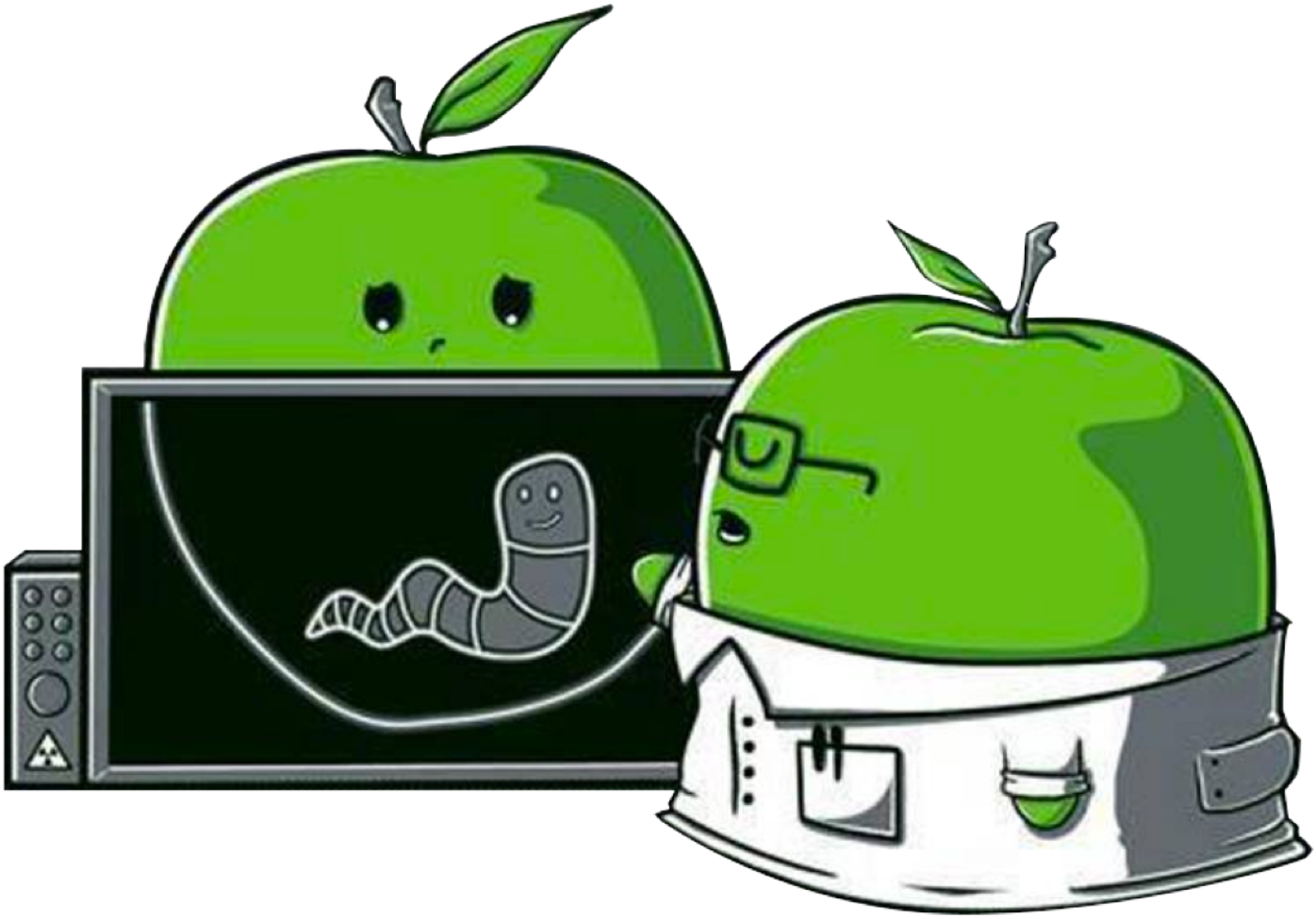


第0x0B章：反分析

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:

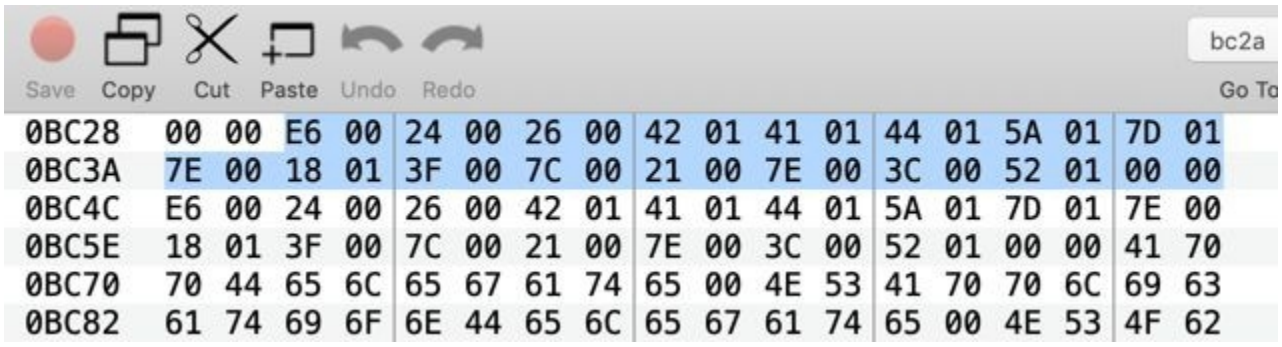
```
r14 = [NSString stringWithFormat:@"%@", [self
yoop:@"F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmGyV7vZdBX2PYYAIfmUcgXHjNZe3ibndAJ
Ah1fA69AHwjVjD0L+Oy/rbhmw9RF/OLs="]];

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：

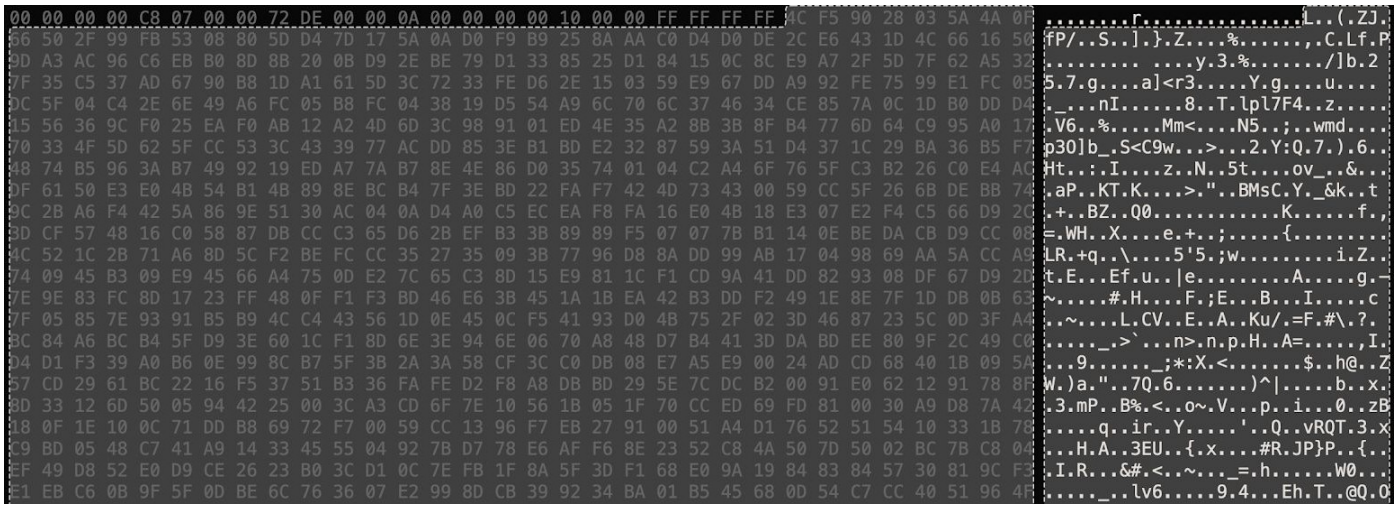
```
$ strings - WindTail/最终演示。app/Contents/MacOS/usrnode

/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！