2016 年 6 月 1 日，星期三

研究热点：ROPMEMU - 复杂代码复用攻击分析框架

作者：Mariano Graziano。

执行摘要

近些年来，攻击变得越来越复杂。威胁形势的演变已经证明了这一点：面对更强的缓解措施，网络攻击者只能改变策略，通过绕过缓解措施来感染系统。代码复用攻击（例如面向返回的编程 [ROP]）是这种演变的一部分，也是防御者目前所面临的挑战，因为这是一个尚未深入研究的研究领域。今天，Talos 发布了用于分析复杂代码复用攻击的框架 ROPMEMU。在这篇博文中，我们将确定并讨论对这些代码复用实例进行反向工程的挑战和重要性。我们还将介绍此框架的方法和组件，用于分析这些攻击并简化分析。

代码复用攻击既不新鲜也不奇特。这种攻击从 1997 年第一起 ret2libc 攻击得到证实起就已出现。从那以后，随着代码注入攻击因软件和硬件缓解措施的数量不断增加而越来越难成功，网络攻击者逐渐转向代码复用攻击。防御措施改善所伴随的一个结果是，攻击者为了绕过防御措施而开发出更复杂的攻击。近年来，恶意软件的编写者也开始采用面向返回的编程 (ROP) 范式来隐藏恶意功能并阻挠分析。如果读者不熟悉 ROP 但希望获得更多了解，请阅读 Shacham 的相关阐述。

遗憾的是，对诸如 ROP 之类代码复用攻击的分析一直被人们完全忽略。虽然只有少数公之于众的例子能证明这些攻击的复杂性，但是毫无疑问，网络攻击者今后会继续利用这种攻击手段。对防御者而言，目前能够用于分析这类威胁的工具少之又少。这正是 Talos 开发 ROPMEMU 的主要动因之一。

Talos 提供的框架采用一系列不同的方法分析 ROP 链，并以可由传统反向工程工具分析的方式重新构建其等效代码。具体而言，该框架基于内存调查分析（因其输入为物理内存转储）、代码仿真（用于如实重建原始 ROP 链）、多路径执行（用于提取 ROP 链负载）、控制流图 (CFG) 恢复（用于重建原始控制流）和多次编译器转换（用于简化 ROP 链的最终指令）。在整篇帖子中，我们以 Chuck（一种持续型面向返回的编程 [ROP] rootkit）来测试 ROPMEMU。

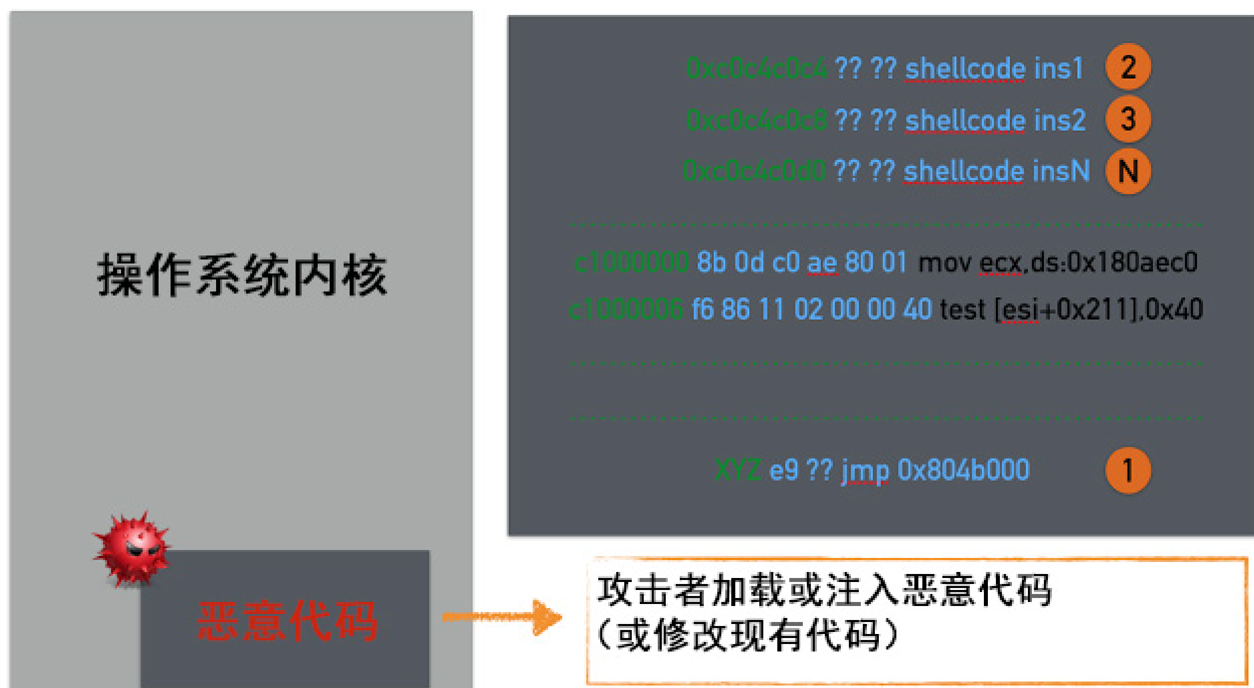
ROPMEMU 是实现代码复用攻击自动分析的第一步。因为此研究领域仍处在探索阶段，而且 ROPMEMU 是研究原型，所以该框架确实还缺乏一些功能来支持对一般输入进行运算，以及处理所有可能存在的代码复用实例。但是，随着我们继续研究并进一步开发此框架，我们相信它可以在研究如何应对此类威胁的过程中成为一个宝贵的工具。

ROPMEMU 已于第 11 届计算机与通信安全亚洲会议 (ASIACCS) 上发表，相关论文可于此处下载。

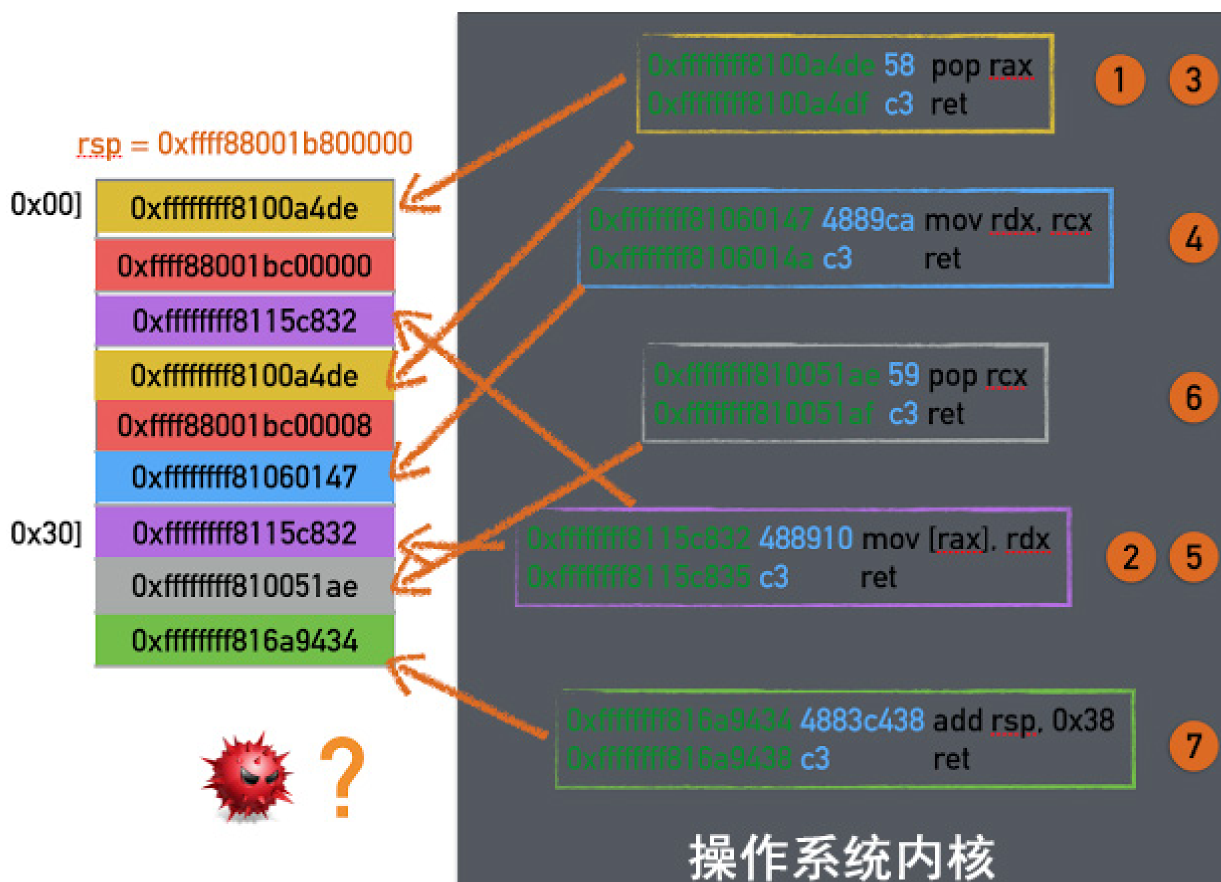
ROP 101

内存调查分析主要着眼于在物理内存中查找入侵证据。证据通常包括已经通过恶意组件创建或注入到内存中的工件。Volatility 插件（例如 psxview 和 malfind）就是典型的有助于确定感染或入侵内存转储的工件的工具。

多年来，攻击者的一贯手法都是注入代码，并劫持目标应用或目标操作系统的正常控制流。为应对此类攻击，整个业界开发出相关工具来确定可能发生此类攻击的时间。典型的代码注入场景如下所示。



在上图中，我们看到 shellcode 在内存中被安排为一个连续的代码块，我们使用跳板代码跳到该代码块（第 1 步）。现代操作系统通常会部署防护或缓解措施使数据区域中无法执行 shellcode，而且一般情况下代码注入攻击如今也不可行。因此，攻击开发者想出了新的手段来绕过这些对策。具体而言，与传统代码注入攻击相比，ROP 攻击的情况大不相同：



主要区别在于指令在内存中并不连续。ROP 是一种通过重新利用程序中已有指令来执行任意代码的方法。每个指令序列（亦称指令片段）负责获取下一个指令序列的地址（通常来自栈并使用 ret 指令），并将许多小段的指令片段拼接到一起执行预定义的计算。在上面显示的示例中，左侧是处于攻击者控制下的内存区域，而右侧各个框中的内容则是指令片段。框旁的数字表示指令片段的执行顺序，这些指令片段在内存中并不是连续的。执行这些指令片段需要跳到包含可执行代码的地址空间的不同位置（以橙色箭头说明）。这样，攻击者所控制的数据区域就只包含指针。最终就能绕过典型的漏洞缓解措施。

鉴于这些攻击的性质，普通代码复用攻击（例如 ROP）为分析师带来了新的挑战。

ROP 分析

多年来，研究人员完全低估了 ROP 代码的复杂性，而且几乎没有多少研究以发现内存中的 ROP 链为重点。但是，这是一个十分吸引人的研究领域，仍然处于起步阶段。我们在 ROPMEMU 开发过程中确定的主要挑战表明，这一领域还大有可为。

挑战 1：冗长

大多数 ROP 指令片段包含假指令。例如，意图增加 EAX 的指令片段也可能会从栈中提取一个值之后再调用 RET 指令（触发 ROP 链中的下一个指令片段）。此外，ROP 链的代码包含很大比例的返回指令或其他间接控制流指令，其唯一目标就是将指令片段连接到一起。这几个例子大致说明了，ROP 代码为何如此冗长，并且包含大量增加分析师辨识难度的无作用代码。不过，这个问题可能是最简单的待解决问题，因为已经面市的编译器著作中提出了许多可简化汇编代码的转换。

挑战 2：基于栈的指令链

ROP 链与正常程序之间最明显的区别在于，ROP 链中的指令在内存中并不连续，而是分组为小段的指令片段并由间接控制流指令连接到一起。例如，典型的程序可能包含一个由 50 条指令组成的代码块，用于执行预定义的计算，而在 ROP 链中，这些指令可能分割为 40 多个指令块，由 RET 指令衔接到一起。乍一看，解决此问题似乎是小事一桩。因为 ROP 链中每个指令片段的地址都保存在内存中，人们可能误以为自动检索这些地址、收集相应的代码片段并用一个指令序列替换整个 ROP 链是件轻而易举的事。但是，基于栈的指令链会产生一个微妙的副作用，那就是难以通过简单的静态分析来确定。例如，由于指令片段的顺序保存在栈中，但每个指令片段的代码也会与栈交互（用于检索参数或只是因为假指令的原因），为了正确找到每个指令片段的地址，就有必要模拟代码中的每一个指令。

挑战 3：缺乏即时值

正常代码与 ROP 链之间的另一个区别是 ROP 链通常由“一般”指令片段（例如“在 RAX 寄存器中存储任意值”）构成，而这些指令片段需要对同样存储于栈中的参数进行运算。因此，分配给寄存器的绝大多数即时值在栈中与指令片段地址交错存在。同样，需要通过代码仿真来查找这些指令片段并将其恢复到其在代码中的原始位置。

挑战 4：条件分支

在 ROP 链中，分支条件意味着栈指针的更改而非更加传统的指令指针更改。这也就是说，可以使用跨多个指令片段的数十个不同指令对简单的条件转移进行编码（例如根据所需条件设置标记寄存器、测试其值并有条件地增加 esp 寄存器）。因此，要将 ROP 链转换为可读性更强的代码，需要根据其语义来确定这些模式并将其替换为单一的分支指令。

挑战 5：返回到函数

ROP 中的函数调用的实施通常只是返回到函数的进入点。但是，由于正常的指令片段通常也是提取自位于库内部的代码，所以要区分函数调用与另一个指令片段就变得很难。正如静态链接的二进制文件一样，缺乏有关外部库调用的信息可能使反向工程过程变得繁琐和复杂得多。

挑战 6：动态生成的 ROP 链

正常程序的指令通常位于可执行代码的只读部分。相反，在恶意软件中，动态修改的代码更常见（例如，作为打包的结果），这严重限制了对恶意代码进行静态分析的能力，也显著减慢了反向工程过程的速度。因为 ROP 链位于栈中而非只读存储器中，使用指令片段来准备其他指令片段的执行就非常简单。这种动态性意味着整个 ROP 链没有必要同时位于内存中。相反，动态创建 ROP 链（或部分 ROP 链）的情况十分常见。

挑战 7：停止条件

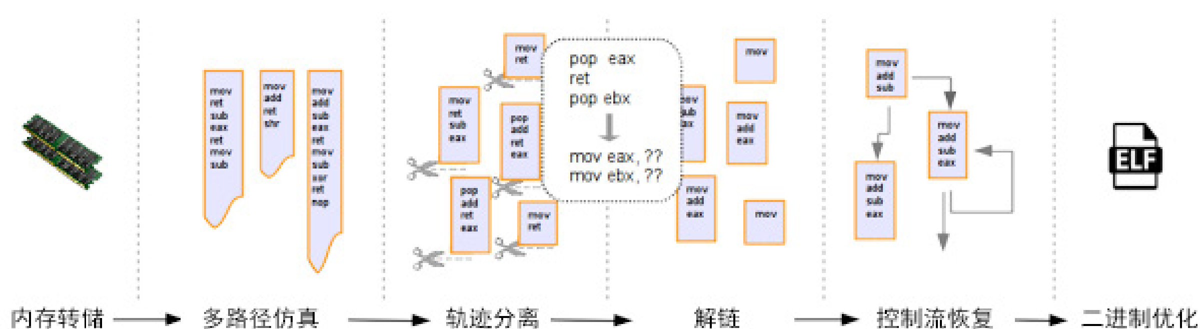
在我们的研究中，我们假设分析师能够找到 ROP 链在内存中的开始位置。然而，由于需要使用仿真程序来分析其内容，设定终止条件来确定何时所有指令片段都已提取并可停止仿真过程也非常重要。复杂的 ROP 链可以调用其间交错着正常指令片段的函数（函数还可依次调用其他函数）这一事实，以及 ROP 链可以在内存的不同部分动态生成另一个 ROP 链这一事实，使得此问题在一般情况下非常难以解决。例如，何时终止 ROP Rootkit（重新利用来自内核中现有代码的指令）并继续执行正常的内核任务？

后四个挑战尚未公开讨论过，因为 ROP 链在过去两年内才变得十分复杂，此前其复杂性尚不足以导致这些问题。

ROPMEMU

这些 ROP 链分析方面的挑战推动着 Talos 研究更好的分析方法和工具，最终开发出 ROPMEMU。ROPMEMU 是一款可帮助分析代码复用攻击的开源工具，可从 GitHub 获取。有关如何安装此工具以及如何使用所有不同插件和脚本的文档，可在维基页面找到。请记住，这是一项正在进行的研究项目，并应考虑到其尚处于 alpha 阶段。如果您发现任何问题，请创建支持请求，以便这些问题得以解决。如果您想贡献代码，敬请向我们发送您的拉取请求。

如下图所示，该框架包括五个主要分析阶段：



多路径仿真

此步骤模拟构成 ROP 链的汇编指令。要重建转储时正在运行的 ROP 链的确切实例，这是唯一的办法。探究所有可能存在的分支，并为每条执行路径生成独立的 JSON 轨迹（注明寄存器的值）。这样，我们就能应对以下两项挑战：基于栈的指令链和动态生成的 ROP 链。此轨迹包含 CPU 环境。该框架还会记录内存并有自己的影子栈。

所有写操作都会被拦截，而且所有后续读操作将从该区域获取数据。影子栈保存在另一个 JSON 文件中。仿真程序还可用于识别多个返回到库函数，略过其正文，而模拟其执行（应对“返回到函数”挑战）。它还可以实施探试法来检测 ROP 链边界并停止仿真。最初，我们实现了一个小的自定义仿真程序，它可以支持少数几个所需的 x86 和 x86_64 指令，并在每个指令后更新 CPU（寄存器和标记）和内存的状态。但是，为了支持整个指令集，我们后来改编了平台，以便使用最近发布的 Unicorn 仿真程序。Volatility 插件 **ropemu** 实现了这些概念并以 Unicorn 和 Capstone 为基础。该插件是概念验证，已经进行过全面的测试，并存在一些公开的 Unicorn 漏洞。

轨迹分离

在此阶段，系统将分析仿真程序生成的所有轨迹，删除重复的部分，然后提取唯一的代码块。此部分分为两步：首先，从每个分支点剪下每条轨迹，生成新的代码块并将其保存在单独的 JSON 轨迹中。在此操作期间，该框架还会记录描述不同代码块之间关系的元数据。第二步，链分离器比较各个代码块，以检测重叠的页脚指令（即，不同代码块末尾共有的指令片段）并将其隔离在单独的文件中。此阶段的结果是一组 JSON 轨迹，它们是 ROP 链的真正基本块。此部分在 Python 脚本 **blocks** 中实现。

解链

在此阶段，系统会应用一系列汇编代码转换，通过删除指令片段之间的连接并将连续指令片段的内容合并为一个基本块来简化每条 ROP 轨迹。只要删除所有 RET、CALL 和无条件 JMP 指令，就可以做到这一点。这一步还要负责删除来自栈的即时值并将其分配到相应的寄存器（用于应对分析基于栈的指令链和知悉即时值方面的挑战）。这可以简化 MOV 指令并将 POP 转换为 MOV 指令。通过从内存提取值可以做到这一点。此阶段的结果是目标架构中的二进制大对象。Volatility 插件 **unchain** 用于实现此阶段。

CFG 恢复

此环节将解链阶段生成的所有二进制大对象合并成一个程序，恢复 ROP 链的原始控制流图。此阶段包括两个步骤。

1. 将轨迹合并为一个图形表现形式。将有关代码块如何连接的信息指定为脚本的输入。此元数据信息在分离阶段生成。
2. 通过确定与分支条件关联的指令并将其替换为更传统的基于 EIP 的条件转移，将该图形转换为真正的 x86 程序（分析条件分支的挑战）。在我们的案例研究中，一个简单的条件转移由 19 个指令片段和 41 条指令实现。我们的框架能够识别此条件并在指令指针域中生成等效的汇编代码。将这 19 个指令片段转换为两个汇编指令：条件转移（在我们的案例中以 JZ 或 JNZ 表示）和无条件转移 (JMP)。

这前两步在 Python 脚本 **Glue** 中实现。CFG 恢复组件还负责检测和重新展开循环。ROP 链可能同时包含构建 ROP 链时以编程方式生成的面向返回的循环和未展开的循环。重新展开循环背后的逻辑在 **loops** 脚本中实现。此阶段结束时，（通过 **dust** 脚本）将程序保存在 ELF 文件中，使传统的反向工程工具（例如 IDA Pro）能够对其进行操作。

二进制优化

最后一步，我们应用已知的编译器转换来进一步简化 ELF 文件中的汇编代码。例如，此阶段会删除指令片段中的无作用指令，并生成经过优化的无干扰版本的负载（应对代码冗长的问题）。这些基本转换在不同的阶段实现。

有关更详细的分析，请参阅原始论文。

评估

我们用已公布的最复杂的 ROP 威胁对 ROPMEMU 进行了评估。在测试 ROPMEMU 时，我们的主要关注点放在提取 ROP 链、转换和 CFG 上。

第一个实验测试 ROPMEMU 的多路径仿真程序正确提取持续 ROP 链（*复制链*）和两个动态生成的 ROP 链（*调度程序链*和*负载链*）的能力。ROPMEMU 仿真程序能够自动检索全部三条 ROP 链的完整代码。复制链最长，有 414,275 条指令，但只包含一个基本块。缺乏控制流逻辑使得此 ROP 链类似于漏洞中使用的传统 ROP 负载，唯一的区别是它由 180,000 多个指令片段组成。其主要任务的结果如下：创建第一个动态组成部分（调度程序链）并将其复制到内存中。相反，*调度程序链*和*负载链*的指令片段数较少，但是其 CFG 更加复杂。具体而言，调度程序链包含三个分支和七个代码块。为了恢复整个代码，仿真程序生成了七条不同的 JSON 轨迹。同时，负载链由 34 个唯一代码块和 26 个分支点组成。这意味着 CFG 的逻辑流更复杂。此外，此负载链还调用了九个唯一的内核函数（`find_get_pid`、`kstrtou16`、`kfree`、`__memcpy`、`printk`、`strncmp`、`strrchr`、`sys_getdents` 和 `sys_read` - 最后两个函数被 `rootkit` 挂接），因此各不同执行路径上共有 17 个函数调用。

此实验证明，ropemu 可以探究并转储无法手动分析的复杂 ROP 链。（请注意，因为 Unicorn 问题，目前实现的 ropemu 在用于负载链时可能还有一些问题。）我们认为，这些 ROP 链反映了当前的恶意软件分析框架在应对面向返回的编程负载时的局限性。

下表总结了实验结果。

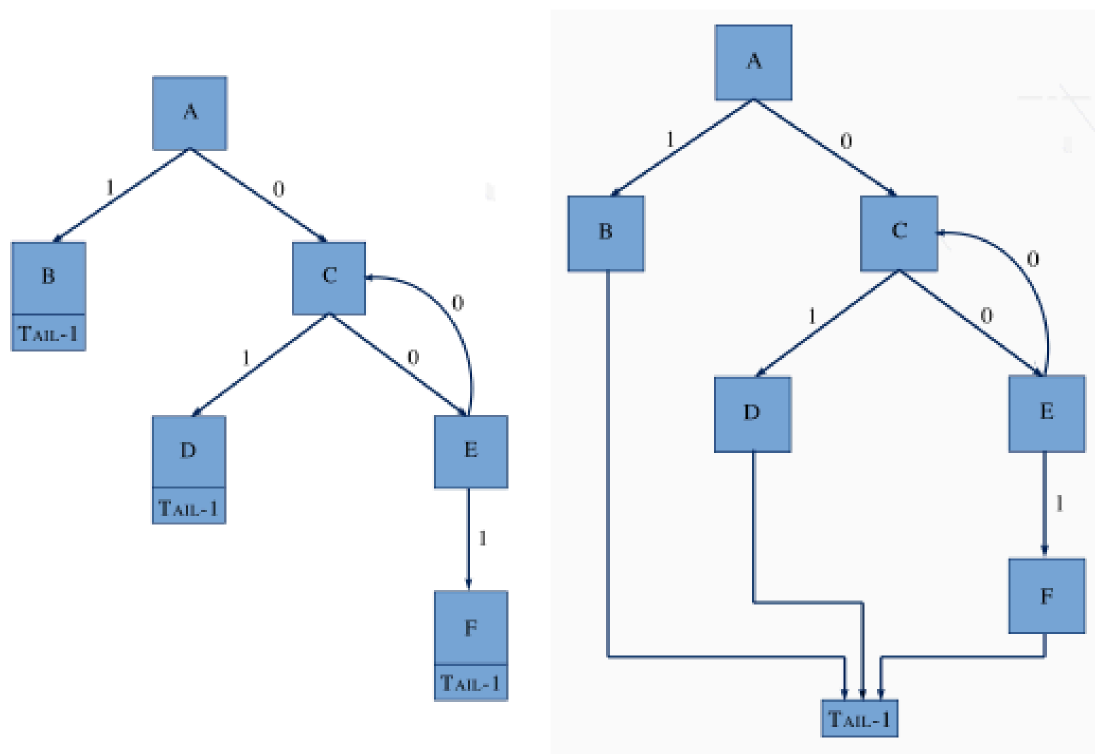
链	指令	指令片段	代码块	分支	函数	调用
复制	414,275	184,126	1	-	-	-
调度程序	63,515	28,874	7	3	1	5
负载	6320	2913	34	26	9	17

第二个实验显示了其他几个分析阶段对提取的 ROP 链的影响。具体而言，因为不可能显示完整的代码，我们展示了转换对负载大小的影响。下表总结了实验结果。

链	初始状态	解链阶段	CFG 阶段
复制	414,275	276,178	75
调度程序	63,515	40,499	16,332
负载	6320	3331	2677

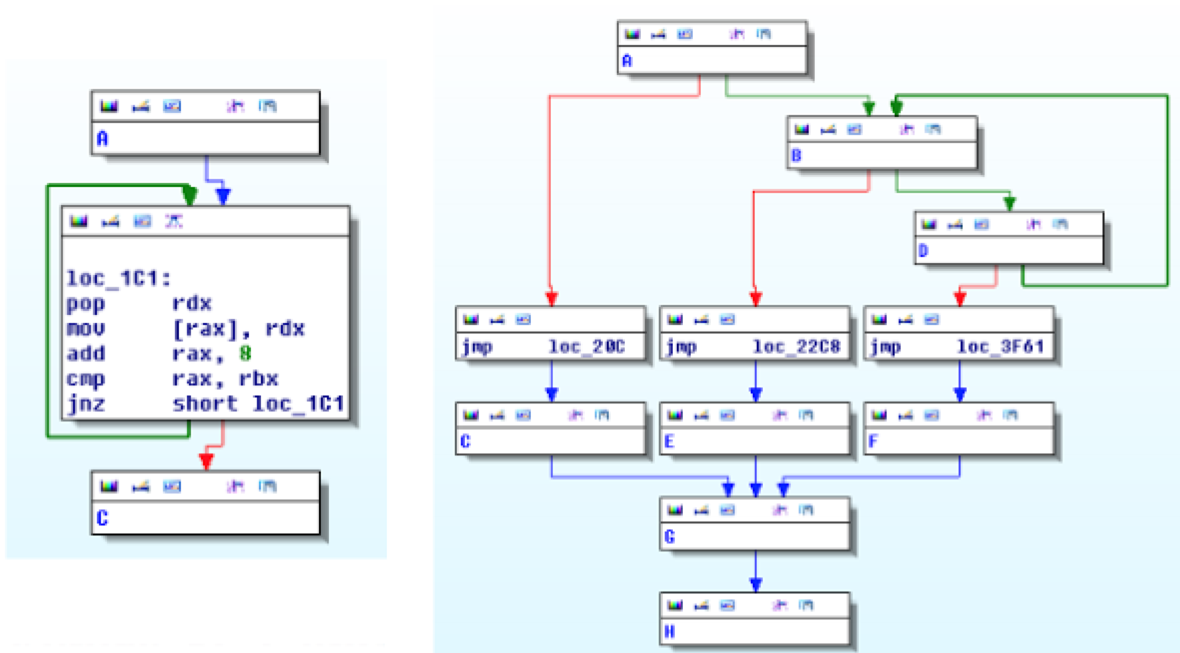
如第三列所示，解链环节显著缩小了 ROP 链的大小，平均缩小 39%。CFG 恢复环节过滤掉实施条件语句的指令，将 ROP 链从栈指针域转换到指令指针域，最后再应用循环压缩步骤。这些转换将复制链从大约 414,000 条指令缩小到仅仅 75 条指令。负载链受这些转换的影响较小，因为它包含的是 ROP 循环而非未展开的循环。

第三个实验测试 ROPMEMU 检索和恢复 ROP 链的控制流图的功能。首先，我们通过连接所有轨迹，显示了应用于*调度程序链*的控制流图的转换。在此情况下，该框架添加了汇聚节点。

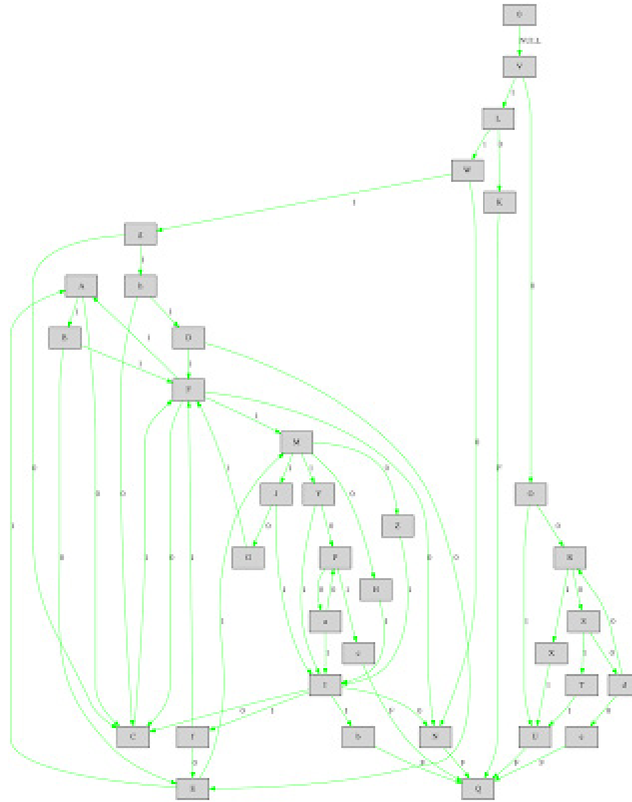


左侧的控制流图显示了未经任何转换的代码块。右侧所示为经过转换后的同一个控制流图。我们确定并添加了“tail”节点。

恢复 CFG 的第二步是对二进制大对象和生成的 ELF 文件进行操作。此 ELF 文件利用元数据信息连接所有代码块。为了测试此功能，我们使用 IDA Pro 打开了得出的文件。在下图中，左侧显示的 ELF 代表已完全转换为“基于 EIP 的”传统编程范式的*复制链*。这张图很简单，没有分支，图中突出显示的主要循环代表核心功能。相比之下，右图所示为 IDA Pro 中看到的*调度程序链*。为清晰起见，每个节点都已折叠起来以缩小生成的图。



*负载链*的 CFG 更加复杂，因为它包含恶意负载逻辑。其控制流图包含 30 多个代码块。我们可以从下一张图中看到，该控制流图包含两个主要代码块：sys_read 代码块和 sys_getdents 代码块。此外，该控制流图还显示了出口点（Q 和 C）和循环（可从回边轻易辨别出来）。即使该控制流图包含少量其他边（因循环简化引入的假优化而导致），下文所述信息也可对负载的行为提供快速但详细的概述。



限制

正如任何其他二进制分析工具一样，ROPMEMU 也存在许多内在局限性。具体而言，提供的解决方案结合了两项技术：内存调查分析和仿真。前者需要在 rootkit 加载到内存中之后获取物理内存转储，故此容易受制于反截获手段。反而，ROP 分析依赖于仿真程序的实现。基于仿真的解决方案的主要限制在于仿真程序本身的准确性。具体而言，这种方法很容易受制于专门针对指令副作用的反仿真手段。

当前的实现尚处于 alpha 阶段，我们已知的限制如下。

1. 此框架的当前版本只能完全支持 x86_64 系统且尚未进行全面测试，因为就我们所知，只有 ROP rootkit 的复杂性足以展示 ROPMEMU 的功能。
2. *ropemu*（Volatility 之上的仿真程序组件）的当前版本基于 Unicorn 并存在一些公开漏洞。论文根据该仿真程序的第一个自定义版本撰写。在论文完稿之后，我们决定重新编写 *ropemu* 以便利用 Unicorn 的独特功能。

结论

代码复用攻击（特别是 ROP）被广泛用作绕过操作系统防护的手段，而且复杂性与日俱增。恶意软件编写者已经开始使用 ROP 作为阻挠分析并在永无休止的“猫和老鼠”游戏中不被发现的手段。ROPMEMU 首次尝试自动分析完全使用 ROP 实施的复杂代码。迄今为止，我们已经使用提供的最复杂案例（一种持续型 ROP rootkit）对该框架进行了测试。

缺乏方法和工具来深入分析日益复杂的 ROP 负载是我们开发所提议框架的动因。虽然仍有许多工作有待完成，但 ROPMEMU 已经完成第一步的概念验证，我们希望整个业界以此为基础继续发展，帮助分析此类攻击。本研究说明了防御者未来将要面临的挑战以及这些威胁可能带来的复杂性。

您可以在 GitHub 上找到此工具和相关文档：
https://github.com/vrtadmin/ROPMEMU

发布者：Alexander Chiu；发布时间：下午 12:52
标签：代码复用，框架，恶意软件分析，研究热点，ROP，ROPMEMU