# ROP is Still Dangerous: Breaking Modern Defenses笔记

内存安全是个十分有历史的问题，自从DEP出现以来，它通过设置可写页不可执行来杜绝大多数经典的代码注入攻击，在其之后，ROP几乎成为了内存安全漏洞的标配，ROP并没有注入新的代码，而是将已经存在的指令序列链起来，以达到某种功能，因此并不会受到DEP的影响。

ROP是Return-into-libc的一个扩展，攻击者造成了程序返回到任意点，ROP有多个配件链接起来，每一个配件做一些小的计算，例如载入内存信息到寄存器中，或者两个寄存器做加法，ROP配件是在原始程序文本中查找的，使他们执行一个与原有功能不同的任务。 经典ROP中配件以ret结束，攻击者通过在栈中写合适的值来将他们串起来。每个配件以有用的指令，比如mov开头，以ret，jmp结尾。

对于ROP的防御也有不少，主要分为两类，第一类是移除二进制代码中潜在的配件或者强制进行控制流的检查，第二类是使用运行时保护透明地保护已经运行多年的二进制软件。

另外现有的操作系统中一般会开启ASLR，但是如果单个模块关闭了ASLR，则可只在这里找配件。或者可以使用信息泄露攻击来解除随机化。

本文将利用三种组合方法来突破现有的ROP防御手段，比如KBouncer和ROPecker。方法一用于打破传统的限制，在“返回地址为call指令的下一条指令”的检查机制下，要嵌入攻击是十分困难的（很多CFI依赖这样的策略来防止ROP）。此外大部分ROP攻击是由短配件组成的，不过也不能排除有长配件来嵌入攻击的，因此通过区分短配件序列和正常的执行流来防御ROP也不安全。最后，我们证明我们可以十分有效地清空历史记录。

我们可以通过使用这些方法来击垮KBouncer和ROPecker，这两种防御手段可以部署在已存在的硬件上，且几乎是没有性能损耗的，而且不需要重新编写软件。ROPecker加大了KBouncer的保护力度，它检查了更多的程序点，并且贯穿整个程序运行的始终。

## 攻击基元

我们先介绍三个攻击基元

1. ret返回到call下一条指令的ROP

因为return指令一般会返回到对应的call指令的下一条地址，因此可以过滤许多不符合这一条件的配件，我们经过评估发现只有6%的配件的return地址是紧跟着call的，因此很多人会很自然地认为使用这一策略可以增加ROP的困难。但是完全使用return地址紧跟call的配件也是可以实现的，只是配件可能会复杂一些，在我们的实验中，我们发现只要70k的二进制代码就足以构造满足要求的配件了。

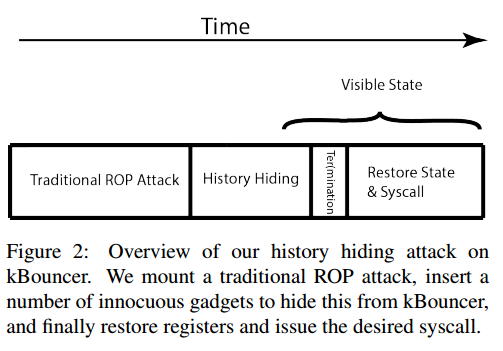
1. 逃避攻击

有些安全机制是基于分类的，通过运行时监视程序，区分正常程序片段或是配件。目前已有一些论文提出来通过程序序列长度实现分类，将短的程序片段视为配件，长的程序片段则认为是非配件。但是仍然存在方法可以让防御机制视配件为正常，比如说构造长配件或者使用长短配件混合的机制。

3.历史清空

一些安全机制通过检查某些关键点来检测攻击（如果检查所有的点，则会太消耗计算机性能），但是他们只保存了有限的历史信息，因此我们可以构造一些配件来来隐藏历史。我们在安全机制不运作的时候实施攻击，然后再通过一些看起来正常的指令来填充安全机制的历史记录，达到清空ROP攻击信号的效果。

我们可以通过以上这些机制来攻击之前提到两种防御方法



## 战胜KBouncer

kBouncer是通过检查某些关键点来实现防御的，当遇到syscall时，它通过读取LBR来检查过去16个间接跳转指令。首先它验证是否所有的return都会返回到call的下一条指令，另外如果检测到最近8个间接跳转指令，都被判别为配件，则终止进程，整个过程如Figure 2 所示。kBouncer十分地高效，因为它只在系统调用的时候进行检测。

**历史清除攻击**

kBouncer 如果检测不到异常，会将其状态标记为有效，也就是说，我们的攻击方法需要满足两个条件，第一点是所有的return地址必须是call指令的下一条，另外至少最后8个配件中有一个长配件。

我们的攻击分为三个步骤，第一部分称作初始化利用，也就是传统的rop攻击，我们将我们需要的参数值等通过这一部分都存入内存，为最后的攻击做准备，因为这一步没有syscall，因此并不会激活kBouncer的检测机制。

为了避开kBouncer的检测，在第二阶段我们需要覆盖历史记录，使其能通过检测，我们称其为隐藏历史阶段。我们将主要使用两种配件，其中一种是满足return地址紧跟call指令的要求的短配件，最理想化的情况是它不修改太多的内存。除此之外，我们也使用长配件，它至少有20条指令来保证不会被认为是配件。

我们先用短配件重复运行16次，来清空我们的历史寄存器LBR，但是，现在依然不是有效状态，因为最后8个跳转中至少有一个长的程序片段，如果这个时候启用kBouncer则会攻击失败。

然后我们选用长配件作为终止配件，因为当最后8个跳转中有一个长配件时，检测机制会认为它是正常的程序。终止配件还有一个重要的作用就是，原先的配件不一定将所有的参数存储就位，则可以通过这个长配件做最后的准备，以及恢复寄存器。在寻找长配件的时候我们要确保其中的条件跳转指令可控，另外就是它做内存读写时，不会引发进程的崩溃。

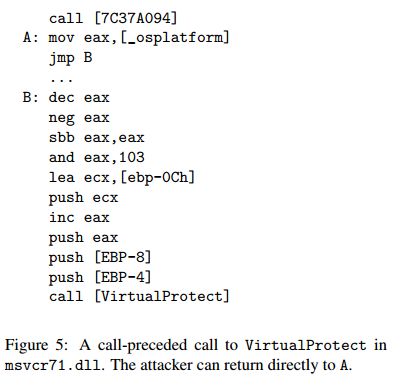
最后一个阶段是恢复寄存器。因为经过清空历史这一阶段，还是有一些参数会不可避免地被修改，为了保证最后的系统调用能够正常运行，我们可以用常规的方法将栈中的参数pop到我们需要的寄存器中，但是需要注意的是我们使用的这些return必须满足要求，并且不能超过8个，否则依然会变成无效状态。

为了弥补这一缺陷，我们需要一些不是return结尾的配件，比如利用JOP的jmp配件，COP的call配件等等，他们不用return结尾，就不用担心返回地址必须是call指令的下一个了指令了，而且像COP的call配件比符合条件return配件更普遍。

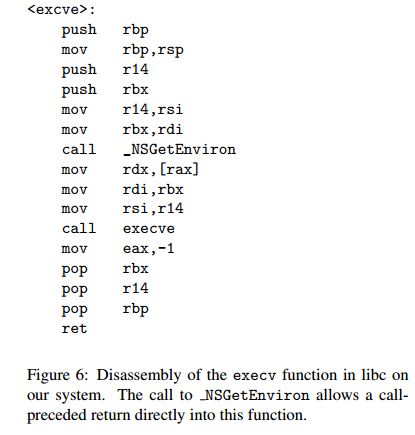
完成了以上步骤之后，我们可以发起syscall攻击了，这些系统调用一般都在libc或者kernel32库中。由于如果直接跳转到系统调用处，当执行完系统调用将返回不到原先的那个位置了，也就是说它的返回地址并不对应到call的下一条地址。对于如何合理地发起系统调用，我们也有三种方法。

第一种是使用反射配件，这是一个符合要求的配件，并且以寄存器间接跳转结尾，这样子我们就可以在更早的时候将系统调用的地址存入寄存器，然后在这里调用它。这是最简单的一种方式，但是这一配件比较难找。

第二种是，我们依然使用平时所用的那种方式，但是我们需要找到系统调用之前的那个call，我们称其为A，如果A执行了，则它在运行时自然会去调用我们想要的系统调用。



如Figure5 所示，当调用完[7C37A094]处的函数，它将返回来执行A处的代码，然后继续执行到B,以及最后的系统调用。



第三种方法则是返回到系统调用函数的内部，正好在一个call指令之后。如Figure6 所示，如果我们最初初始化一个eax来包含一个有效的环境指针，我们可以直接通过返回到(execv+18)调用execv函数。

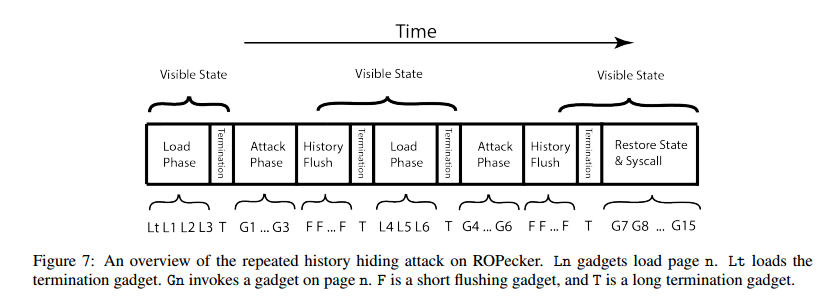
本文还探索了当存储历史为无限时的攻击，我们可以一直使用逃避攻击，也就是说只使用return地址返回到call的下一个地址的配件，同时不断地切换短配件和长配件，最后也能实现，但是构建配件将会困难一些，不过如之前所述，只要有70k的程序，就足够我们构成配件。

## 战胜ROPecker

ROPecker只允许一小部分内存页运行，当检查某一页正常时，再设另一个页为可执行页，然后将最近最少运行的页面设为不可执行，当不可执行页开始执行时，程序将会被暂停。

ROPecker比kBouncer更为复杂，它除了检测LBR中的数据，还会模拟执行程序，当发现到LBR中疑似为配件的数量和模拟执行过程中疑似为配件的数量的和超过某个阈值，将会认为这是一个攻击。

我们通过重复的历史隐藏攻击来欺骗ROPecker。



如Figure 7所示，我们利用 载入时期 载入有用的页，同时利用 攻击时期 调用在这些页上的配件，然后利用 清空时期 来清空历史记录。我们需要重复利用这三个部分来实现我们的攻击。

我们将介绍这三个攻击步骤，载入期间 将调用 页面调入 配件来将我们所需要的页面加入到可运行集合中，页载入配件是这个页中的任何一个满足返回要求的配件，它应该满足两个要求。第一，它必须使攻击者控制整个控制流。 第二，它必须不崩溃。 这两个要求并不难满足，因此我们可以轻易找到所需的页面载入配件。当运行完载入配件之后，我们将调用一个终止长配件，这个终止长配件可以阻止ROPecker向前进行探索。

在 攻击期间 我们将执行已经载入的可执行页中的攻击配件，由于攻击配件足够小，且要执行很多次，因此我们的每一个攻击片段并不需要完成所有的操作。

而在 历史隐藏 期间，在调用这些页上的配件之后，我们执行跟KBouncer中一样的清空方法来清空这里的LBR。

由于我们分了多个部分，且需要重复执行好多次，因此在阶段与阶段之间实现值的传递也是一个值得我们考虑的问题，如果存放在寄存器中，在过程中难免被后面的操作覆盖，因此我们通常将值存入内存，然后在下一个阶段将其取出来存入寄存器，接着再使用。

最后发起系统调用，这里比较特殊，即使我们的攻击配件没有在可执行集中，由于我们的攻击配件最多也不超过10条，因此并不会超过ROPecker的阈值（11条）

这两个保护手段在不同的场合下是十分有用的，但是他们依然还是存在一些能够被我们利用的弱点。本文主要目的是打破两个传统智慧，符合return规范的return指令是安全的以及ROP配件都是短的。日后的安全防护措施还是需要在这两方面进行一些保护。