0x100058a64);
dlsym(dlopen(0x0, 0xa), 0x100058a50);
dlsym(dlopen(0x0, 0xa), 0x100058a30);
dlsym(dlopen(0x0, 0xa), 0x100058a10);
dlsym(dlopen(0x0, 0xa), 0x1000589f0);
```

Listing 9-18: Spurious function calls (Pirrit)

Elsewhere in Pirrit's decompilation, we find spurious code control blocks whose logic is not relevant to the core functionality of the malware. Take, for instance, Listing 9-19, which contains several pointless comparisons of the `RAX` register. (The final check can only evaluate to true if `RAX` is equal to `0x6b1464f0`, so the first two checks are entirely unnecessary.) Following this is a large sequence of instructions that modify a section of the binary's memory, which is otherwise unused:

```
if (rax != 0x6956b086) {
   if (rax != 0x6ad066c0) {
       if (rax == 0x6b1464f0) {
       *(int8_t *)byte_1000589fa = var_29 ^ 0x37;
       *(int8_t *)byte_1000589fb = *(int8_t *)byte_1000589fb ^ 0x9a;
       *(int8_t *)byte_1000589fc = *(int8_t *)byte_1000589fc ^ 0xc8;
       *(int8_t *)byte_1000589fd = *(int8_t *)byte_1000589fd ^ 0xb2;
       *(int8_t *)byte_1000589fe = *(int8_t *)byte_1000589fe ^ 0x15;
       *(int8_t *)byte_1000589ff = *(int8_t *)byte_1000589ff ^ 0x78;
       *(int8_t *)byte_100058a00 = *(int8_t *)byte_100058a00 ^ 0x1d;
       ...
       *(int8_t *)byte_100058a20 = *(int8_t *)byte_100058a20 ^ 0x69;
       *(int8_t *)byte_100058a21 = *(int8_t *)byte_100058a21 ^ 0xab;
       *(int8_t *)byte_100058a22 = *(int8_t *)byte_100058a22 ^ 0x02;
       *(int8_t *)byte_100058a23 = *(int8_t *)byte_100058a23 ^ 0x46;
```

Listing 9-19: Spurious instructions (Pirrit)

In almost every subroutine in Pirrit's disassembly, we find massive amounts of such garbage instructions. Though they do slow down our analysis and initially mask the malware's true logic, once we understand their purpose, we can simply ignore them and scroll past. For more information on this and other similar obfuscation schemes, you can read "Using LLVM to Obfuscate Your Code During Compilation."[3]

Bypassing Packed Binary Code

Another common way to obfuscate binary code is with a packer. In a nutshell, a *packer* compresses binary code to prevent its static analysis while also 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, retaining the binary's original functionality.

Packers are payload-agnostic and thus can generally pack any binary. This means that legitimate software can also be packed, as software developers occasionally seek to thwart analysis of their proprietary code. Thus, we can't assume any packed binary is malicious without further analysis.

The well-known UPX packer is a favorite among both Windows and macOS malware authors.[4] Luckily, unpacking UPX-packed files is easy. You can simply execute UPX with the `-d` command line flag (Listing 9-20). If you'd like to write the unpacked binary to a new file, use the `-o` flag as well.

```
% 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.
```

Listing 9-20: Unpacking via UPX (ColdRoot)

As you can see, we've unpacked a UPX-packed variant: the malware known as ColdRoot. Once it's unpacked and decompressed, we can commence static and dynamic analysis.

Here is a valid question: How did we know the sample was packed? And how did we know it was packed with UPX specifically? One semiformal approach to figuring out which binaries are packed 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. I've added code to the Objective-See TaskExplorer utility to generically detect packed binaries in this manner.[5]

A less formal approach is to leverage the `strings` command or load the binary in your disassembler of choice and peruse the code. With experience, you'll be able to infer that a binary is packed if you observe the following:

- Unusual section names
- A majority of strings obfuscated
- Large chunks of executable code that cannot be disassembled
- A low number of imports (references to external APIs)

Unusual section names are an especially good indicator, as they can also help identify the packer used to compress the binary. For example, UPX adds a section named `__XHDR`, which you can see in the output of the `strings` command or in a Mach-O viewer (Figure 9-3).

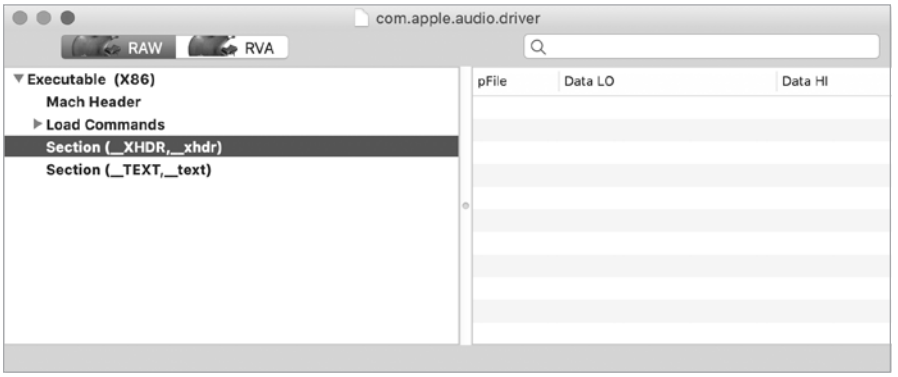


Figure 9-3: UPX section header (ColdRoot)

It is worth noting that UPX is an exception among packers in the sense that it can unpack any UPX-packed binary. More sophisticated malware may leverage custom packers, which may mean that you have no unpacking utility available. Not to worry: if you encounter a packed binary and have 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, dump the unprotected binary from memory with the `memory read` LLDB command.

For another thorough discussion of both analyzing other packers (such as MPRESS) and the process of dumping a packed binary from memory, see Pedro Vilaça's informative 2014 talk, "F*ck You HackingTeam."[6]

Decrypting Encrypted Binaries

Similar to packers are *binary encryptors*, which encrypt the original malware code at the binary level. To automatically decrypt the malware at runtime, the encryptor will often insert a decryptor stub and keying information at the start of the binary unless the operating system natively

supports encrypted binaries, which macOS does. As noted, the infamous HackingTeam is fond of packers and encryptors. In the blog post "HackingTeam Reborn . . ." I noted that the installer for the HackingTeam's macOS implant, RCS, leveraged Apple's proprietary and undocumented Mach-O encryption scheme in an attempt to thwart static analysis.[7]

Let's take a closer look at how to decrypt binaries, such as HackingTeam's installer, that have been protected via this method. In macOS's open source Mach-O loader, we find an `LC_SEGMENT` flag value named `SG_PROTECTED_VERSION_1` whose value is `0x8`:[8]

```
#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. */
```

Comments show that this flag specifies that a Mach-O segment is encrypted (or "protected," in Apple parlance). Via `otool`, we can parse the embedded Mach-O loader commands in HackingTeam's installer and note that, indeed, the flag's value within the `__TEXT` segment (the segment that contains the binary's executable instructions) is set to the value of `SG_PROTECTED_VERSION_1` (Listing 9-21):

```
% otool -l HackingTeam/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
```

Listing 9-21: An encrypted installer; note that the `flags` field is set to `0x8`, `SG_PROTECTED_VERSION_1` (HackingTeam)

From the macOS loader's source code, we can see that the `load_segment` function checks the value of this flag.[9] If the flag is set, the loader will invoke a function named `unprotect_dsmos_segment` to decrypt the segment, as in Listing 9-22:

```
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;
            }
}
```

Listing 9-22: macOS's support of encrypted Mach-O binaries

Continued analysis reveals that the encryption scheme is symmetric (either Blowfish or AES) and uses a static key that is stored within the Mac's System Management Controller. As such, we can write a utility to decrypt any binary protected in this manner. For more discussion of this macOS encryption scheme, see Erik Pistelli's blog post "Creating undetected malware for OS X."[10]

Another option for recovering the malware's unencrypted instructions is to dump the unprotected binary code from memory once the decryption code has executed. For this specific malware specimen, its unencrypted code can be found from address `0x7000` to `0xbffff`. The following debugger command will save its unencrypted code to disk for static analysis:

```
(lldb) memory read --binary --outfile /tmp/dumped.bin 0x7000 0xbffff --force
```

Note that due to the large memory range, the `--force` flag must be specified as well.

I've shown that dynamic analysis environments and tools are generally quite successful against anti-static-analysis approaches. As a result, 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 whether it is executing in a dynamic analysis environment like a virtual machine or within a dynamic analysis tool like a debugger.

Malware may leverage several common approaches to detecting dynamic analysis environments and tools:

- **Virtual machine detection**: Often, malware analysts will execute the suspected malicious code within an isolated virtual machine in order to monitor it or perform dynamic analysis. Malware, therefore, is probably right to assume that if it finds itself executing within a virtual machine, it is likely being closely watched or dynamically analyzed. Thus, malware often seeks to detect if it's running in a virtualized environment. Generally, if it detects such an environment, it 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. In an attempt to prevent analysis, it will likely prematurely exit. Alternatively, it might attempt to prevent debugging in the first place.

How can we figure out whether a malicious specimen contains anti-analysis logic to thwart dynamic analysis? Well, if you're attempting to dynamically analyze a malicious sample in a virtual machine or debugger, and the sample prematurely exits, this may be a sign that it implements anti-analysis logic. (Of course, there are other reasons malware might exit; for example, it might detect that its command and control server is offline.)

If you suspect that the malware contains such logic, the first goal should be to uncover the specific code that is responsible for this behavior. Once you've identified it, you can bypass this code by patching it out or simply skipping it in a debugger session. One effective way to uncover a sample's anti-analysis code is using static analysis, which means you have to know what this anti-analysis logic might look like. The following sections describe various programmatic methods that malware can leverage to detect if it is executing within a virtual machine or a debugger. Recognizing these approaches is important, as many are widespread and found within unrelated Mac malware specimens.

Checking the System Model Name

Malware may check if it's running within a virtual machine by querying the machine's name. The macOS ransomware named MacRansom performs such a check. Take a look at the following snippet of decompiled code, which corresponds to the malware's anti-virtual-machine check. Here, after decoding a command, the malware invokes the `system` API to execute it. If the API returns a nonzero value, the malware will prematurely exit (Listing 9-23):

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

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

Listing 9-23: Obfuscated anti-VM logic (MacRansom)

To uncover the command executed by the malware, we can leverage a debugger. Specifically, by setting a breakpoint on the `system` API function, we can dump the decoded command. As it is passed as an argument to

system, as shown in the debugger output in Listing 9-24, this command can be found in the RDI register:

```
(lldb) b system
Breakpoint 1: where = libsystem_c.dylib`system, address = 0x00007fff67848fdd
(lldb) c

Process 1253 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00007fff67848fdd libsystem_c.dylib`system
libsystem_c.dylib`system:
->  0x7fff67848fdd <+0>: pushq  %rbp

(lldb) x/s $rdi
0x100205350: "sysctl hw.model|grep Mac > /dev/null" ❶
```

Listing 9-24: Deobfuscated anti-VM command (MacRansom)

Turns out the command ❶ first retrieves the system's model name from hw.model and then checks to see if it contains the string Mac. In a virtual machine, this command will return a nonzero value, as the value for hw.model will not contain Mac but rather something similar to VMware7,1 (Listing 9-25):

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

Listing 9-25: System's hardware model (in a virtual machine)

On native hardware (outside of a virtual machine), the sysctl hw.model command will return a string containing Mac and the malware will not exit (Listing 9-26):

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

Listing 9-26: System's hardware model (on native hardware)

Counting the System's Logical and Physical CPUs

MacRansom contains another check to see if it is running in a virtual machine. Again, the malware decodes a command, executes it via the system API, and prematurely exits if the return value is nonzero. Here is the command it executes:

```
echo $((`sysctl -n hw.logicalcpu`/`sysctl -n hw.physicalcpu`))|grep 2 > /dev/null
```

This command checks the number of logical CPUs divided by the number of physical CPUs on the system where the malware is executing. On a virtual machine, this value is often just 1. If it isn't 2, the malware will exit. On native hardware, dividing the number of logical CPUs by the number of physical CPUs will often (but not always!) result in a value of 2, in which case the malware will happily continue executing.

Checking the System's MAC Address

Another Mac malware sample that contains code to detect if it is running in a virtual machine is Mughthesec, which masquerades as an Adobe Flash installer. If it detects that it is running within a virtual machine, the installer doesn't do anything malicious; it merely installs a legitimate copy of Flash. Security researcher Thomas Reed noted that this virtual machine detection is done by examining the system's MAC address.

If we disassemble the malicious installer, we find the snippet of code responsible for retrieving the system's MAC address via the I/O registry (Listing 9-27):

```
❶ r14 = IOServiceMatching("IOEthernetInterface");
if (r14 != 0x0) {
  rbx = CFDictionaryCreateMutable(...);
  if (rbx != 0x0) {
    CFDictionarySetValue(rbx, @"IOPrimaryInterface", **_kCFBooleanTrue);
    CFDictionarySetValue(r14, @"IOPropertyMatch", rbx);
    CFRelease(rbx);
  }
}
...
rdx = &var_5C0;
if (IOServiceGetMatchingServices(r15, r14, rdx) == 0x0) {
  ...
  r12 = var_5C0;
  rbx = IOIteratorNext(r12);
  r14 = IORegistryEntryGetParentEntry(rbx, "IOService", rdx);
  if (r14 == 0x0) {
    rdx = **_kCFAllocatorDefault;
  ❷ r15 = IORegistryEntryCreateCFProperty(var_35C, @"IOMACAddress", rdx, 0x0);
```

Listing 9-27: Retrieving the primary MAC address (Mughthesec)

The malware first creates an iterator containing the primary Ethernet interface by invoking APIs such as `IOServiceMatching` with the string `"IOEthernetInterface"` ❶. Using this iterator, it then retrieves the MAC address ❷. Note that this code is rather similar to Apple's "GetPrimaryMACAddress" sample code, which demonstrates how to programmatically retrieve the device's primary MAC address.[11] This is not surprising, as malware authors often consult (or even copy and paste) Apple's sample code.

MAC addresses contain an *organizationally unique identifier (OUI)* that maps to a specific vendor. If malware detects a MAC address with an OUI matching a virtual machine vendor such as VMware, it knows it is running within a virtual machine. Vendors' OUIs can be found online, such as on company websites. For example, online documentation found at *https://docs.vmware.com/* notes that VMware's OUI ranges include `00:50:56` and `00:0C:29`, meaning that for the former, VMware VMs will contain MAC addresses in the following format: `00:50:56:XX:YY:ZZ`.[12]

Of course, there are a myriad of other ways for malware to programmatically detect if it is executing within a virtual machine. For a fairly comprehensive list of such methods, see "Evasions: macOS."[13]

Checking System Integrity Protection Status

Of course, not all analysis is done within virtual machines. Many malware analysts leverage dedicated analysis machines to dynamically analyze malicious code. In this scenario, as the analysis is performed on native hardware, anti-analysis logic that is based on detecting virtual machines is useless. Instead, malware must look for other indicators to determine if it's running within an analysis environment. One such approach is to check the status of *System Integrity Protection (SIP)*.

SIP is a built-in macOS protection mechanism that, among other things, may prevent the debugging of processes. Malware analysts, who often require the ability to debug any and all processes, will often disable SIP on their analysis machines. The prolific Pirrit malware leverages this fact to check whether it's likely running on an analysis system. Specifically, it will execute macOS's `csrutil` command to determine the status of SIP. We can observe this passively via a process monitor, or more directly in a debugger. In the case of the latter, we can break on a call to the `NSConcreteTask`'s `launch` method and dump the launch path and arguments of the task object (found in the `RDI` register), as shown in Listing 9-28:

```
(lldb) po [$rdi launchPath]
/bin/sh

(lldb) po [$rdi arguments]
<__NSArrayI 0x10580dfd0>(
 -c,
 command -v csrutil > /dev/null && csrutil status |
 grep -v "enabled" > /dev/null && echo 1 || echo 0
)
```

Listing 9-28: Retrieving the System Integrity Protection status (Pirrit)

From the debugger output, we can confirm that indeed the malware is executing the `csrutil` command (via the shell, `/bin/sh`) with the `status` flag. The output of this command is passed to `grep` to check if SIP is still enabled. If SIP has been disabled, the malware will prematurely exit in an attempt to prevent continued dynamic analysis.

Detecting or Killing Specific Tools

Malware might also contain anti-analysis code to detect and thwart dynamic analysis tools. As you'll see, this code usually focuses on debugger detection, but some malware specimens will also take into account other analysis or security tools that might detect the malware and alert the user, which is something malware often seeks to avoid at all costs.

A variant of the malware known as Proton looks for specific security tools. When executed, the Proton installer will query the system to see if any third-party firewall products are installed. If any are found, the malware chooses not to infect the system and simply exits. This is illustrated

in the following snippet of decompiled code extracted from the installer (Listing 9-29):

```
❶ rax = [*0x10006c4a0 objectAtIndexedSubscript:0x51];

  rdx = rax;
❷ if ([rbx fileExistsAtPath:rdx] != 0x0) goto fileExists;

  fileExists:
  rax = exit(0x0);
```

Listing 9-29: Basic firewall detection (Proton)

The installer first extracts a filepath from a decrypted array ❶. Dynamic analysis reveals that this extracted path points to the kernel extension of Little Snitch, a popular third-party firewall: */Library/Extensions/LittleSnitch.kext*. If this file is found on the system the malware is about to infect, installation is aborted ❷.

The Proton installer has other tricks up its sleeve. For example, in an attempt to thwart dynamic analysis, it will terminate tools such as the macOS's log message collector (the Console application) and the popular network monitor Wireshark. To terminate these applications, it simply invokes the built-in macOS utility, `killall`. Though rather primitive and quite noticeable, this technique will prevent the analysis tools from running alongside the malware. (Of course, the tools can simply be restarted, or even just renamed.)

Detecting a Debugger

The debugger is arguably the most powerful tool in the malware analyst's arsenal, so most malware that contains anti-analysis code seeks to detect whether it is running in 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, a process should first invoke the `sysctl` API with `CTL_KERN`, `KERN_PROC`, `KERN_PROC_PID`, and its process identifier (`pid`), as parameters. Also, a `kinfo_proc` structure should be provided.[14] The `sysctl` function will then populate the structure with information about the process, including a `P_TRACED` flag. If set, this flag means the process is currently being debugged. Listing 9-30, taken directly from Apple's documentation, checks for the presence of a debugger in this manner:

```
static bool AmIBeingDebugged(void)
    // Returns true if the current process is being debugged (either
    // running under the debugger or has a debugger attached post facto).
{
    int                 junk;
    int                 mib[4];
    struct kinfo_proc   info;
    size_t              size;

    // Initialize the flags so that, if sysctl fails for some bizarre
    // reason, we get a predictable result.
```

```
    info.kp_proc.p_flag = 0;

    // Initialize mib, which tells sysctl the info we want, in this case
    // we're looking for information about a specific process ID.

    mib[0] = CTL_KERN;
    mib[1] = KERN_PROC;
    mib[2] = KERN_PROC_PID;
    mib[3] = getpid();

    // Call sysctl.

    size = sizeof(info);
    junk = sysctl(mib, sizeof(mib) / sizeof(*mib), &info, &size, NULL, 0);
    assert(junk == 0);

    // We're being debugged if the P_TRACED flag is set.

    return ( (info.kp_proc.p_flag & P_TRACED) != 0 );
}
```

Listing 9-30: Debugger detection (via the `P_TRACED` flag)

Malware will often use this same technique, in some cases copying Apple's code verbatim. This was the case with the Russian malware known as Komplex. Looking at a decompilation of Komplex's main function, you can see that it invokes a function named `AmIBeingDebugged` (Listing 9-31):

```
int main(int argc, char *argv[]) {
...
   if ((AmIBeingDebugged() & 0x1) == 0x0) {

   //core malicious logic

   }
   else {
    remove(argv[0]);
}

return 0;
```

Listing 9-31: Debugger detection (Komplex)

If the `AmIBeingDebugged` function returns a nonzero value, the malware will execute the logic in the `else` block, which causes the malware to delete itself in an attempt to prevent continued analysis. And as expected, if we examine the code of the malware's `AmIBeingDebugged` function, it is logically equivalent to Apple's debugger detection function.

Preventing Debugging with ptrace

Another anti-debugging approach is attempting to prevent debugging altogether. Malware can accomplish this by invoking the `ptrace` system call with the `PT_DENY_ATTACH` flag. This Apple-specific flag prevents a debugger from

attaching and tracing the malware. Attempting to debug a process that invokes ptrace with the PT_DENY_ATTACH flag will fail (Listing 9-32):

```
% lldb proton
...

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

Listing 9-32: A premature exit due to ptrace with the PT_DENY_ATTACH flag (Proton)

You can tell the malware has the PT_DENY_ATTACH flag set because it prematurely exits with a status of 45.

Calls to the ptrace function with the PT_DENY_ATTACH flag are fairly easy to spot (for example, by examining the binary's imports). Thus, malware may attempt to obfuscate the ptrace call. For example, Proton dynamically resolves the ptrace function by name, preventing it from showing up as an import, as you can see in the following snippet (Listing 9-33):

```
0x0000000010001e6b8    xor       edi, edi
0x0000000010001e6ba    mov       esi, 0xa
0x0000000010001e6bf    call    ❶ dlopen
0x0000000010001e6c4    mov       rbx, rax
0x0000000010001e6c7    lea       rsi, qword [ptrace]
0x0000000010001e6ce    mov       rdi, rbx
0x0000000010001e6d1    call    ❷ dlsym
0x0000000010001e6d6    mov       edi, ❸0x1f
0x0000000010001e6db    xor       esi, esi
0x0000000010001e6dd    xor       edx, edx
0x0000000010001e6df    xor       ecx, ecx
0x0000000010001e6e1    call      rax
```

Listing 9-33: Obfuscated anti-debugger logic via ptrace, PT_DENY_ATTACH (Proton)

After invoking the dlopen function ❶, the malware calls dlsym ❷ to dynamically resolve the address of the ptrace function. As the dlsym function takes a pointer to the string of the function to resolve, such as [ptrace], that function won't show up as a dependency of the binary. The return value from dlsym, stored in the RAX register, is the address of ptrace. Once the address is resolved, the malware promptly invokes it, passing in 0x1F, which is the hexadecimal value of PT_DENY_ATTACH ❸. If the malware is being debugged, the call to ptrace will cause the debugging session to forcefully terminate and the malware to exit.

Bypassing Anti-Dynamic-Analysis Logic

Luckily, the anti-dynamic-analysis methods covered thus far are all fairly trivial to bypass. Overcoming most of these tactics involves two steps: identifying the location of the anti-analysis logic and then preventing its execution. Of these two steps, the first is usually the most challenging, but

it becomes far easier once you're familiar with the anti-analysis methods discussed in this chapter.

It's wise to first statically triage a binary before diving into a full-blown debugging session. During this triage, keep an eye out for telltale signs that may reveal dynamic-analysis-thwarting logic. For example, if a binary imports the `ptrace` API, there is a good chance it will attempt to prevent debugging with the `PT_DENY_ATTACH` flag.

Strings or function and method names may also reveal a malware's distaste for analysis. For example, running the `nm` command, used to dump symbols, against EvilQuest reveals functions named `is_debugging` and `is_virtual_mchn` (Listing 9-34):

```
% nm EvilQuest/patch
...

0000000100007aa0 T _is_debugging
0000000100007bc0 T _is_virtual_mchn
```

Listing 9-34: Anti-analysis functions? (EvilQuest)

Unsurprisingly, continued analysis reveals that both functions are related to the malware's anti-analysis logic. For example, examining the code that invokes the `is_debugging` function reveals that EvilQuest will prematurely exit if the function returns a nonzero value; that is, if a debugger is detected (Listing 9-35):

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

Listing 9-35: Anti-debugging logic (EvilQuest)

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

This trial and error approach could be conducted in the following manner:

1. Start a debugger session that executes the malicious sample. It is important to start the debugging session at the very beginning rather than attaching it to the already running process. This ensures that the malware has not had a chance to execute any of its anti-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` and `ptrace`.
3. Instead of allowing the malware 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 for anti-analysis reasons. For example, check for `ptrace` invoked with the `PT_DENY_ATTACH` flag, or perhaps `sysctl` 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.
4. If stepping over a function call causes the malware to exit (a sign it likely detected either the virtual machine or the debugger), restart the debugging session and, this time, step into this function. Repeat this process until you've identified the location of the anti-analysis logic.

Armed with the locations of the anti-analysis logic, you can now bypass it by modifying the execution environment, patching the on-disk binary image, modifying program control flow in a debugger, or modifying the register or variable value in a debugger. Let's briefly look at each of these methods.

Modifying the Execution Environment

It may be possible to modify the execution environment such that the anti-analysis logic no longer triggers. Recall that Mughthesec contains logic to detect if it's running within a virtual machine by examining the system's MAC address. If the malware detects a MAC address with an OUI matching a virtual machine vendor such as VMware, it won't execute. Luckily, we can modify our MAC address in the virtual machine's settings, choosing an address that falls outside the range of any virtual machine provider's OUI. For example, set it to the OUI of your base macOS machine, like `F0:18:98`, which belongs to Apple. Once the MAC address has been changed, Mughthesec will no longer detect the environment as a virtual machine and so will happily execute its malicious logic, allowing our dynamic analysis to continue.

Patching the Binary Image

Another more permanent approach to bypassing anti-analysis logic involves patching the malware's on-disk binary image. The Mac ransomware KeRanger is a good candidate for this approach, as it may sleep for several days before executing its malicious payload, perhaps in an effort to impede automated or dynamic analysis.

Though the malware is packed, it leverages the UPX packer, which we can fully unpack using the `upx -d` command. Next, static analysis can identify the function aptly named `waitOrExit` that is responsible for implementing the wait delay. It is invoked by the `startEncrypt` function, which begins the process of ransoming users' files:

```
startEncrypt:
...
```

```
0x0000000100002390    test    eax, eax
0x0000000100002392    je      leave
```

To bypass the delay logic so that the malware will immediately continue execution, we can modify the malware's binary code to skip the call to the `waitOrExit` function.

In a hex editor, we change the bytes of the malware's executable instructions from a `call` to a `nop`. Short for "no operation," a `nop` is an instruction (`0x90` on Intel platforms) that instructs the CPU to do, well, 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 overwritten call failed (Listing 9-36):

```
startEncrypt:
...
0x000000010000238b    nop
0x000000010000238c    nop
0x000000010000238d    nop
...
0x0000000100002396    nop
0x0000000100002397    nop
```

Listing 9-36: Anti-analysis logic, now `nop`'d out (KeRanger)

Now whenever this modified version of KeRanger is executed, the `nop` instructions will 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 a permanent solution, 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 of its cryptographic signatures. This could prevent the malware from executing successfully. Thus, it's more common for malware analysts to use a debugger or other runtime method, such as injecting a custom library, to circumvent anti-dynamic-analysis logic.

Modifying the Malware's Instruction Pointer

One of the more powerful capabilities of a debugger is its ability to directly modify the entire state of the malware. This capability proves especially useful when you need to bypass dynamic-analysis-thwarting logic.

Perhaps the simplest way to do so involves manipulating the program's instruction pointer, which points to the next instruction that the CPU will execute. This value is stored in the program counter register, which

on 64-bit Intel systems is the RIP register. You can set a breakpoint on the anti-analysis logic, and when the breakpoint is hit, modify the instruction pointer to, for example, skip over problematic logic. If done correctly, the malware will be none the wiser.

Let's return to KeRanger. After setting a breakpoint on the call instruction that invokes the function that sleeps for three days, we can allow the malware to continue until that breakpoint is hit. At this point, we can simply modify the instruction pointer to 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.

Recall that in a debugger session, you can change the value of any register via the `reg write` debugger command. To specifically modify the value of the instruction pointer, execute this command on the RIP register.

```
(lldb) reg write $rip <new value>
```

Let's walk through another example. The EvilQuest malware contains a function named `prevent_trace` that invokes the `ptrace` API with the `PT_DENY_ATTACH` flag. Code at address `0x0000000100000b8b2` 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 logic, we can avoid the call to `prevent_trace` altogether by setting a breakpoint at `0x0000000100000b8b2`. Once the breakpoint is hit, we modify the value of the instruction pointer to skip the call, as in Listing 9-37:

```
% (lldb) b 0x10000b8b2
Breakpoint 1: where = patch[0x0000000100000b8b2]

(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
```

Listing 9-37: Skipping anti-debugger logic (EvilQuest)

Now the `prevent_trace` function is never invoked, and our debugging session can continue.

Note 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 program may crash. Sometimes, a simpler approach can be taken to avoid manipulating the instruction pointer and modify other registers instead.

Modifying a Register Value

Note that EvilQuest contains a function named `is_debugging`. Recall that the function returns a nonzero value if it detects a debugging session, which will cause the malware to abruptly terminate. Of course, if no debugging session is detected because `is_debugging` returns zero, the malware will happily continue.

Instead of manipulating the instruction pointer, we can set a breakpoint on the instruction that performs the check of the value returned by the `is_debugging` function. Once this breakpoint is hit, the `EAX` register will contain a nonzero value, as the malware will have detected our debugger. However, via the debugger, we can surreptitiously toggle the value in `EAX` to 0 (Listing 9-38):

```
* 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
```

Listing 9-38: Modifying register values to bypass anti-debugging logic

Changing the value of the `EAX` register to 0 (via `reg write $eax 0`) ensures the comparison instruction will now result in the zero flag being set. Thus, the `je` instruction will take the branch to address `0x10000b8b2`, avoiding the call to `exit` at `0x10000b8ad`. Note that we only needed to modify the lower 32 bits of the `RAX` register (`EAX`), as this is all that is checked by the compare instruction (`cmp`).

A Remaining Challenge: Environmentally Generated Keys

At this point, it may seem that malware analysts have the upper hand; after all, no anti-analysis measures can stop us, right? Not so fast. Sophisticated malware authors employ protection encryption schemes that use *environmentally generated keys*. These keys are generated on the victim's system and are thus unique to a specific instance of an infection.

The implications of this are rather profound. If the malware finds itself outside the environment for which it was keyed, it will be unable to decrypt itself. This also means that attempts to analyze the malware will likely fail, as it will remain encrypted. If this environmental protection mechanism is implemented correctly and the keying information is not externally recoverable, the only way to analyze the malware is either by performing the analysis directly on the infected system or by performing it on a memory dump of the malware captured on the infected system.

We've seen this protection mechanism in Windows malware written by the infamous Equation Group, as well as more recently on macOS by the Lazarus Group.[15] The latter encrypted all second-stage payloads with the serial number of the infected systems. For more on the intriguing topic of environmental key generation, see my 2015 Black Hat talk "Writing Bad @$$ Malware for OS X."[16] Also check out James Riordan and Bruce Schneier's seminal paper on the topic, "Environmental Key Generation Towards Clueless Agents."[17]

Up Next

In this chapter, we discussed common anti-analysis approaches that malware may leverage in an attempt to thwart our analysis efforts. After discussing how to identify this logic, I illustrated how to use static and dynamic approaches in order to bypass it. Armed with the knowledge presented in this book thus far, you're now ready to analyze a sophisticated piece of Mac malware. In the next chapter we'll uncover the malware's viral infection capabilities, persistence mechanism, and goals.

Endnotes

1 Ilfak Guilfanov, "FindCrypt2," *Hex-Rays*, February 7, 2006, *https://www.hex-rays.com/blog/findcrypt2/*.

2 Jon Gabilondo, "How to Inject Code into Mach-O Apps, Part II," *Jon Gabilondo* (blog), September 22, 2019, *https://medium.com/@jon.gabilondo.angulo_7635/how-to-inject-code-into-mach-o-apps-part-ii-ddb13ebc8191/*.

3 Yakov Matvienko, "Using LLVM to Obfuscate Your Code During Compilation," *Apriorit Dev Blog*, June 25, 2020, *https://www.apriorit.com/dev-blog/687-reverse-engineering-llvm-obfuscation/*.

4 UPX, *https://upx.github.io/*.

5 TaskExplorer, *https://objective-see.com/products/taskexplorer.html*.

6 Pedro Vilaça, "F*ck You HackingTeam," *https://papers.put.as/papers/macosx/2014/SyScan360-FuckYouHackingTeam.pdf*.

7 Patrick Wardle, "HackingTeam Reborn: A Brief Analysis of an RCS Implant Installer," *Objective-See*, February 26, 2016, *https://objective-see.com/blog/blog_0x0D.html*.

8 "mach-o/loader.h," *Apple*, *https://opensource.apple.com/source/xnu/xnu-7195.141.2/EXTERNAL_HEADERS/mach-o/loader.h.auto.html*.

9 "kern/mach_loader.c," *Apple*, *https://opensource.apple.com/source/xnu/xnu-7195.141.2/bsd/kern/mach_loader.c*.

10 Erik Pistelli, "Creating undetected malware for OS X," *NTCore*, October 7, 2013, *https://ntcore.com/?p=436/*.

11 "GetPrimaryMACAddress," *Apple Developer Documentation Archive*, *https://developer.apple.com/library/archive/samplecode/GetPrimaryMACAddress/Introduction/Intro.html*.

12 "VMware OUI in Static MAC Addresses," *VMware*, May 31, 2019, *https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.networking.doc/GUID-ADFECCE5-19E7-4A81-B706-171E279ACBCD.html*.

13 "Evasions: macOS," *Check Point Research*, *https://evasions.checkpoint.com/techniques/macos.html*.

14 "Technical Q&A QA1361: Detecting the Debugger," *Apple Developer Documentation Archive*, *https://developer.apple.com/library/archive/qa/qa1361/_index.html*.

15 "Equation Group: Questions and Answers," *Kaspersky Lab*, February 2015, *https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/08064459/Equation_group_questions_and_answers.pdf*; Patrick Wardle, "Weaponizing a Lazarus Group Implant," *Objective-See*, February 22, 2020, *https://objective-see.com/blog/blog_0x54.html*.

16 Patrick Wardle, "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*.

17 James Riordan and Bruce Schneier, "Environmental Key Generation Towards Clueless Agents," *Schneier on Security*, *https://www.schneier.com/wp-content/uploads/2016/02/paper-clueless-agents.pdf*.