


第0x7章：反编译和反编译

📝 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
就在文档的边界上）。

 出现在屏幕上的图标

根据定义，**Mach-O**二进制文件是“二进制的”...这意味着，虽然计算机很容易读取，但它们编译的二进制代码并不是为人类直接读取而设计的。

由于绝大多数**Mach-O**恶意软件仅以这种编译的二进制形式“可用”（即其源代码不可用），我们作为恶意软件分析师依赖于能够从此类二进制文件中提取有意义信息的工具。

在前一章中，我们介绍了各种静态分析工具，它们可以帮助对未知的**Mach-O**二进制文件进行分类。然而，如果我们真的想全面了解**Mach-O**二进制文件（例如，一个似乎是Mac恶意软件新部件的样本），就需要其他更复杂的工具。

高级逆向工程工具提供了反汇编、反编译（甚至动态调试）二进制文件的能力。在本章中，我们将坚持反汇编和反编译的静态分析方法（尽管在后面的章节中，我们也将介绍动态调试）。虽然这些工具至少需要对低级反转概念（如汇编代码）有一个基本的了解，并且可能会导致耗时的分析会话，但它们的分析能力是无价的，无与伦比的。即使是最复杂的恶意软件样本也无法与使用这些工具的熟练分析师匹敌！

在讨论反汇编程序和反编译程序的细节之前，需要对汇编代码进行一次简短的探讨。

汇编语言基础

📝 笔记:

整本书都是关于二进制代码的反汇编和汇编语言的。

在这里，我们只提供基本知识（并在简化各种概念方面采取一些自由），并假设读者熟悉各种基本的反转概念（如寄存器等）。

有两本关于逆向工程（包括组装/拆卸）的优秀书籍：

软件（包括恶意软件）是用编程语言编写的 ...一种明确的“人性化”语言，可以被翻译（编译）成二进制代码。我们在第0x5章（“非二进制分析”）中讨论的脚本本身不是编译的，而是在运行时“解释”为系统理解的命令或代码。

如前所述，在分析被怀疑是恶意的已编译Mach-O二进制文件时，通常无法获得原始源代码。我们必须利用一个能够理解已编译的二进制机器级代码的工具，并将其翻译回更可读的东西：汇编代码！这个过程被称为分解。

汇编语言是一种低级编程语言，直接翻译成二进制指令。这种直接翻译意味着编译后的二进制代码可以（稍后）直接编译回汇编。例如，二进制序列100100000111000000111000可以在汇编代码中表示为：`add rax, 0x38`（“向rax寄存器添加38个十六进制”）。

反汇编程序的核心是将已编译的二进制文件（如恶意软件样本）作为输入，并将其转换回汇编代码。当然，这取决于我们如何理解提供的组件！

📝 笔记:

汇编有各种“版本”。我们将重点介绍x86_64（x86指令集的64位版本），即采用英特尔语法的System V ABI

汇编指令“由一个助记符表示，它通常与一个或多个操作数相结合” [3]。助记符通常描述指令：

助记符	示例	说明
添加	添加rax, 0x100	将第二个操作数（例如0x100）添加到第一个操作数。
mov	mov rax, 0x100	将第二个操作数（例如0x100）移到第一个操作数中。
jmp	jmp 0x100000100	跳转到操作数中的地址（即继续执行）。
打电话	打电话给rax	执行操作数地址指定的子例程。

通常，操作数是寄存器（CPU上的命名内存“插槽”）或数值。在反转64位Mach-O二进制时会遇到一些寄存器

包括rax、rbx、rcx、rdx、rdi、rsi、rbp、rsp和r8。 - r15。正如我们将很快看到的，通常特定的寄存器被一致地用于特定的目的，这简化了逆向工程的工作。

📝 笔记:

**所有64位寄存器也可以由其32位（或更小）组件“引用”
...在二进制分析中，你仍然会遇到。**

“所有寄存器都可以在16位和32位模式下访问。在16位模式下，寄存器由上面列表中的两个字母缩写标识。在32位模式下，这个两个字母缩写的前缀是‘E’（扩展）。例如，‘EAX’是作为32位值的累加器寄存器。

类似地，在64位版本中，“E”被替换为“R”（寄存器），因此64位版本的“EAX”被称为“RAX”[3]

在结束对汇编代码的（粗略）讨论之前，让我们简要讨论一下调用约定。这将让我们了解如何进行API（方法）调用，如何传入参数，以及如何处理响应 ...在汇编代码中。

为什么这是相关的？通过研究Mach-O二进制文件调用的系统API方法，通常可以对其有一个相当全面的了解。例如，一个恶意二进制文件调用“write file” API方法，传入~/Library/LaunchAgents目录下的属性列表和路径，很可能会作为启动代理持久存在！

因此，我们通常不需要花费数小时来理解二进制文件中的所有汇编指令，而是可以专注于“围绕” API调用来理解的指令：

- 调用了什么（API）调用
- 向（API）调用传递了哪些参数
- 它根据（API）调用的结果采取什么操作

...通常，这种理解足以对我们正在分析的（潜在恶意的）二进制样本获得相对全面的理解。

为了便于解释调用约定和方法调用（在程序集级别），我们将重点介绍一个代码段Objective-C,它创建一个NSURL对象，该对象的初始化为“www.google.com”:

```
01 //url对象
02 NSURL* url = [NSURL URLWithString:@"www.google.com"];
```

当程序想要调用方法或系统API调用时，它首先需要“准备”

通话的理由。在上面的源代码中，当调用URLWithString:方法（该方法需要一个string对象作为其唯一参数）时，Objective-C代码在字符串中传递“www.google.com”。

在程序集级别，关于如何将参数传递给方法或API函数，有一些特定的“规则”。这被称为呼叫约定。调用约定的规则在应用程序二进制接口（ABI）中明确规定，64位macOS系统的规则如下：

论据	寄存器
第一个论点	rdi
第二个论点	rsi
第三个论点	rdx
第四个论点	rcx
第五个论点	r8
第六个论点	r9

macOS（英特尔64位）呼叫约定

由于这些规则始终适用，因此作为恶意软件分析师，我们可以准确地了解通话的方式。例如，对于采用单个参数的方法，在调用之前，该参数（参数）的值将始终存储在rdi寄存器中！

因此，一旦在反汇编中识别了调用（通过调用助记符），在汇编代码中向后看就会发现传递给方法或API的参数值。这通常可以提供对代码逻辑的有价值的洞察（即恶意软件样本试图连接的URL、打开的文件路径等）。

调用指令什么时候返回呢？咨询ABI发现，方法或API调用的返回值将始终存储在rax寄存器中。因此，一旦NSURL的URLWithString: method调用返回，新构造的NSURL对象将位于rax寄存器中。

由于在call指令完成时，rax寄存器保存返回值，因此您经常会看到使用call指令进行反汇编，紧接着是指令检查，并根据rax寄存器中的值的结果执行操作。例如（我们将很快看到），如果检查网络连接的函数在rax寄存器中返回零（NO/false），则恶意样本选择不感染系统。

当反转Objective-C二进制代码时，必须理解的另一件事是

`objc_msgSend` [4]函数。

回想一下下面的Objective-C代码，它只是构造了一个URL对象：

```
01 //url对象
02 NSURL* url = [NSURL URLWithString:@"www.google.com"];
```

编译此代码时，编译器（`llvm`）会将此Objective-C调用（以及大多数其他Objective-C调用）转换为调用`objc_msgSend`的代码。或者正如苹果所解释的那样：

“当遇到[Objective-C]方法调用时，编译器生成对
...`objc_msgSend`” [4]

Apple developer文档包含此函数的一个条目，说明它“向类的实例发送带有简单返回值的消息” [4]：

Function

`objc_msgSend`

Sends a message with a simple return value to an instance of a class.

Parameters

self

A pointer that points to the instance of the class that is to receive the message.

op

The selector of the method that handles the message.

...

A variable argument list containing the arguments to the method.

`objc_msgSend`函数

由于绝大多数Objective-C调用都是通过此函数进行路由的，因此在反转编译的Objective-C代码时必须理解它。所以，让我们把它分解一下！

首先，什么是“发送信息” ... 一个类的实例“甚至是什么意思？”？简单地说，这意味着调用对象的方法。

📝 笔记:

Objective-C运行时基于发送消息的概念，以及其他相当独特的源于对象的范例。

有关Objective-C运行时及其内部的深入讨论，请参阅nemo提供的以下内容：

其次，`objc_msgSend`的参数呢：

- 第一个参数（`self`）是“指向要接收消息的类实例的指针” [4]。或者更简单地说，它是调用该方法的对象。如果方法是类方法，那么它将是类对象的一个实例（作为一个整体），而对于实例方法，`self`将指向类的一个实例化实例作为一个对象。
- 第二个参数（`op`）是“处理消息的方法的选择器” [4]。更简单地说，这只是方法的名称。
- 其余参数是方法（`op`）所需的任何值。

最后，`objc_msgSend`返回方法（`op`）返回的任何内容。

回想一下，ABI定义了如何将参数传递给函数调用。因此，我们可以准确地映射在调用时哪些寄存器将保存`objc_msgSend`的参数：

论据	寄存器	（适用于） <code>objc_msgSend</code>
第一个论点	<code>rdi</code>	<code>self</code> ：调用方法的对象
第二个论点	<code>rsi</code>	<code>op</code> ：方法的名称
第三个论点	<code>rdx</code>	方法的第一个参数
第四个论点	<code>rcx</code>	方法的第二个参数

第五个论点	r8	方法的第三个参数
第六个论点	r9	方法的第四个论点
7+参数	rsp+ (在堆栈上)	方法的5+参数

当然，寄存器`rdx`、`rcx`、`r8`、`r9`仅在被调用的方法需要它们（用于参数）时使用。例如，只接受一个参数的方法将只使用`rdx`寄存器。

此外，与任何其他函数或方法调用一样，一旦对`objc_msgSend`的调用完成，`rax`寄存器将保存返回值（实际上是调用的方法的返回值）。

这就结束了我们关于汇编语言基础知识的简短讨论。有了这个低级语言的基础知识，现在让我们看一下拆解二进制代码。

拆卸

反汇编包括将二进制代码（**1**和**0**）转换回汇编指令。然后可以分析这些汇编代码，以全面了解二进制代码。反汇编程序（将在稍后讨论）是一个能够执行这种转换并方便分析已编译二进制文件的程序。

在这里，我们将讨论各种反汇编概念，并通过真实世界的示例（直接取自恶意代码）加以说明。重要的是要记住，一般来说，分析恶意代码的目的是全面了解其逻辑和功能 ... 不一定要理解每一个装配说明。正如我们前面提到的，关注方法和函数调用的逻辑通常可以提供一种有效的方法来获得这样的理解。

因此，让我们简单地看一个反汇编代码的示例，以说明如何识别此类调用、参数和（**API**）响应。最终结果如何？全面理解反汇编代码片段。

恶意软件有时包含检查其主机是否连接到互联网的逻辑。如果受感染的系统处于脱机状态，恶意软件通常会在尝试连接到其命令和控制服务器以执行任务之前等待（睡眠）。

检查网络连接的恶意软件的一个具体例子是**OSX。Komplex [7]**，其中包含一个名为**connectedToInternet**的函数。通过研究这个民族国家后门的反汇编二进制代码，我们可以确认这个功能确实可以检查受感染的系统是否在线，并了解它是如何完成这个检查的。

具体来说，我们的分析将揭示恶意软件通过Apple的NSData类，调用dataWithContentsOfURL: method [8]来检查网络连接。如果无法访问远程URL（www.google.com）（即受感染的系统处于脱机状态），则呼叫将失败，表明系统处于脱机状态。

现在让我们深入了解OSX的分解。Komplex的ConnectedPointerNet功能（为清晰起见进行了注释）。请注意，我们将逐个分解函数，首先显示一个Objective-C表示，它是从反汇编中重建的。

```
01  connectedToInternet() {
02
03      //url对象
04      NSURL* url = [NSURL URLWithString:@"www.google.com"];
```

前面我们提到，Objective-C方法调用被“转换”为对objc_msgSend函数的调用。因此，在反汇编中看到对该函数的调用也就不足为奇了：

```
01  connectedToInternet
02
03  ;将指向NSURL类的指针移动到rdi中
04  ; rdi寄存器保存第一个参数（“self”）
05  mov     rdi, qword [objc_cls_ref_NSURL]
06
07  ;将指向方法名“URLWithString:”的指针移动到rsi中
08  ; rsi寄存器保存第二个参数（“op”）
09  利亚    rsi, qword [URLWithString:]
10
11  ;在rdx中加载url的地址
12  ; rdx寄存器保存第三个参数，这是传递给rdx的第一个参数
13  ; 正在调用的方法（URLWithString:）
14  利亚    rdx, qword [_www_google_com]
15
16  ;将指向objc_msgSend的指针移动到rax寄存器中
17  ; 然后调用它
18  mov     rax, cs:_objc_msgSend_ptr
19  打电话   rax
20
21  ;将响应保存到名为“url”的（堆栈）变量中
    ; rax寄存器保存方法调用的结果
    mov     qword [rbp+url], rax
```

我们还看到，一行Objective-C代码（`NSURL* url = [NSURL URLWithString:@"www.google.com"]`）被翻译成了几行汇编代码。

首先初始化参数（在预期寄存器中）以调用`objc_msgSend`，然后进行调用，并保存结果。

具体来说，`rdi`寄存器（第一个参数）加载了对`NSURL`类的引用。然后，用方法的名称加载第二个参数（`rsi`）：`URLWithString:`。最后用字符串“`www.google.com`”初始化`rdx`。现在可以制作`objc_msgSend`了。调用完成后，新初始化的`NSURL`对象将返回到`rax`寄存器中，并存储到局部变量中。

一旦构建了`NSURL`对象，恶意软件就会调用`NSData`的`dataWithContentsOfURL:`方法。同样，在查看反汇编之前，让我们在Objective-C中构造一个可能的表示：

```
01 //数据对象
02 //通过尝试连接/读取谷歌进行初始化。通用域名格式
03 NSData* data = [NSData dataWithContentsOfURL:url];
```

以下是OSX的（相关）反汇编代码。Komplex的连接点网络法：

```
01 ;以下代码准备了相关的寄存器
02 ; 然后通过objc_msgSend函数进行objective-c调用
03
04 ;将指向NSData类的指针移动到rdi中
05 ; rdi寄存器保存第一个参数（“self”）
06 mov     rdi, qword [objc_cls_ref_NSDData]
07
08 ;将指向方法名“dataWithContentsOfURL:”的指针移动到rsi中
09 ; rsi寄存器保存第二个参数（“op”）
10 利亚   rsi, qword [dataWithContentsOfURL:]
11
12 ;将（先前创建的）url对象移动到rdx中
13 ; rdx寄存器保存第三个参数，这是传递给rdx的第一个参数
14 ; 正在调用的方法（'dataWithContentsOfURL:'）
15 mov     rdx, qword [rbp+url]
16
17 ;将指向objc_msgSend的指针移动到rax寄存器中
18 ; 然后调用这个函数，进行objective-c调用
19 mov     rax, cs:_objc_msgSend_ptr
```

```

20 打电话    rax
21
22  ;将响应保存到名为data的（堆栈）变量中
23  ; rax寄存器保存方法调用的结果
24  mov      qword [rbp+data], rax

```

与NSURL的URLWithString:方法调用的反汇编类似，这里我们看到调用objc_msgSend的参数被初始化（在预期的寄存器中），然后进行调用，结果被保存（到名为data的变量中）。

奥斯。Komplex的ConnectedPointerNet函数根据NSData的dataWithContentsOfURL:方法的结果，通过向调用者返回一个整数值（0x0/0x01）来完成。具体来说，如果方法成功指示恶意软件能够连接到互联网并到达谷歌，则返回0x1（'true'）。通用域名格式。如果dataWithContentsOfURL方法失败（意味着它返回了一个空白（nil）数据对象），ConnectedPointerNet函数将返回0x0（'false'），以指示调用方无法访问网络。

恶意软件作者可能编写了类似于以下Objective-C代码的代码来实现这个返回值逻辑：

```

01  //设旗
02  //是（正确）如果谷歌是可访问的
03  isConnected = （数据 != nil）？是：不是；
04  返回（int）已连接；

```

这在（反汇编）汇编代码中是什么样子的？很高兴你问：

```

01  ;将数据变量与零（nil）cmp进行比较    qword
02  [rbp+data], 0x0
03
04  ;如果数据为零，
05  ; 跳转到“未连接”标签
06      notConnected
07
08  ;将“断开连接”设置为0x1
09  mov字节[rbp+未连接], 0x1
10
11  ;跳过“notConnected”逻辑jmp    离开
12
13  notConnected:
14

```

```

15
16 ;将“断开连接”设置为0x0
17 mov     字节[rbp+未连接], 0x0
18
19 离开:
20
21 ;将该值移到rax中
22 ; 注：al是rax的下位字节
23 mov     al, 字节[rbp+未连接]
24 以及    al, 0x1
25 movzx   eax, al
26
27 返回

```

首先，`cmp`指令用于比较数据变量的值（通过调用`dataWithContentsOfURL`返回）。如果为0（`nil`），则程序集代码跳转到`notConnected`标签，并将`isConnected`变量的值设置为0。否则，如果`dataWithContentsOfURL`方法返回非`nil`值，则`isConnected`变量设置为1。

最后，通过一些指令将断开连接的变量移到`rax`（`eax`）寄存器中。需要此类指令来确保布尔值正确转换为（更大的）整数值，并返回给调用者。

通常情况下，几行Objective-C代码通常被扩展成许多汇编指令，这使得分析反汇编代码相当耗时。然而，由于无法获得源代码，我们通常别无选择。而且，汇编指令确实提供了对恶意软件内部工作的无与伦比的洞察...如此之多以至于我们通常可以用更高级的语言完全重建恶意软件的代码。例如，这里是`ConnectedPointerNet`函数的完整重建：

```

01 int connectedToInternet()
02 {
03     //result
04     BOOL isConnected = 否;
05
06     //url对象
07     //让我们使用谷歌。通用域名格式
08     NSURL* url = [NSURL URLWithString:@"www.google.com"];
09

```

```

10
11 //数据对象
12 //通过尝试连接/读取谷歌来初始化。通用域名格式
13 NSData* data = [NSData dataWithContentsOfURL:url];
14
15 //设旗
16 //是的（真的），如果谷歌是可及的！
17 isConnected = (数据 != nil) ? 是 : 不是；
18
19 返回（int）已连接；
}

```

重建连接检查 (OSX.Komplex)

现在，让我们来了解一下恶意软件代码的（带注释的）反汇编，它们都调用ConnectedPointerNet函数，然后根据其响应进行操作。

```

01 isConnected:
02
03 ;调用函数
04 打电话      connectedToInternet()
05
06 ;检查是否返回了0x0或0x1
07 以及      al, 0x1
08 mov      字节[rbp+未连接], al
09 测试      字节[rbp+未连接], 0x1
10
11 ;如果0x0（未连接），则使用此选项
12 jz      notConnected
13
14 ;如果0x1（已连接），则使用此选项
15 jmp      继续
16
17 ;睡觉
18 notConnected:
19 mov      edi, 0x3c
20 打电话      睡眠
21
22 ;检查连接（再次）
23 jmp      isConnected
24

```

```
25 继续：
26  ...
```

调用ConnectedPointerNet并处理结果 (OSX.Komplex)

首先，代码调用ConnectedPointerNet函数。由于该函数不带参数，因此不需要设置寄存器。在调用之后，恶意软件会检查返回值是否为0x0（否/假）。这是通过测试和jz（跳转零）指令实现的。测试指令“对两个操作数执行按位and运算” [9]，并根据结果设置零标志。因此，如果ConnectedPointerNet函数返回一个零，则将执行jz指令，跳转到notConnected标签。在这里，代码调用sleep函数 ...在跳回已断开连接的标签之前，请再次检查连接。换句话说，恶意软件会等到系统连接到互联网后再继续。

有了这种全面的理解，我们可以（重新）构建以下逻辑 目标c代码：

```
01 while(0x0 == connectedToInternet()) {
02     sleep(0x3c);
03 }
```

*...在Objective-C
中*

当然，并不是所有的Mac二进制文件（包括恶意软件）都是用Objective-C编写的。这次让我们看看Lazarus Group first stage implant loader（最初是用C++编写的）中的另一个（经过删节和注释的）反汇编片段 [10]。具体来说，我们将浏览名为getDeviceSerial的函数中的一段汇编代码：

```
01 ;函数：getDeviceSerial (char*)
02 ; 第一个参数 (rdi)：输出缓冲区 ...用于设备串行#
03 ; 返回 (rax)：状态 (成功/错误)
04
05 ;将指向输出缓冲区的指针移到r14中
06 mov     r14, rdi
07
08 ;将kIOMasterPortDefault移到r15寄存器中
09 mov     rax, qword [_kIOMasterPortDefault]
10 mov     r15d, 德沃德[rax]
11
12
```

```
13
14 ;调用IOServiceMatch
15 ;第一个参数(rdi)：字符串“IOPlatformExpertDevice”
16 利亚    rdi, qword [IOPlatformExpertDevice]
17 打电话  IOServiceMatching
18
19 ;调用IOServiceGetMatchingService
20 ; 第一个参数(rdi)：kIOMasterPortDefault
21 ; 第二个参数(rsi)：调用IOServiceMatching的结果
22 mov     edi, r15d
23 mov     rsi, rax
24 打电话  IOServiceGetMatchingService
25
26 ;调用IORegistryEntryCreateCFProperty
27 ; 第一个参数(rdi)：调用IOServiceGetMatchingService的结果
28 ; 第二个参数(rsi)：字符串“IOPlatformSerialNumber”
29 ; 第三个参数(rdx)：分配程序(默认) kCFAllocatorDefault
30 ; 第四个参数(rcx)：选项
31 mov     r15d, eax
32 mov     rax, qword [_kCFAllocatorDefault]
33 mov     rdx, qword [rax]
34 利亚    rsi, qword [IOPlatformSerialNumber]
35 xor     ecx, ecx
36 mov     edi, r15d
37 打电话  IORegistryEntryCreateCFProperty
38
39 ;调用CFStringGetCString
40 ; 第1个参数(rdi)：调用IORegistryEntryCreateCFProperty的结果
41 ; 第二个参数(rsi)：输出缓冲区
42 ; 第三个参数(rdx)：缓冲区大小
43 ; 第四个参数(rcx)：编码
44 mov     edx, 0x20
45 mov     ecx, 0x8000100
46 mov     rdi, rax
47 mov     rsi, r14
48 打电话  CFStringGetCString

返回
```

...绝对是一个更大的汇编代码块！但别担心，我们会详细讲解。

首先，请注意反汇编程序已经提取了函数声明，其中（幸运的是）包括其原始名称以及参数的数量和格式。根据名称`getDeviceSerial`，我们假设（尽管我们也会验证）该函数将检索受感染系统的序列号。由于函数将字符串缓冲区（`char*`）的指针作为其唯一参数，因此可以合理地假设函数将在该缓冲区中存储提取的序列号（以便调用者可以使用它）。

从第#06行开始，我们看到函数首先将输出缓冲区的地址这个参数（`recall rdi`始终保留第一个参数）移动到`r14`寄存器中。为什么？如前所述，`rdi`寄存器用任何函数调用的第一个参数初始化。如果`getDeviceSerial`函数进行任何其他调用（确实如此），则必须重新初始化`rdi`寄存器（对于这些其他调用）。因此，该函数必须将输出缓冲区的地址“保存”到另一个（未使用）寄存器中，以便以后可以使用该地址 ...例如，在函数末尾，当它填充了提取的序列号时。

然后，函数（第#09 - 10行）将一个指向`kIOMasterPortDefault`的指针移动到`rax`中，并将其解引用到`r15`寄存器中。根据苹果开发人员的文档，`kIOMasterPortDefault`是“用于启动与`IOKit`通信的默认mach端口” [11] 似乎该恶意软件将与`IOKit`通信，以提取受感染设备的序列号。

在第14行和第15行中，函数`getDeviceSerial`首次调用Apple API:`IOServiceMatch`函数。Apple注意到，该函数创建“一个指定`iService`类匹配的匹配字典”，接收单个参数，并返回匹配字典[12]:

IOServiceMatching

Create a matching dictionary that specifies an IOService class match.

Declaration

```
CFMutableDictionaryRef IOServiceMatching(const char *name);
```

Parameters

name

The class name, as a const C-string. Class matching is successful on IOService's of this class or any subclass.

*IOServiceMatch*函数

我们知道，当调用函数或方法时，**rdi**寄存器保存第一个参数。在第14行，我们看到汇编代码用“IOPlatformExpertDevice”的值初始化这个寄存器。换句话说，它使用字符串“IOPlatformExpertDevice”调用IOServiceMatch函数。

一旦创建了匹配字典，代码就会调用IOServiceGetMatchingService函数（第22行）。Apple文档声明，该函数将“查找与匹配字典匹配的已注册iService对象” [14]。对于参数，它需要一个主端口和一个匹配的字典：

IOServiceGetMatchingService

Look up a registered IOService object that matches a matching dictionary.

Declaration

```
io_service_t IOServiceGetMatchingService(mach_port_t masterPort, CFDictionaryRef
```

Parameters

masterPort

The master port obtained from IOMasterPort(). Pass kIOMasterPortDefault to look up the default master port.

matching

A CF dictionary containing matching information, of which one reference is always consumed by this function.

IOServiceGetMatchingService 函数

在第20行，汇编代码将一个值从r15寄存器移到edi寄存器（rdi寄存器的32位部分）。回顾第9-10行，我们看到前面的代码将kIOMasterPortDefault移动到r15寄存器中。第20行的代码只是将kIOMasterPortDefault移动到edi寄存器中（作为调用IOServiceGetMatchingService的第一个参数）。

在第21行，我们看到rax被移动到rsi寄存器中（回想一下，rsi寄存器被用作函数调用的第二个参数）。并且（在函数调用之后），rax寄存器保存调用的结果。这意味着rsi寄存器将包含调用IOServiceMatching（在第15行生成）的匹配字典。

调用IOServiceGetMatchingService后，将返回一个io_服务（在rax寄存器中）。具体而言，是与IOPlatformExpertDevice匹配的服务。

接下来，代码为调用IORegistryEntryCreateCFProperty函数设置参数，苹果文档称该函数“创建注册表项属性的即时快照” [14] 换句话说，代码正在提取某些（IOKit）注册表属性的值。但是哪一个呢？

IORegistryEntryCreateCFProperty

Create a CF representation of a registry entry's property.

Declaration

```
CTypeRef IORegistryEntryCreateCFProperty(io_registry_entry_t entry, CFString
Ref key, CFAllocatorRef allocator, IOOptionBits options);
```

Parameters

entry

The registry entry handle whose property to copy.

key

A CFString specifying the property name.

allocator

The CF allocator to use when creating the CF container.

options

No options are currently defined.

调用IORegistryEntryCreateCFProperty函数的参数设置首先将kCFAllocatorDefault加载到rdx寄存器中（第29-13行）。rdx寄存器用于第三个参数，调用IORegistryEntryCreateCFProperty的参数是“要使用的分配器” [12]。

接下来（第32行），字符串“IOPlatformSerialNumber”的地址被加载到rsi寄存器中。由于rsi寄存器用于第二个参数，因此这（根据苹果关于IORegistryEntryCreateCFProperty函数的文档）是感兴趣的属性名称！

在第33行，rcx, 第四个参数（“选项”）被初始化为零（自身的xor ing，将自身设置为零）。最后，在进行调用之前，r15d中的值被移动到rdi寄存器（edi）的32位部分。这具有初始化的效果

第一个参数（`rdi`）的值为`kIOMasterPortDefault`（以前存储在`r15d`中）。

调用`IORegistryEntryCreateCProperty`后，`rax`寄存器将保存所需属性的值：`IOPlatformSerialNumber`。

最后，该函数调用`CFStringGetCString`函数，将提取的属性（即（`CF`）字符串对象）转换为纯空终止的“C字符串”。当然，参数必须在调用之前初始化（第42-45行）。

`edx`寄存器（`rdx`的32位部分，参数#3）设置为`0x20`，用于指定输出缓冲区大小。然后将`ecx`寄存器（`rcx`的32位部分，参数#4）设置为`kCFStringEncodingUTF8（0x8000100）`。第一个参数（`rdi`）设置为`rax`的值，这是调用`IORegistryEntryCreateCProperty`的结果：`IOPlatformSerialNumber`的提取属性值。

最后，第二个参数（`rsi`）设置为`r14`。 `r14` 寄存器中有什么？一直向后滚动到第6.，我们看到它来自`rdi`，它是传递给`getDeviceSerial`的参数的值。由于Apple的`CFStringGetCString`文档中指出，第二个参数是“将字符串复制到其中的C字符串缓冲区，” [15. 我们现在知道传递给`getDeviceSerial`函数的参数是字符串缓冲区！

这就完成了我们的（非常彻底！）分析恶意软件的`getDeviceSerial`功能。通过关注该函数发出的API调用，我们能够确定其确切功能：通过`IOKit`检索受感染系统的序列号（`IOPlatformSerialNumber`）。此外，通过参数分析，我们能够确定`getDeviceSerial`函数将通过序列号缓冲区调用。

...谁需要源代码！？

然而在这一点上，我们都同意读取汇编代码是相当乏味的。 幸运的是，由于反编译器的最新进展，有希望了！

反编译

给定一个二进制文件，比如`Mach-O`，反汇编程序可以解析该文件并将二进制代码翻译回人类可读的程序集，从而开始详细的分析。

反编译器试图通过重新创建提取的二进制代码的源代码级表示，将这种转换进一步推进。源代码（即C或Objective-C）表示法比（dis）汇编更简洁、更“可读”，使未知二进制文件的分析变得更简单。

回想一下Lazarus Group第一阶段植入装载机的getDeviceSerial功能。该功能的完全分解约为50行。反编译？...15岁左右：

```

01 int getDeviceSerial(int * arg0) {
02     r14 = arg0;
03     ...
04     r15 = kIOMasterPortDefault;
05     rax = IOServiceMatching("IOPlatformExpertDevice");
06     rax = IOServiceGetMatchingService(r15, rax);
07     如果 (rax != 0x0) {
08         rbx = CFStringGetCString(IORegistryEntryCreateCFProperty(rax,
09             @"IOPlatformSerialNumber", kCFAllocatorDefault, 0x0), r14, 0x20,
10             kCFStringEncodingUTF8) != 0x0 ? 0x1 : 0x0;
11         IOObjectRelease(rax);
12     }
13     rax = rbx;
14     返回rax;
15 }

```

getDeviceSerial 反编译

反编译是相当可读的，因此它相对容易理解这个函数的逻辑！

类似地，本章前面讨论的ConnectedPointerNet函数也进行了适当的反编译（尽管反编译器确实对Objective-C语法感到有点困惑）...不过，谁不是呢？）：

```

01 集成 connectedToInternet()
02 {
03     if( (@class(NSData), &@selector(dataWithContentsOfURL:), (@class(NSURL),
04         &@selector(URLWithString:), @"http://www.google.com")) != 0x0)
05     {
06         var_1 = 0x1;
07     }
08     否则{

```

```
09         var_1 = 0x0;  
10     }  
11     rax = var_1 & 0x1 & 0xff  
12     ; 返回rax;  
13 }
```

已反编译ConnectedPointerNet (
OSX.Komplex)

笔记:

考虑到反编译比反编译有很多好处，人们可能会想知道为什么要讨论反编译。

首先，即使是最好的反编译器有时也难以分析复杂的二进制代码（例如带有反分析逻辑的恶意软件）。简单翻译二进制代码（而不是试图（重新）创建源代码级表示）的反汇编程序受影响要小得多。因此，“下拉”到反汇编程序提供的汇编级代码可能是唯一的选择。

其次，正如我们在上面对getDeviceSerial和ConnectedPointerNet函数的反编译中所看到的，汇编代码概念（如寄存器）仍然存在于代码中，因此是相关的。

虽然反编译可以大大简化二进制代码的分析，但理解（dis）汇编代码的能力可以说是全面恶意软件分析的一项基本技能。

和Hopper一起动手

到目前为止，我们讨论了反汇编和反编译的概念，但没有提到提供这些服务的特定工具。这样的工具可能有点复杂，因此对于初学恶意软件分析的人来说有点吓人。因此，在这里，我们将简要讨论一个这样的工具（Hopper），它为二进制分析提供了一个高级的、实践性的“快速入门”指南！

[Hopper](#) [16] 被其创作者描述为，

“反向工程工具，可用于反汇编、反编译和调试应用程序。” [16]

Hopper价格合理，专为macOS设计，拥有强大的反汇编和反编译功能，擅长分析Mach-O二进制文件。这是Mac恶意软件分析的可靠选择。

笔记:

Hopper的免费演示版可从以下网站获得：

<https://www.hopperapp.com/download.html>

如果您熟悉或喜欢另一个（可能更强大的）反汇编程序/反编译程序（如IDA Pro或Ghidra），本节的细节可能不适用。

然而，在概念层面上，它们广泛适用于大多数逆向工程工具。

在对Hopper的简要介绍中，我们将分解并反编译Apple的标准“Hello World”（Objective-c）代码：

```
01  #import <Foundation/Foundation.h>
02
03  int main(int argc, const char * argv[]) {
04      @autoreleasepool {
05          //在这里插入代码...
06          NSLog(@"Hello, World!");
07      }
08      返回0;
09  }
```

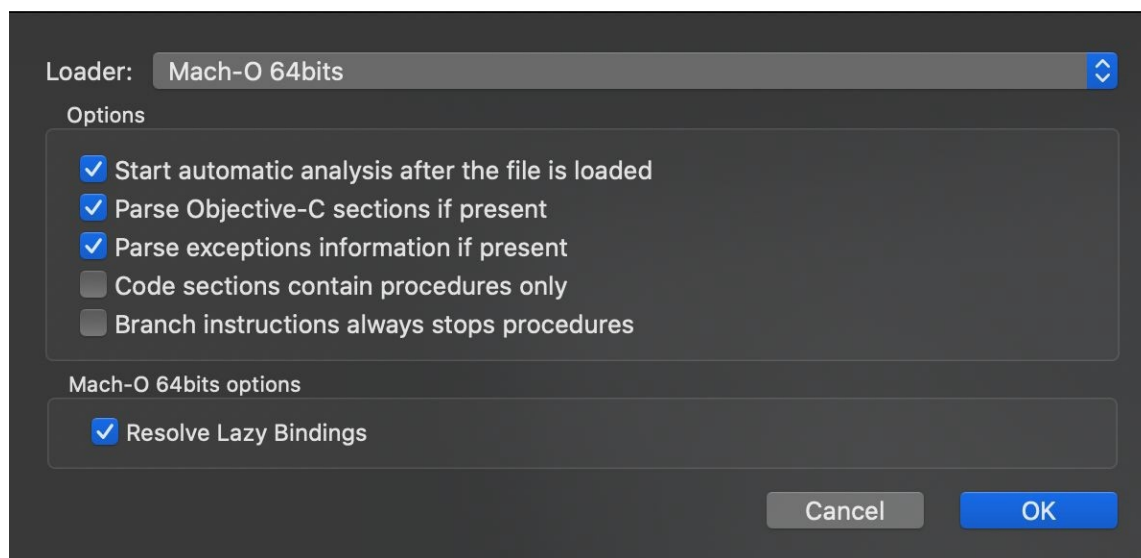
苹果的“Hello World”模板代码

虽然它给我们提供了一个简单的二进制示例，足以说明Hopper的许多特性和功能。当然，了解这些特性和功能对于分析更复杂（恶意）的二进制文件是必不可少的。

我们首先编译上面的Objective-C代码，并确认它现在（如预期的那样）是一个标准的64位Mach-O二进制文件：

```
$ file helloWorld/Build/Products/Debug/helloWorld
helloWorld: Mach-O 64位可执行文件x86_64
```

首先，发射漏斗。应用程序。要开始分析helloWorld（或任何）Mach-O二进制文件，只需选择：**File -> Open**（⌘+O）。选择Mach-O二进制文件进行分析，并在显示的加载程序窗口中选择默认值，然后单击“确定”：



加载器窗口（
Hopper.app）

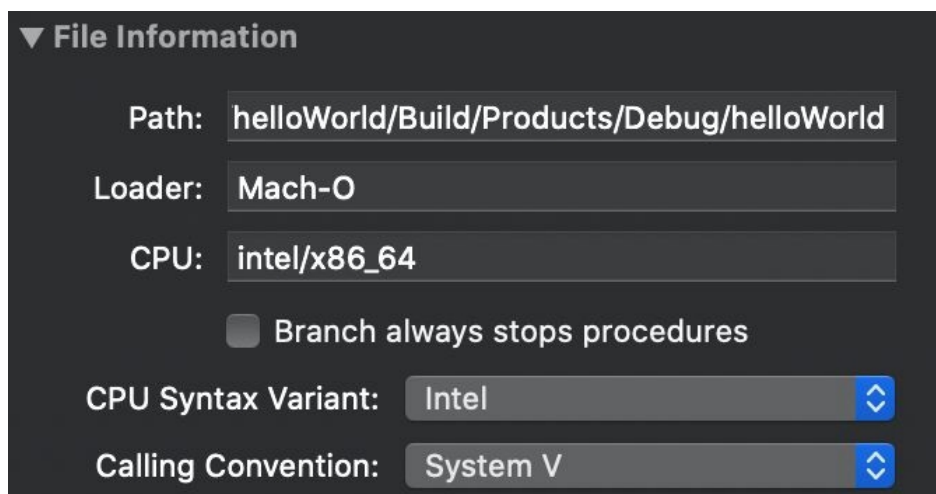
Hopper将自动开始分析二进制文件，包括：

- 解析Mach-O头
- 反汇编二进制代码
- 提取嵌入字符串、函数/方法名称等。

分析完成后，Hopper将在二进制文件的入口点（从Mach-O标题中的LC_主加载命令中提取）自动显示反汇编代码。

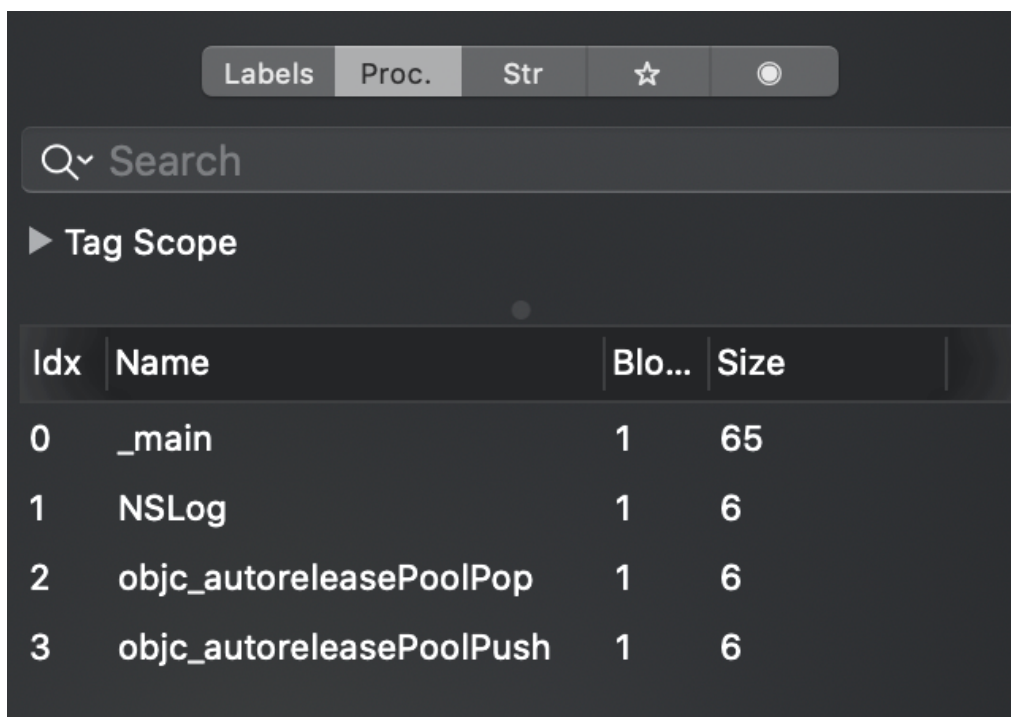
...但首先，让我们看看Hopper UI中的各种信息和选项。

最右边是“检查员”视图。Hopper将在此处显示有关正在分析的二进制文件的一般信息，包括二进制文件的类型（Mach-O）、体系结构/CPU（Intel x86_64）和调用约定（System V）：



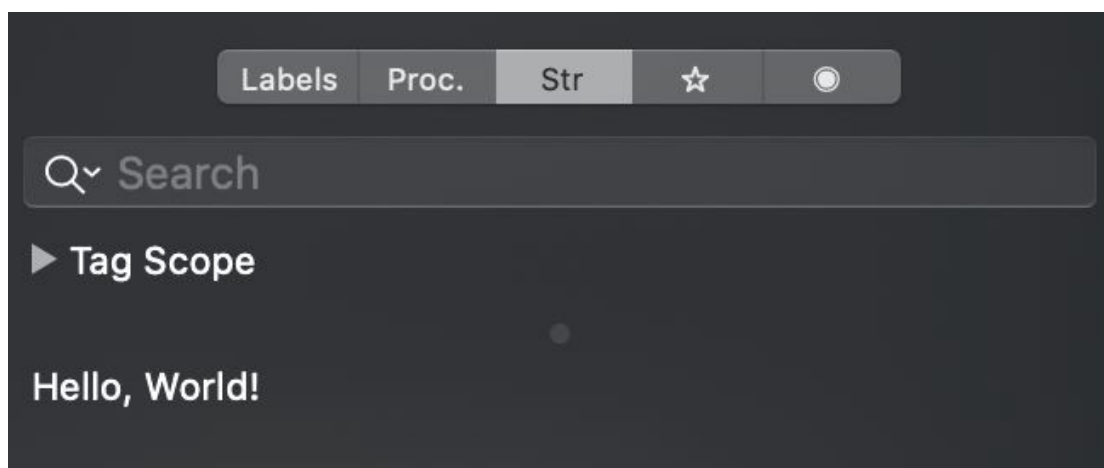
基本文件信息（
Hopper.app）

最左边是一个段选择器，可以在与二进制文件中的符号和字符串相关的各种视图之间切换。例如，“Proc”视图显示了Hopper在分析过程中识别的程序。这包括来自原始源代码的函数和方法，以及代码调用的API。例如，在我们的“hello world”二进制文件中，Hopper识别了主函数和对Apple的NSLog API的调用：



程序视图（Hopper.app）

“Str”视图显示Hopper从二进制文件中提取的嵌入字符串。在我们的简单二进制文件中，唯一嵌入的字符串是“你好，世界！”：



(嵌入式) 字符串视图 (
Hopper.app)

在深入反汇编之前，仔细阅读提取的过程名称和嵌入的字符串是明智的，因为它们通常是有关恶意软件（可能）功能的宝贵信息源。此外，它们还可以指导分析工作。过程名或嵌入的字符串看起来有趣吗？只需点击它，Hopper就会显示二进制文件中引用它的确切位置。

默认情况下，Hopper将自动显示二进制程序入口点的反汇编（通常是主功能）。下面是主函数的整体分解：

```
; ===== BEGINNING OF PROCEDURE =====  
  
; Variables:  
;   var_4: int32_t, -4  
;   var_8: int32_t, -8  
;   var_10: int64_t, -16  
;   var_18: int64_t, -24  
  
_main:  
0x00000000100000f20    push    rbp  
0x00000000100000f21    mov     rbp, rsp  
0x00000000100000f24    sub     rsp, 0x20  
0x00000000100000f28    mov     dword [rbp+var_4], 0x0  
0x00000000100000f2f    mov     dword [rbp+var_8], edi  
0x00000000100000f32    mov     qword [rbp+var_10], rsi  
0x00000000100000f36    call    imp___stubs_objc_autoreleasePoolPush ; objc_autoreleasePoolPush  
0x00000000100000f3b    lea     rcx, qword [cfstring_Hello_World_] ; @"Hello, World!"  
0x00000000100000f42    mov     rdi, rcx ; argument "format" for method imp___stubs_NSLog  
0x00000000100000f45    mov     qword [rbp+var_18], rax  
0x00000000100000f49    mov     al, 0x0  
0x00000000100000f4b    call    imp___stubs_NSLog ; NSLog  
0x00000000100000f50    mov     rdi, qword [rbp+var_18] ; argument "pool" for method imp___stubs_objc_autoreleasePoolPop  
0x00000000100000f54    call    imp___stubs_objc_autoreleasePoolPop ; objc_autoreleasePoolPop  
0x00000000100000f59    xor     eax, eax  
0x00000000100000f5b    add     rsp, 0x20  
0x00000000100000f5f    pop     rbp  
0x00000000100000f60    ret  
; endp
```

“Hello World”已分解（ Hopper.app）

...相当标准的（dis）组装。不过，Hopper确实提供了一些有用的注释，比如识别函数名（即将imp存根NSLog映射到NSLog）。此外，由于它通常也理解API原型，因此它将识别函数/方法参数，并对汇编代码进行注释。

例如，地址0x0000000100000f42处的汇编代码移动rcx寄存器（指向“Hello, World!”的指针）Hopper将其识别为初始化调用NSLog的参数（几行之后）。

反汇编中的各种组件实际上是指向二进制文件中其他位置数据的指针。例如，0x0000000100000f3b（lea rcx, qword [cfstring_Hello World_]）处的汇编代码正在加载“Hello, World!”的地址将字符串插入rcx寄存器。

Hopper足够聪明，可以将cfstring_Hello World_变量识别为指针，从而用字符串的值（字节）注释汇编代码（“Hello, World!”）。此外，如果双击任何指针，Hopper将跳转到指针的地址。例如，单击反汇编中的cfstring_Hello World_uu变量两次，可以找到地址为0x0000000100001008的字符串对象：

```
01 你好，世界；“你好，世界！”
02 0x0000000100001008 dq 0x0000000100008008,
03 0x0000000100001010 dq 0x00000000000007c8,
04 0x0000000100001018 dq 0x0000000100000fa2,
05 0x0000000100001020 dq 0x000000000000000d
```

这个字符串对象（CFConstantString类型）本身包含指针 ...再次双击这些按钮会将您带到指定的地址。

例如，偏移量+0x0处是一个值为0x0000000100008008. 指针。双击该值，我们将看到一个标记为

__CFConstantStringClassReference（字符串对象的类类型）。在偏移时+0x10是指向字符串实际字节的指针（位于0x0000000100000fa2）：

```
01 aHelloWorld:
02 0x0000000100000fa2 db “你好，世界！”，0 ; 数据XREF=cfstring_Hello World_
```

请注意，Hopper还跟踪（向后）交叉引用！例如，它已确定字符串字节（地址0x0000000100000fa2处）由cfstring_Hello World_u变量交叉引用。也就是说，cfstring_Hello World_uu变量包含对0x0000000100000fa2地址的引用。

这种交叉引用极大地促进了二进制代码的静态分析。例如，如果您注意到一个感兴趣的字符串，您可以简单地询问Hopper该字符串在代码中的何处被引用。要查看此类交叉引用，请单击地址或项目并选择“引用到...” ...或者选择地址/项目后，只需点击“X”。

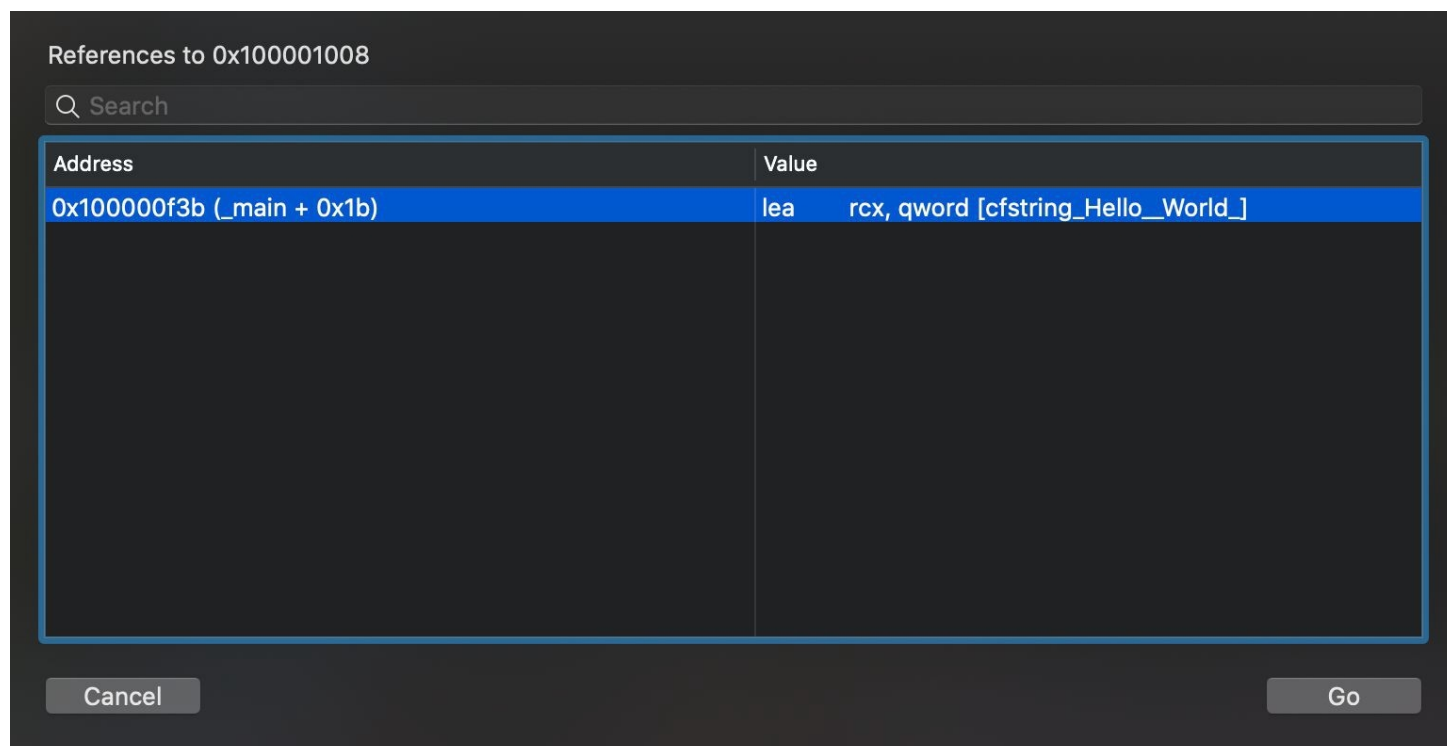
例如，假设我们想看看反汇编中的“Hello World！”字符串对象被引用。首先，我们选择字符串对象（地址为0x0000000100001008），

控件单击以打开上下文菜单和“对cfstring_Hello World的引用”：



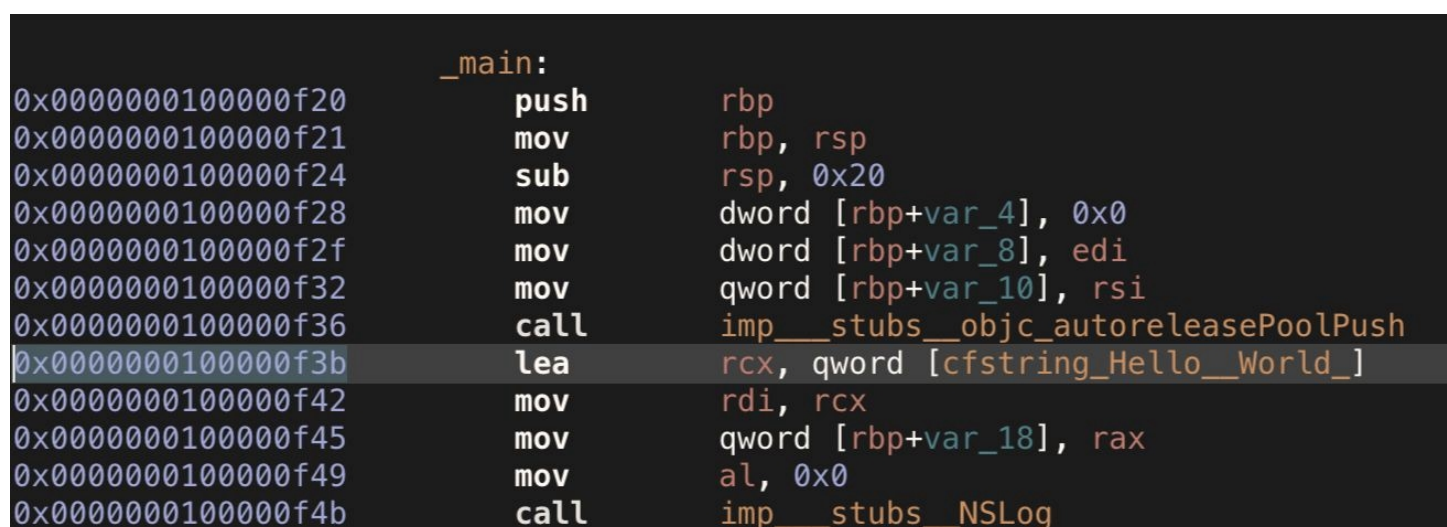
交叉引用（
Hopper.app）

...这将打开该项目的“交叉引用”窗口：



交叉参考窗口 (Hopper.app)

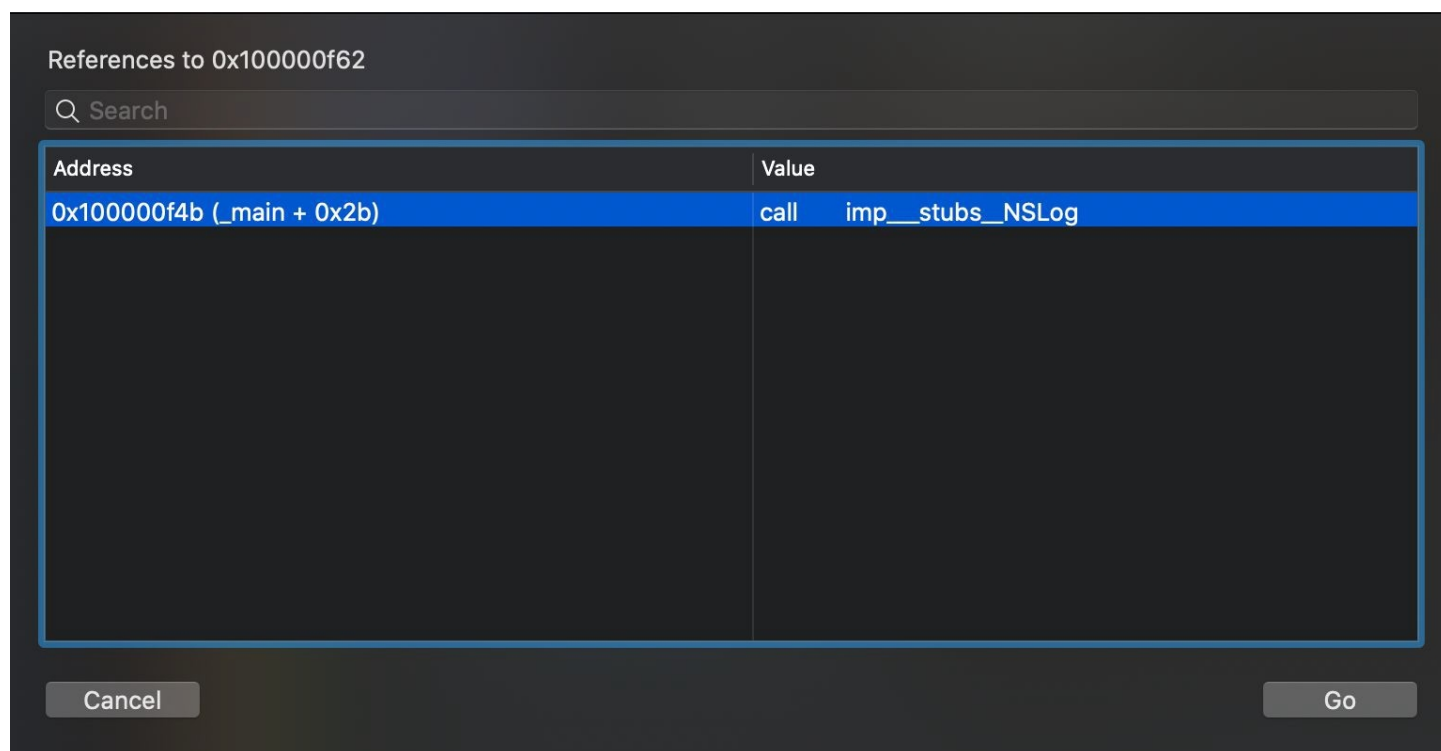
在本例中，只有一个交叉引用，地址0x0000000100000f3b处的代码（属于主函数）。单击此按钮可跳转到main函数中的代码，该函数引用“Hello World”字符串对象：



“Hello World” (cf) 字符串 (Hopper.app)

Hopper还为函数、方法和API调用创建交叉引用，以便

可以轻松确定在代码中调用它们的位置。例如，我们可以通过下面的“交叉引用”窗口看到，`NSLog` API在主函数中被调用，特别是在`0x0000000100000f4b`：

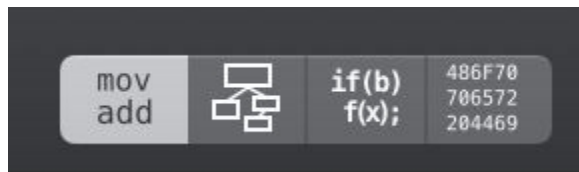


交叉参考窗口（
Hopper.app）

交叉引用极大地促进了分析，并能有效地帮助理解二进制的功能。例如，在分析可疑的恶意软件样本时，可以找到感兴趣的API（可能是苹果的网络方法，可能会发现与C&C服务器的连接？）在料斗“Proc”视图中。从这个角度来看，遵循他们的交叉引用，快速找到相关代码，以充分了解这些API是如何使用的。

当在Hopper中跳转时（例如跟随指针或交叉引用），人们通常希望快速返回到之前的分析点。幸运的是，“esc”键被映射到“back”，并将带您回到刚才所在的位置，或者更远（在多次按键时）。

到目前为止，我们一直处于Hopper的默认显示模式：“组装模式”顾名思义，这种模式显示二进制代码的（dis）组装。显示模式可以通过Hopper主工具栏中的段控件进行切换：

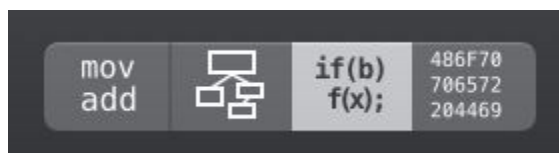


显示模式 (
Hopper.app)

Hopper支持的显示模式包括：

- 装配模式：
标准的反汇编模式，在这种模式下，Hopper “一行接一行地打印汇编代码” [15]
- 控制流图模式：
该模式将程序（如功能）分解为条件块，并说明它们之间的控制流程。
- 伪码模式：
这是Hopper的反编译模式，在这种模式下生成“类似于源代码”或伪代码表示。
- 十六进制模式：
此模式显示二进制文件的原始十六进制字节，这大约是最低级别！

在四种显示模式中，伪代码（反编译器）模式可以说是最强大的。要进入该模式，首先选择一个程序，然后单击显示模式段控件中的第三个按钮：



显示模式：反编译 (*Hopper.app*
)

这将指示Hopper对过程中的代码进行反编译，以生成二进制代码的伪代码表示。对于我们的简单示例“Hello World”程序，它做得很好：

```
int _main(int arg0, int arg1) {  
    var_18 = objc_autoreleasePoolPush();  
    NSLog(@"Hello, World!");  
    objc_autoreleasePoolPop(var_18);  
    return 0x0;  
}
```

...它看起来几乎和原始源代码一模一样：

```
01  #import <Foundation/Foundation.h>  
02  
03  int main(int argc, const char * argv[]) {  
04      @autoreleasepool {  
05          //在这里插入代码...  
06          NSLog(@"Hello, World!");  
07      }  
08      返回0；  
09  }
```

苹果的“你好世界”

...因此，使二进制分析（这个微不足道的二进制）变得轻而易举！

这是我们对Hopper反向工程工具的概述。虽然简短，但它提供了开始反转Mach-O二进制文件的基础！

笔记:

有关使用和理解Hopper的更全面的“操作方法”，请查看该应用程序的官方教程：

<https://www.hopperapp.com/tutorial.html> [16]

下一个

有了对静态分析技术的深入了解，从基本的文件类型识别到高级反编译，我们现在准备将注意力转向

动力分析方法。正如我们将看到的，这种动态分析通常提供了执行恶意软件分析的更有效方法。

但最终，静态和动态分析是互补的；它们的结合提供了最终的分析方法。

参考文献

1. “黑客”
<https://www.amazon.com/Hacker-Disassembling-Uncovered-Kris-Kaspersky/dp/1931769648>
2. “逆向：逆向工程的秘密”
<https://www.amazon.com/Reversing-Secrets-Engineering-Eldad-Eilam/dp/0764574817>
3. “x86汇编语言”
https://en.wikipedia.org/wiki/X86_assembly_language
4. objc_msgSend函数
https://developer.apple.com/documentation/objectivec/1456712-objc_msgsend
5. “现代Objective-C开发技术”
<http://www.phrack.org/issues/69/9.html>
6. “Objective-C运行时：理解和滥用”
<http://www.phrack.org/issues/66/4.html>
7. “Sofacy的‘Komplex’ OS X特洛伊木马”
<https://unit42.paloaltonetworks.com/unit42-sofacys-komplex-os-x-trojan/>
8. NSData的dataWithContentsOfURL:方法
https://developer.apple.com/documentation/foundation/nsdata/1547245-datawithcontent_sofurl
9. “测试（x86指令）”
[https://en.wikipedia.org/wiki/TEST_\(x86_instruction\)](https://en.wikipedia.org/wiki/TEST_(x86_instruction))
10. “Lazarus集团变得‘无文件化’” https://objective-see.com/blog/blog_0x51.html
11. “kIOMasterPortDefault”
<https://developer.apple.com/documentation/iokit/kiomasterportdefault?language=objc>
12. “IOServiceMatching”
<https://developer.apple.com/documentation/iokit/1514687-ioservicematching?language=objc>
13. “IOServiceGetMatchingService”
<https://developer.apple.com/documentation/iokit/1514535-ioservicegetmatchingservice>

[?language=objc](#)

14.“IORegistryEntryCreateCFProperty”

<https://developer.apple.com/documentation/iokit/1514293-ioregistryentrycreatecfproperty?language=objc>

15.“CFStringGetCString”

<https://developer.apple.com/documentation/corefoundation/1542721-cfstringgetcstring?language=objc>

16.漏斗

<https://www.hopperapp.com>
/

17.Hopper教程

<https://www.hopperapp.com/tutorial.html>
1