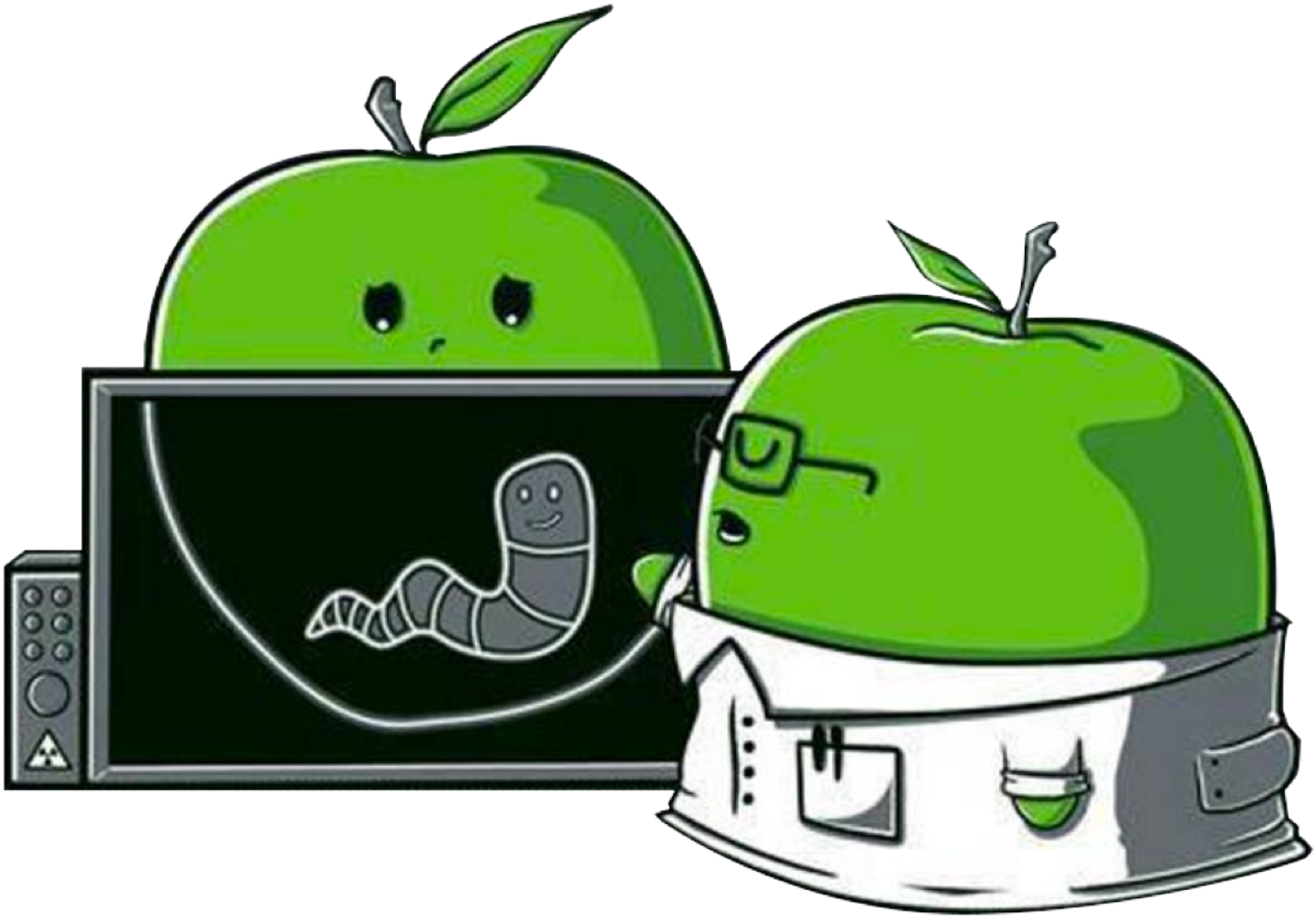


第0x0A章：调试

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）。

在前一章中，我们介绍了被动动态分析工具，包括进程、文件和网络监视器。而这些工具通常可以（很快！）提供对恶意样本的宝贵洞察，但有时他们无法。此外，虽然它们可能允许人们观察受审查样本的行为，但此类观察是间接的，因此可能无法提供对样本内部工作的真实见解。需要更强大的东西！

最终的动态分析工具是调试器。简单地说，调试器允许一条一条地执行二进制指令。任何时候都可以检查（或修改）寄存器和内存内容，跳过（绕过）整个函数，等等。

在深入研究调试概念之前，先来看一个简单的例子 ...这个例子清楚地说明了调试器的功能。奥斯。Mami [1]包含大量嵌入的加密数据，并将其传递给名为setDefaultConfiguration的方法：

```
[SBConfigManager设置默认配置:
@"uZmgulcipekSbayTO9ByamTUu_zVtsflazc2Nsuqgq0dXkoOzKMJMNTULoLpd-QV9qQy6VRluzRXqWOG
scgheRvikLkPRzs1pJbey2QdaUSXUZCX-UNERrosul22NsW2vYpS7HQO4VG5l8qic3rSH_fAhxsBXpEe55
7eHIr245LUYcEIpemnvSPTZ_lNp2XwyOJjzcJWirKbKwtc3Q61pD..."];
```

一般来说，恶意样本中的加密数据是恶意软件作者试图隐藏的数据 ...要么来自检测工具，要么来自恶意软件分析师。作为后者，我们当然非常有动力解密这些数据，以揭示其隐藏的秘密。

就OSX而言。Mami，基于上下文（即调用名为setDefaultConfiguration:）的方法），似乎可以合理地假设该嵌入数据是恶意软件的（初始）配置，其中可能包含有价值的信息，如命令和控制服务器的地址、对恶意软件功能的了解等。

那么如何解密呢？静态分析方法将非常缓慢且效率低下，而文件或进程监视器将几乎没有用处，因为加密的配置信息既不会写入磁盘，也不会传递给任何（其他）进程。换句话说，它只存在于内存中，被解密。

通过调试器（稍后将介绍更多），我们可以指示恶意软件执行，并在SBConfigManager的setDefaultConfiguration:方法处停止。然后，一条指令一条指令地“步进”（执行），我们允许恶意软件以受控的方式继续执行，在完成对其配置信息的解密后再次暂停。

由于调试器可以直接检查正在调试的进程的内存，因此我们可以简单地“转储”（打印）现在已解密的配置信息（由rax寄存器指向）：

```
# lldb MaMi
（lldb）目标创建“MaMi”
当前可执行文件设置为“MaMi”（x86_64）。

...

(lldb) po $rax
{
 "dnsChanger" = {
   "affiliate" = "";
   "blacklist_dns" = ();
   "encrypt" = true;
   "external_id" = 0;
   "product_name" = dnsChanger;
   "publisher_id" = 0;

    ...

   "setup_dns" =         (
      "82.163.143.135",
      "82.163.142.137"
    );

    “共享存储”=“/Users/%USER\u NAME%/Library/Application Support”；“存储
    超时”=120；
   };

 "installer_id" = 1359747970602718687;
 ...
}
```

各种解密的密钥/值对（如“product_name”=Dnshanger）提供了对恶意软件最终目标的洞察；劫持受感染的系统DNS设置，迫使域名解析通过攻击者控制的服务器路由（如解密配置中所述，在82.163.143.135和82.163.142.137处找到）。

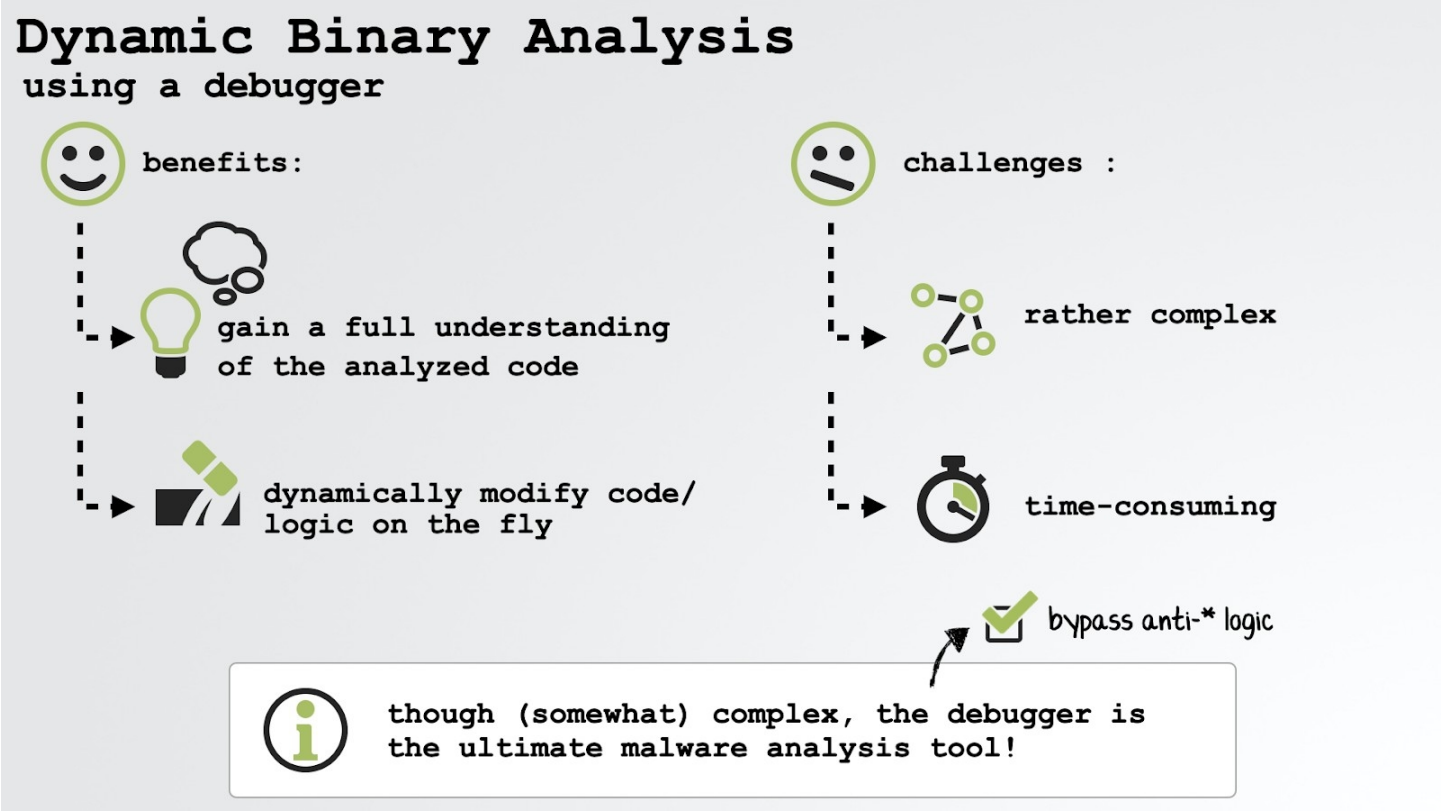
这种分析方法和对嵌入式配置信息的解密最值得注意的一点可能就是几乎不需要动一根手指！相反，通过调试器，我们只允许恶意软件愉快地执行 ...然后（恶意软件不知道），从内存中提取解密数据。

这只是一个例子，说明了调试器的威力！更全面地说，调试器的好处包括：

- 全面理解所分析的代码
- 动态修改代码，例如绕过反分析逻辑

当然，还有一些挑战在一定程度上削弱了这些好处，包括调试器：

- 是一个相当复杂的工具（需要特定的、低级的知识）
- 完成分析可能需要大量时间

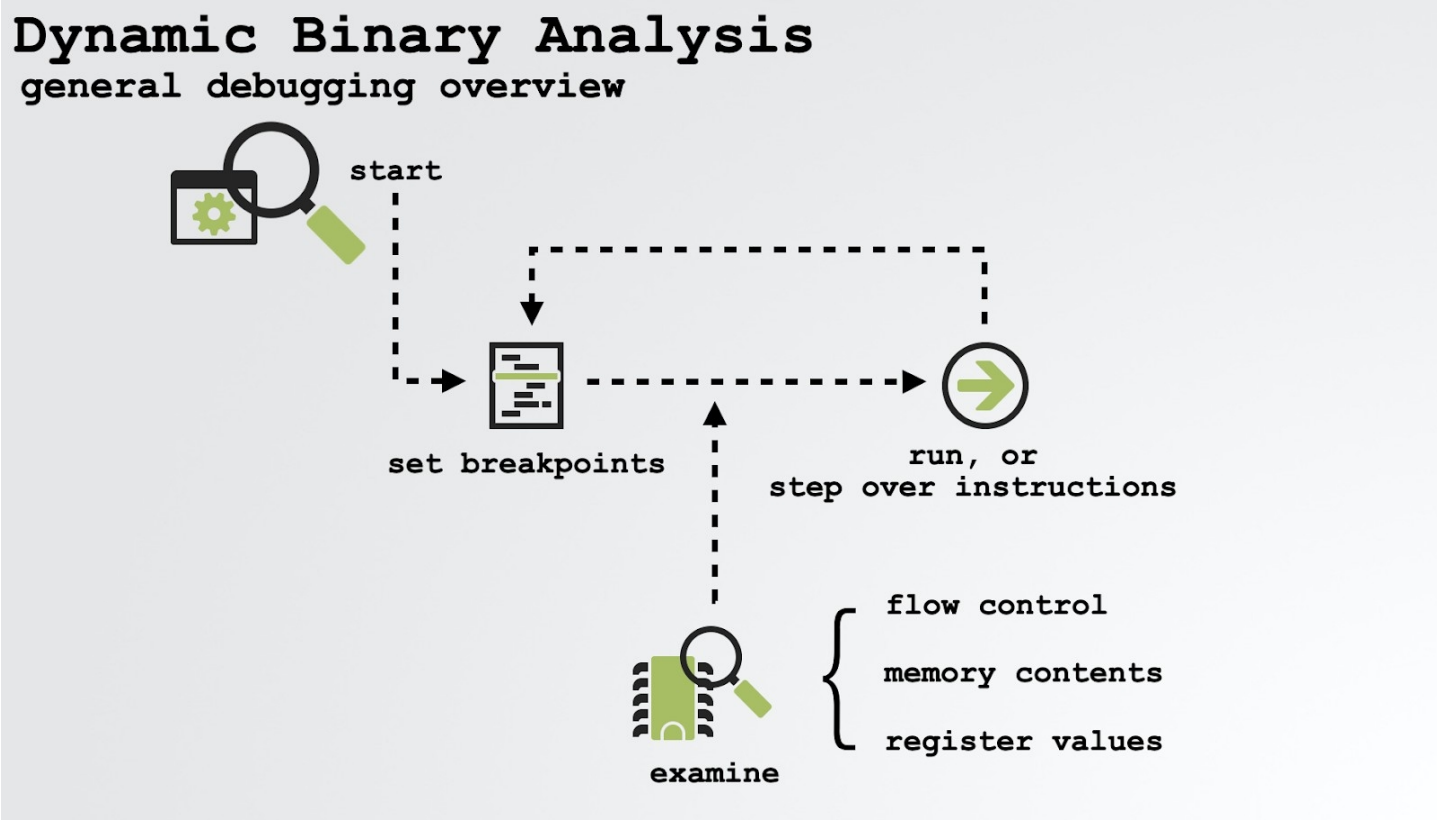


但是，一旦您了解了调试器的概念和技术，能够高效地进行调试，调试器将成为您最好的（恶意软件分析）朋友。

调试概念

在高层，调试会话通常以以下方式流动：

1. 目标进程（即要分析的恶意软件样本）被“加载”到调试器中。
2. 断点设置在代码中的不同位置，例如恶意软件的主入口点或感兴趣的方法调用处。
3. 示例将启动并运行，直到遇到断点，此时执行将停止。
4. 一旦停止，就可以自由地四处查看、检查内存和寄存器值（例如检索未加密的数据）、控制流（即调用堆栈）等等。
5. 然后，执行要么继续（并一直运行，直到命中另一个断点），要么一次执行一条指令。



笔记:

调试恶意样本时，允许其执行（尽管是在检测环境中）。因此，请始终在虚拟机中执行调试。

除了确保不会发生持久性损坏外，虚拟机还可以恢复到以前的状态。这通常在调试会话期间非常有用（例如，当断点丢失且恶意软件全部执行时）。

要启动调试会话，只需将示例加载到调试器中。另一个选项是将其附加到已经运行的进程。然而，在分析恶意样本时，我们通常希望在恶意软件开始执行时开始调试。

笔记:

这里我们重点介绍如何使用lldb,这是macOS二进制调试的实际工具。尽管应用程序（如Hopper）在其上构建了用户友好的界面，但通过其命令行界面直接与lldb交互可以说是最强大、最有效的调试方法。

lldb与Xcode（/usr/bin/lldb）一起安装。但是，如果希望只从终端安装lldb：键入lldb,点击enter,并同意安装提示。

lldb网站[2]提供了丰富的详细知识，例如该工具的深入教程[3]。

此外，任何命令都可以参考lldb help命令，以提供内联信息。例如，下面描述了在断点上操作的命令:

（lldb）帮助断点

lldb官方教程还指出，有一个“apropos”命令，“将在帮助文本中搜索特定单词的所有命令，并为每个匹配的命令转储摘要帮助字符串。” [3]

在lldb中启动调试会话有几种方法。最简单的是从终端执行lldb，以及要分析的二进制文件的路径（加上所述二进制文件的任何附加参数）:

```
$ lldb ~/下载/恶意软件。bin任意args（

lldb）目标创建“malware.bin”
```

```
当前可执行文件设置为“恶意软件”。bin'（x86_64）。
```

如上所示，lldb将显示一条目标创建消息，记录要调试的可执行文件集，并识别其体系结构。尽管调试会话已经创建，但示例的所有指令（例如malware.bin）都尚未执行。

lldb还可以通过-pid <target pid>连接到正在运行的进程的实例。然而，在分析恶意软件的环境中，这种方法并不常用（因为恶意软件已经关闭并运行，因此实际上可以防止这种连接）。

最后，在lldb外壳中，可以利用 --waitfor命令，“它等待具有该名称的下一个进程出现，并附加到它” [3]。

```
$ lldb

（lldb）进程附加 --命名恶意软件。箱子 --等待
```

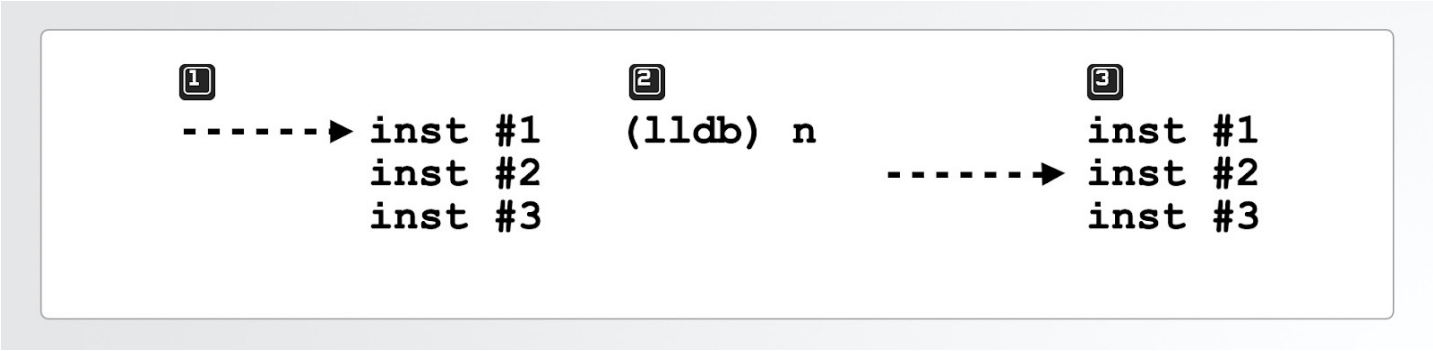
现在，每当一个进程被命名为恶意软件。bin启动后，调试器将自动附加:

```
（lldb）进程附加 --命名恶意软件。箱子 --等待
...

进程14980已停止
*线程#1,队列= 'com。苹果主线程'，停止原因=信号SIGSTOP
...

可执行模块设置为“~/Downloads/malware.bin”。 架构设置
为：x86_64h-apple-macosx-。
```

这个 --waitfor命令在恶意软件衍生出您想要调试的其他（恶意）进程时特别有用。

流量控制

在讨论断点（指示调试器在指定位置停止）之前，让我们先简单地讨论一下执行控制。调试器最强大的一个方面是它能够精确地控制正在调试的进程的执行 ...例如，指示进程执行一条指令（然后停止）。



下表介绍了与执行控制相关的几个lldb命令:

（lldb）命令	说明
跑（右）	运行调试过的进程。 开始执行，直到遇到断点或进程终止为止。
继续（c）	继续执行已调试的进程。 与run命令类似，执行将继续，直到断点或进程终止。
nexti (n)	执行程序计数器（rip）寄存器指向的下一条指令，然后停止。此命令将跳过函数调用。
stepi (s)	执行程序计数器（rip）寄存器指向的下一条指令，然后停止。与nexti命令不同，该命令将进入函数调用。
完成（f）	执行当前函数（“帧”）中的其余指令返回并停止。
控制+ c	暂停执行。如果进程已经运行（r）或继续（c），这将导致进程停止 ...无论它目前在哪里执行。

笔记:

一般来说，lldb与gdb命令保持向后兼容性（gdb是GNU项目调试器，通常在lldb之前使用）。例如，对于单步，lldb既支持“线程步骤inst”，也支持与gdb匹配的“步骤”。

为了简单起见（以及由于作者对gdb的熟悉），我们通常描述与gdb兼容的lldb命令名。

最后，请注意，大多数命令可以缩短为单字母或双字母。例如，“s”将被解释为“step”命令。

有关gdb到lldb命令的详细映射，请参见:

“GDB到LLDB命令映射”[4]

虽然我们可以单步执行二进制文件的每个可执行指令，但这相当乏味。另一方面，简单地指示调试器允许被调试器运行（不受限制），从一开始就违背了调试的目的。“解决方案”？断点！

断点

断点是调试器的“命令”，指示调试器在指定位置停止执行。通常在二进制文件的入口点（或其构造函数之一）、方法调用或相关指令的地址上设置断点。如前所述，一旦调试器停止执行，就可以检查被调试器的当前状态，包括其内存和CPU寄存器内容、调用堆栈等等。

使用breakpoint命令（简称b），可以在指定的位置（例如函数或方法名）或地址设置断点。

假设我们想要调试一个恶意样本（malware.bin），并且想要在其主要功能处停止执行。启动lldb调试会话后，我们可以简单地键入b main:

```
(lldb) b main
断点1：其中=恶意软件。宾曼，
              地址= 0x0000000100004bd9
```

设置此断点后，如果我们随后指示调试器（通过run命令）运行被调试的进程，执行将开始，但在到达主函数（开始）处的指令时停止:

```
（lldb）运行

（lldb）进程1953停止原因=断点1.1
->  0x100004bd9 <+0>: pushq  %rbp
```

很大一部分Mac恶意软件是用Objective-C编写的，这意味着（即使是以编译的形式）它将同时包含类名和方法名。因此，我们还可以在这些方法名上设置断点。怎样只需将完整的方法名传递给breakpoint（b）命令即可。例如，要中断NSDictionary类的objectForKey: method,请键入：b -[NSDictionary objectForKey:]。

> 笔记:
>
> 有关调试Objective-C代码的深入讨论，请参阅优秀的writeup: “Dancing in the Debugger – A Waltz with LLDB” [5]

如前所述，断点也可以按地址设置（例如，用于在函数中设置断点）。要在地址上设置断点，请指定前面带有0x的（十六进制）地址。例如，我们还可以通过：b 0x0000000100004bd9在恶意样本（位于地址0x0000000100004bd9）的主函数上设置断点。

断点可以通过下面描述的命令列出或删除:

（lldb）命令	说明
断点（b）<函数/方法名称>	在指定的函数或方法名上设置断点。
断点（b）<地址>	在指定内存地址的指令上设置断点。
断点列表（br l）	显示（列出）所有当前断点。
断点启用/禁用（br e/dis）	启用或禁用断点（由数字指定）。
断点删除<#>	删除断点（由数字指定）。

help命令（带有断点参数）提供了一个完整的

断点相关命令：

```
（lldb）帮助断点
语法：断点<子命令> [<命令选项>] 支持以下子命令：

清除 --      删除或禁用与指定源文件和行匹配的断点。
命令 -- 用于添加、删除和列出遇到断点时执行的LLDB命令的命令。

删除 -- 删除指定的断点。
            如果未指定断点，请将其全部删除。
禁用 -- 禁用指定的断点而不删除它们。
            如果未指定任何断点，请禁用所有断点。使可能 -- 启用
指定的禁用断点。
            如果未指定断点，请启用所有断点。
列表      -- 以可配置的详细级别列出部分或所有断点。修改 -- 在中修改断点或断点集上
的选项
            可执行文件。如果未指定断点，则对上次创建的断点执行操作。
姓名      -- 用于管理断点的名称标记的命令
阅读      -- 读取并设置以前保存到带有“断点写入”的文件中的断点。
设置      -- 在可执行文件中设置一个或多个断点。写 -- 将列出的断点写入可读入
的文件
            使用“断点读取”。如果没有参数，则写入所有断点。
```

笔记:

有关lldb支持的断点命令的更多信息，请参阅

“断点命令”[6]

检查所有的东西

在本章的开头，我们看了一个例子来强调调试器的功能。具体来说，我们演示了如何转储已解密并（仅）存储在内存中的恶意软件配置信息。

到目前为止，我们已经讨论了流控制（即单步执行指令）和设置断点。然而，我们还没有讨论指示调试器显示CPU寄存器或被调试器内存的状态。这种强大的功能允许人们检查运行时信息或“状态”，而这些信息在静态分析期间通常（直接）不可用。例如，（如我们所见），查看恶意软件在内存中解密的配置信息。

要转储CPU寄存器的内容，请使用寄存器读取命令（或缩短的reg r）。要查看特定寄存器的值，请传入寄存器名（前缀为$）作为最终参数：

```
（lldb）注册号为$rax
rax = 0x0000000000000000
```

一般来说，我们更感兴趣的是寄存器指向什么 ...也就是说，检查实际内存地址的内容。内存读取或（gdb兼容）x命令可用于读取内存内容。但是，除非我们明确指定数据的（预期）格式，否则lldb将只打印原始（十六进制）字节。

有多种格式说明符（用于读取内存），指示lldb将指定的内存地址视为字符串、指令等：

（lldb）命令	说明
x/s <reg/memory address>	将内存显示为以null结尾的字符串。
x/i <reg/memory address>	将存储器显示为汇编指令。
x/b <reg/memory address>	将内存显示为字节。

还可以指定应显示的项目数。例如，要在指定地址反汇编指令，请键入：x/10i <address>。

最强大的显示命令是“打印对象”（po）命令。此命令可用于打印任何Objective-C对象的内容（Objective-C术语中的“描述”）。例如，在本章开头的示例中，在[SBConfigManager setDefaultConfiguration:]方法中，恶意软件将其配置信息解密为RAX寄存器引用的Objective-C对象。

因此，使用print object（po）命令，我们可以打印对象的详细描述，包括所有键/值对:

```
（lldb）打印对象$rax
{
 dnsChanger = { “附属公
   司”=”；
   "blacklist_dns" = ();
   “加密”=真；
   "external_id" = 0;
   "product_name" = dnsChanger;
   "publisher_id" = 0;

    ...

   "setup_dns" =         (
      "82.163.143.135",
      "82.163.142.137"
    );

   ...
```

笔记:

给定任意值或地址，如何知道使用哪个显示命令？也就是说，它是指向Objective-C对象的指针吗？还是绳子？还是一系列指令？

如果要显示的值是一个参数或记录的API的返回值，则其类型将在其文档中注明。例如，苹果的大多数Objective-C API或方法返回对象，因此应该通过“打印对象”（po）命令显示。

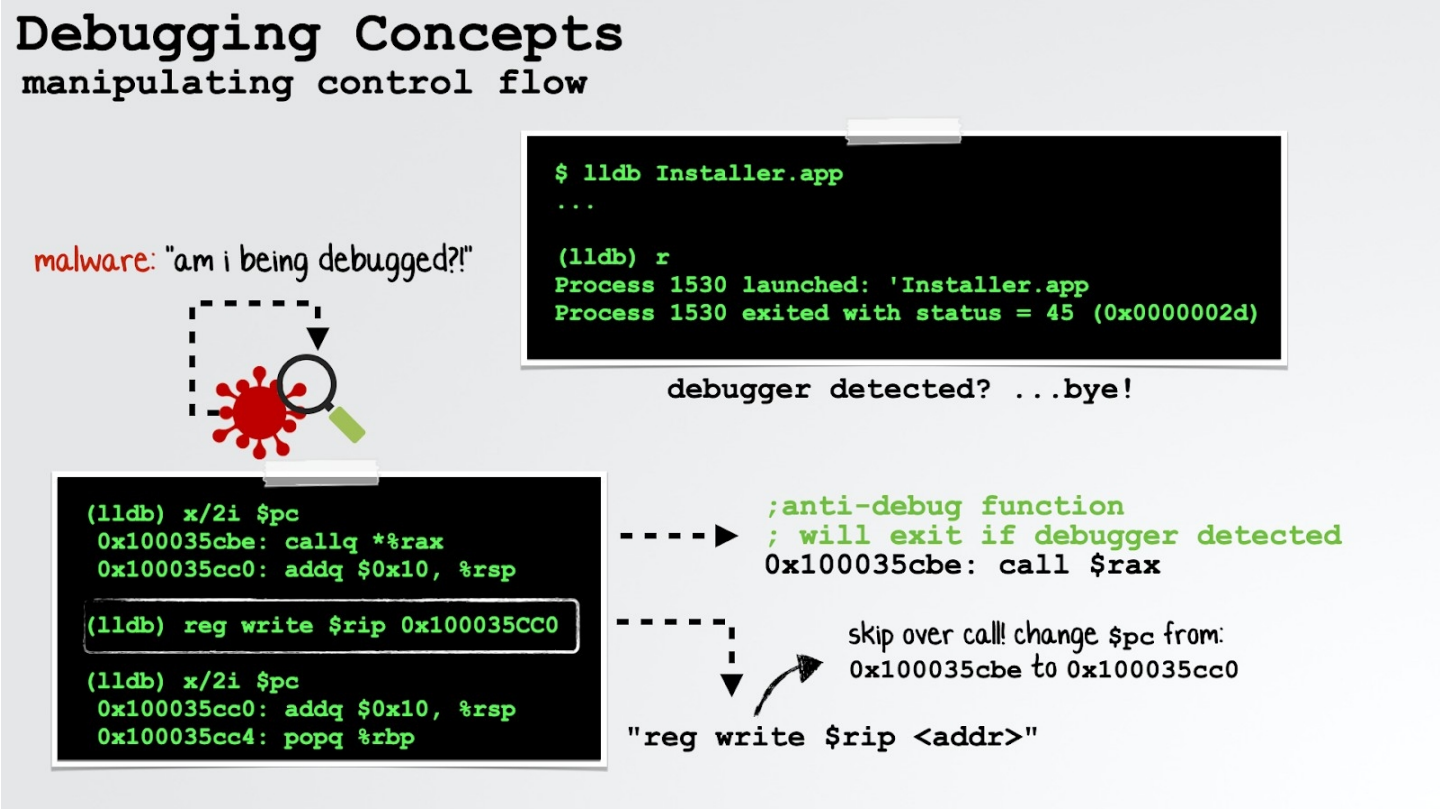
然而，如果没有可用的上下文，它实际上可以归结为尝试和错误。“print object” lldb命令不能产生有意义的输出吗？也许可以尝试x/b将其转储为原始（十六进制）字节。

修改进程状态

本章将介绍的最后一个调试主题涉及修改已调试进程的状态和操作控制流。通常在调试会话期间

分析是相当被动的（而不是设置断点来停止在不同位置的执行）。然而，通过直接修改流程的状态甚至控制流，可以更“侵入性”地与流程交互。这在分析实现反调试逻辑（将在下一章中讨论）的恶意样本时尤其有用。

如下图所示，一旦找到此类反分析逻辑，就可以通过修改指令指针来指示调试器跳过代码：



修改调试对象状态的最常用方法是更改CPU寄存器值或内存内容。寄存器写入命令可用于更改前者的值，而内存写入命令可修改后者。

寄存器写入（或缩短的`reg write`）命令接受两个参数：目标寄存器及其新值。例如，在上图中，我们跳过了对实现反调试逻辑（`0x100035cbe: call $rax`）的函数的调用，方法是首先在调用上设置断点，然后修改指令指针（`rip`）以指向下一条指令（`0x100035cc0`）。

```
（lldb）注册写入$rip 0x100035cc0
```

这确保了永远不会调用调用（地址为0x100035cbe），因此永远不会执行恶意软件的反调试器逻辑 ...这意味着我们的调试过程可以畅通无阻地继续下去！

修改CPU寄存器值以影响调试过程还有其他原因。 例如，想象一个恶意软件在持续安装自己之前试图连接（签入）到远程命令和控制服务器。如果服务器（现在）处于脱机状态，但我们希望恶意软件继续执行（以便观察其自身的安装方式），我们可能必须修改包含此连接检查结果的寄存器。由于函数调用的返回值存储在rax寄存器中，这可能涉及将rax（函数调用后）的值设置为1（true），从而使恶意软件相信连接检查成功：

```
（lldb）注册写入$rax 1
```

放松点！

如前所述，我们可以通过memory write命令更改任何（可写）内存的内容，从而“侵入式”操纵调试对象。苹果公司的《LLDB入门指南》（Getting Started with LLDB）[7]指出，该命令允许用户“写入正在调试的进程的内存”。

该命令在恶意软件分析期间有用的一个例子是更改加密配置文件（仅在内存中解密）的默认值。
这种配置可以包括触发日期，该触发日期指示恶意软件保持休眠状态，直到遇到所述日期为止。要强制立即进行活动（观察恶意软件的完整行为），可以直接将内存中的触发日期修改为当前时间。

另一个例子是修改保存远程命令和控制服务器地址的内存。这为分析人员指定备用服务器提供了一种简单（非破坏性）的方法 ...很可能是在他们的控制下。

> 笔记:
>
> 能够修改恶意软件的命令和控制服务器（或指定备用服务器）的地址，有它的特权！
>
> 在一篇题为“攻击性恶意软件分析：通过定制C&C服务器剖析OSX/Frutfly.b”的研究论文[8]中，我举例说明了恶意软件如何连接备用服务器

（在分析员的控制下）可以被分派任务以展示其能力。

“恶意软件分析是一个耗时且往往费力的过程。虽然静态分析和调试等传统分析技术可以揭示恶意软件样本的全部功能，但可能有更好的方法。在本研究论文中，我们通过创建自己的自定义命令a来全面分析一个有趣的macOS恶意软件nd控制（C&C）服务器。结合各种监控工具，通过这台服务器，我们可以简单地对恶意软件进行任务处理，以迫使其暴露其全部功能。” [8]

内存写入命令的格式通过lldb的help命令描述:

```
（lldb）帮助内存写入

写入当前目标进程的内存。

语法：内存写入<cmd options> <address> <value> [<value> [...]]命令选项

用法:
  内存写入[-f <format>] [-s <byte size>] <address> <value> [<value> [...]]内存写入-i
  <filename> [-s <byte size>] [-o <offset>] <address> <value>
[<value> [...]]

       -f <format> ( --格式<格式>)
            指定要用于显示的格式。

       -i <filename> ( --infile <filename> )
            使用文件内容写入内存。

       -o <offset> ( --偏移量<offset>)
            开始从输入文件中的偏移量写入字节。

       -s <byte-size> ( --大小<字节大小>)
            以选定格式显示时使用的大小（字节）。
```

例如，要将地址0x100100000. 的内存更改为0x414141. 可以编写：memory write 0x100600000. -s 4. 0x414141. 然后可以使用内存读取命令确认修改:

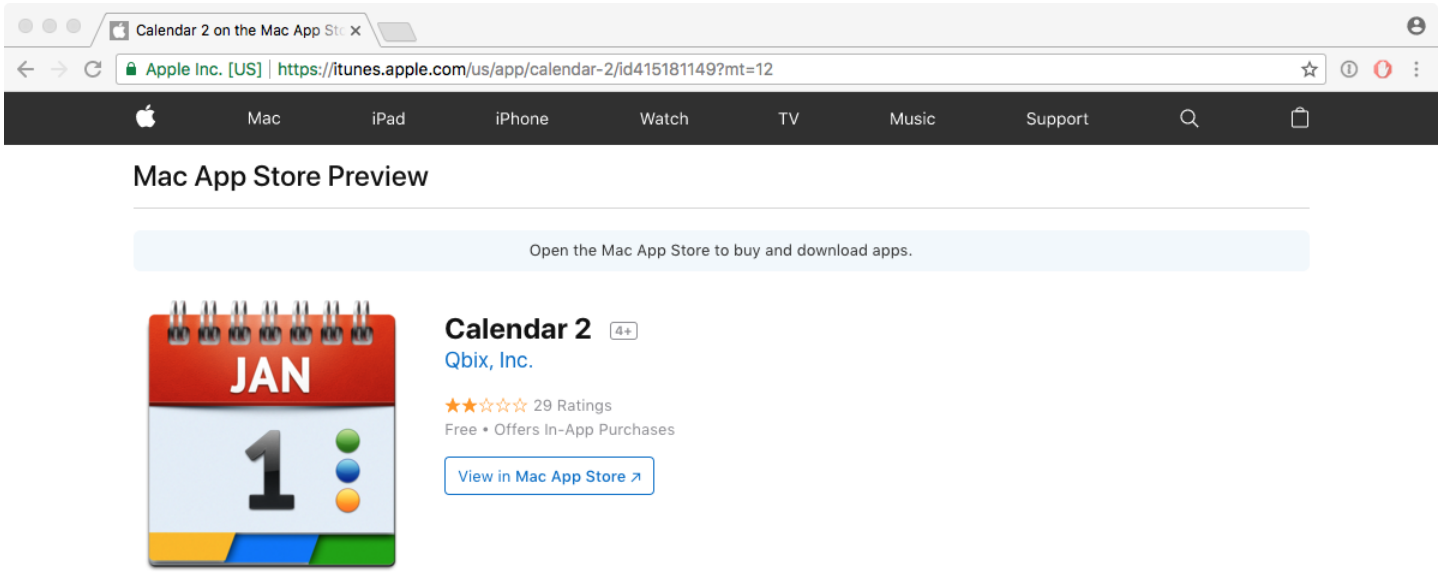
```
（lldb）内存写入0x100600000 -s 4 0x414141（lldb）内存

读取0x100600000
```

```
0x100600000: 41 41 41 41 00 00 00 00 00 00 00 00 00 00 00 00  AAAA............
```

调试会话示例

在本章结束时，让我们简单地看一下另一个实际示例，它演示了我们刚才讨论的许多调试概念。

2018年初，一款名为“Calendar 2”（或CalendarFree）的流行应用程序在Mac官方应用商店中被发现包含（相当秘密地）在用户电脑上挖掘加密货币的逻辑[9]。虽然不是恶意软件本身，但该应用程序提供了一个说明性的案例研究，展示了调试器如何提供对二进制文件的全面理解，甚至揭示隐藏的或颠覆性的功能。



在静态分析分类过程中，发现了一些有趣的方法调用，例如对-[MinerManager runMining]的调用，该调用反过来调用了+[Coinstash_XMRSTAK.Coinstash startMiningWithPort:password:coreCount:slowMemory:currency:]:

```
01  /* @class MinerManager */
02  -(void)runMining {
03      rdx = self->_coreLimit;
```

```
    r14 = [自计算工作核：rdx]；
    [Coinstash_XMRSTAK9Coinstash setCPULimit:self->_cpuLimit];
    r15 = [self getPort];
    r12 = [自我算法]；
    [self getSlotMemoryMode];

    [Coinstash_XMRSTAK9Coinstash startMiningWithPort:r15
                                 password:self->_token
                                 coreCount:r14
                                 slowMemory:self->_slowMemoryMode
                                 currency:r12];
    ...

    返回；
}
```

分析的目标之一是发现加密货币账户。原因是，在调试会话中，在runMining方法（或startMiningWithPort:password:coreCount:slowMemory:currency: method）上设置断点会显示此信息。

启动lldb,我们可以首先在-[MinerManager runMining]方法上设置一个断点:

```
$ lldb CalendarFree.app
（lldb）目标创建“CalendarFree.app”
当前可执行文件设置为“CalendarFree”。应用程序（x86_64）。

(lldb) b -[MinerManager runMining]
断点1: where = CalendarFree`-[MinerManager runMining],地址=
              0x0000000100077fc0
```

设置断点后，我们指示调试器运行应用程序（通过r命令）。正如所料，它会在我们设置的断点处停止:

```
(lldb) r
782进程免费启动。app/Contents/MacOS/CalendarFree（x86_64）

日历自由[782:7349]矿工：已停止进程782已
停止
停止原因=断点1.1
```

```
CalendarFree`-[MinerManager runMining]:
-> 0x100077fc0 <+0>: pushq %rbp
    0x100077fc1 <+1>: movq   %rsp, %rbp
    0x100077fc4 <+4>: pushq %r15
    0x100077fc6 <+6>: pushq  %r14
```

让我们（一步一步地）看一下说明，直到我们接到电话`+[Coinstash_XMRSTAK.Coinstash startMiningWithPort:password:corecont:slowMemory:currency:]`方法。

因为我们想在`startMiningWithPort`之前跳过（其他）方法调用：`...` 方法，我们使用`nexti`（或`n`）命令。

最终我们可以调用`startMiningWithPort: ...` 方法回想一下，`Objective-C`（和`Swift`）调用是通过`objc_msgSend`函数进行的。

在调试器中可以观察到这一事实：`objc_msgSend`函数的地址被移动到`r13`寄存器中（在`0x100077fe3`处找到的指令处）：

```
(lldb) n

进程782已停止
 停止原因=指令跳过

 CalendarFree`-[MinerManager runMining] + 35:
 ->  0x100077fe3 <+35>: movq 0xaa3d6(%rip), %r13   ;0x00007fff58acba00: objc_msgSend
```

对`startMiningWithPort`的实际调用：`...` 方法（通过`objc_msgSend`函数，谁的地址`recall`存储在`r13`寄存器中）发生在地址`0x100078067:`

```
(lldb) n

进程782已停止
 停止原因=指令跳过

CalendarFree`-[MinerManager runMining] + 167:
->  0x100078067 <+167>: callq  *%r13
```

```
（lldb）注册号为13美元
r13 = 0x00007fff58acba00  libobjc.A.dylib`objc_msgSend
```

注意，通过reg read命令，我们确认调用的目标（在r13寄存器中找到）确实是objc_msgSend函数。

正如我们在关于静态分析的第0x7章中所讨论的，在调用objc_msgSend函数时，寄存器及其值如下所示：

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

通过调试器，很容易确认这一点！

第一个参数（保存在rdi寄存器中）是调用该方法的类（或实例）。在静态分析分类期间，这被确定为一个名为“Coinstash_XMRSTAK.Coinstash”的类。使用“print object”（po）命令，我们可以动态地看到是的，这确实是正确的：

```
(lldb) po $rdi
Coinstash_XMRSTAK.Coinstash
```

第二个参数将是一个（以null结尾）字符串，它是要调用的方法的名称。通过静态分析，我们知道这应该是“startMiningWithPort”：
...“。要打印（以null结尾的）字符串，我们使用带有“s”格式说明符的x命令：

```
(lldb) x/s $rsi
0x1000f1576: "startMiningWithPort:password:coreCount:slowMemory:currency:"
```

笔记:

调用objc_sendMsg函数时，rsi寄存器将方法的名称保存为以null结尾的（“C”）字符串。

如果我们试图打印出这个字符串，但没有指示lldb它的格式（一个以null结尾的字符串），lldb只会以默认格式显示一个无符号长字符串:

（lldb）打印$rsi
（未签名的长款）$1=42955830

有几个选项可以让实际的方法名显示为字符串。其中包括:

- x/s $rsi
  (lldb) x/s $rsi
  0x1000f1576: "startMiningWithPort:password:coreCount:slowMemory:currency:"
- 打印（字符*）$rsi
  （lldb）打印（字符*）$rsi
  （字符*）$1 = 0x00000001000f1576
  "startMiningWithPort:password:coreCount:slowMemory:currency:"
- 注册阅读$rsi
  （lldb）注册号为$rsi
  rsi = 0x00000001000f1576
  "startMiningWithPort:password:coreCount:slowMemory:currency:"

最后一个选项（reg read命令）将自动计算
$rsi是指向以null结尾的字符串的指针。

类（rdi）和方法名（rsi）后面是该方法的参数，例如端口、密码和货币。通过静态分析方法，这些参数的值不容易确定。然而，通过调试器，这是轻而易举的事。

参考上表，我们看到参数将在rdx、rcx、r8、r9寄存器中，然后（由于此方法需要4个以上的参数），最后一个参数将在堆栈（rsp）上找到。让我们看一看:

```
(lldb) po $rdx
7777

(lldb) po $rcx
qbix:greg@qbix.com

（lldb）注册号为$r8
r8 = 0x0000000000000001

（lldb）po
$r9始终

(lldb) x/s $rsp
0x7ffeefbfe0d0: "Ugraft"
```

使用各种显示命令（po、reg read等），我们能够看到传递给cryptocurrency框架方法的值，提供我们分析所寻求的洞察力（以及一些可能可归因于帐户的信息）！

下一个...

在本章中，我们详细介绍了调试器 ...可以说是分析最复杂恶意软件威胁的最彻底、最全面的方法。

具体来说，我们展示了如何逐个指令调试二进制代码，同时检查（或修改）寄存器和内存内容，跳过（绕过）函数，等等。有了这种分析能力，恶意软件就没有机会了！

当然，如果你是一个恶意软件作者，你不必担心你的恶意创作会被琐碎地解构。那你该怎么办？在下一章中，我们将深入研究恶意软件作者使用的各种反分析逻辑，这些逻辑旨在阻止（或至少使）分析工作复杂化。

参考文献

1. OSX.Mami
https://objective-see.com/blog/blog_0x26.html

2. lldb
https://lldb.llvm.org/

3. lldb教程
https://lldb.llvm.org/use/tutorial.htm
l

4. “GDB到LLDB命令映射”
https://lldb.llvm.org/use/map.htm
l

5. “在调试器中跳舞-与LLDB跳华尔兹”
https://www.objc.io/issues/19-debugging/lldb-
debugging/

6. “断点命令”
https://lldb.llvm.org/use/map.html#breakpoint-
commands

7. “LLDB入门”
https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/gdb_to_l
l db_transition_guide/document/lldb-basics.html

8. “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

9. “Mac应用商店里一个秘密的加密货币矿工？” https://objective-
see.com/blog/blog_0x2B.html