1. 系统实现

我们在Intel Ivy Bridge平台下的Ubuntu 15.04系统上实现了一个BranchChecker的原型系统，如图1所示。该系统主要由一个Linux可装载内核模块构成，该模块会劫持Linux内核的PMI事件处理以及重要系统调用，并利用LBR信息以及应用内存信息进行CFI检测。

下面将对检测触发点及具体的CFI检测算法进行详细描述。

* 1. 检测触发点选择

前文中提到，最理想状态的检测触发点为用户态预测失败的跳转间接分支提交事件，但Ivy Bridge平台下并不支持该采样事件（支持的分支采样事件如表1所示），因此我们选择了最接近它的两个“超集”事件作为备选事件：用户态预测失败的跳转分支提交事件（事件编码为0xc5，掩码为0x20）和用户态预测失败的跳转间接分支执行事件（事件编码为0x89，掩码为0xac）。

**Table 1** Branch-related pmu sample event supported by Ivy Bridge

表**1** Ivy Bridge支持的分支采样事件

|  |  |  |  |
| --- | --- | --- | --- |
| 事件编码 | 掩码 | 事件名称 | 事件描述 |
| 0x89 | 0x01 | BR\_MISP\_EXEC.COND | Qualify conditional near branch instructions mispredicted. |
| 0x89 | 0x04 | BR\_MISP\_EXEC.INDIRECT\_JMP\_NON\_CALL\_RET | Qualify mispredicted indirect near branch instructions that are not calls or returns. |
| 0x89 | 0x08 | BR\_MISP\_EXEC.RETURN\_NEAR | Qualify mispredicted indirect near branches that have a return mnemonic. |
| 0x89 | 0x10 | BR\_MISP\_EXEC.DIRECT\_NEAR\_CALL | Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed. |
| 0x89 | 0x20 | BR\_MISP\_EXEC.INDIRECT\_NEAR\_CALL | Qualify mispredicted indirect near calls, including both register and memory indirect, executed. |
| 0x89 | 0x40 | BR\_MISP\_EXEC.NONTAKEN | Qualify mispredicted non-taken near branches executed. |
| 0x89 | 0x80 | BR\_MISP\_EXEC.TAKEN | Qualify mispredicted taken near branches executed |
| 0x89 | 0xFF | BR\_MISP\_EXEC.ALL\_BRANCHES | Counts all near executed branches (not necessarily retired) |
| 0xc5 | 0x00 | BR\_MISP\_RETIRED.ALL\_BRANCHES | Mispredicted branch instructions at retirement. |
| 0xc5 | 0x01 | BR\_MISP\_RETIRED.CONDITIONAL | Mispredicted conditional branch instructions retired. |
| 0xc5 | 0x04 | BR\_MISP\_RETIRED.ALL\_BRANCHES | Mispredicted macro branch instructions retired. |
| 0xc5 | 0x20 | BR\_MISP\_RETIRED.NEAR\_TAKEN | Number of near branch instructions retired that were taken but mispredicted. |

针对这两种事件类型，我们利用perf工具对随机选择的Spec2006部分程序进行了采样，采样数据如图2所示：

Fig.2 sample data of the two branch event on part of SPEC2006 benchmark

图3 部分SPEC2006程序上两种分支事件的采样数据

从上图可看出，用户态预测失败的跳转间接分支执行事件相比于用户态预测失败的跳转分支提交事件更少，因此我们选择用户态预测失败的跳转间接分支执行事件作为采样事件，并通过设置PMU计数器初始值，让其在发生该事件时计数器溢出，进而触发PMI事件。

* 1. ROP攻击检测

本文借助Intel CPU的LBR机制精确的获取了当前触发事件分支指令的地址及目标地址，并通过动态反汇编的方式判定当前分支是否为疑似gadget。

Intel 的 LBR记录有限，因此我们借助了两种硬件机制过滤掉了不必要的分支事件以免影响真实所需的分支数据：

1. 直接利用LBR本身的过滤机制进行分支采样过滤，Ivy Bridge支持的过滤方式如表2所示，通过该过滤机制，我们可以将LBR记录限定为用户态预测失败的跳转间接分支信息（LBR本身不记录不跳转的分支信息）。

**Table 2** Event filter mechanism supported by Ivy Bridge LBR

表**2** Ivy Bridge lbr支持的事件过滤机制

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 位偏移 |  | 描述 |
| CPL\_EQ\_0 | 0 |  | When set, do not capture branches occurring in ring 0 |
| CPL\_NEQ\_0 | 1 |  | When set, do not capture branches occurring in ring >0 |
| JCC | 2 |  | When set, do not capture conditional branches |
| NEAR\_REL\_CALL | 3 |  | When set, do not capture near relative calls |
| NEAR\_IND\_CALL | 4 |  | When set, do not capture near indirect calls |
| NEAR\_RET | 5 |  | When set, do not capture near returns |
| NEAR\_IND\_JMP | 6 |  | When set, do not capture near indirect jumps |
| NEAR\_REL\_JMP | 7 |  | When set, do not capture near relative jumps |
| FAR\_BRANCH | 8 |  | When set, do not capture far branches |
| Reserved | 63:9 |  |  |

1. 启用Intel支持的”freeze\_on\_pmi”特性，让触发PMI事件过后的分支事件不再写入LBR记录当中。

借助LBR中的分支地址及目标地址信息，我们可以在PMI事件处理函数中进行两个方向的动态反汇编进而进行疑似gadget和critical gadget的判定：

1. 以当前分支地址为起点，可以判定当前分支信息，如果为ret指令，那么可以向目标地址低地址方向进行反汇编，判定目标地址上一条指令是否为call指令，如果不是，那么当前分支片段为疑似gadget；
2. 以上一触发事件分支的目标地址为起点，向高地址方向进行反汇编，如果在不包含无条件跳转（包含中断）的情况经过较短（未超过gadget指令序列长度阈值）的指令片段到达当前记录分支地址，那么当前分支片段为疑似gadget；
3. 如果当前指令为系统调用指令，如果其参数和上一个疑似gadget的参数是相同的，则认为当前片段为critical gadget。

为了支持长无效片段的检测，我们需要进行如下支持：

1. 记录上一个疑似gadget末尾系统的状态；
2. 每次触发检查时，与当前分支的状态进行比较，如果未超过阈值，那么将其标为长无效片段；

为了进行疑似critical gadget判定，我们针对部分敏感系统调用进行了修改，在这些系统调用处理函数开始处插入对应的gadget检查机制，通过LBR记录的中断指令信息，采用类似的反汇编技术判定当前系统调用片段是否为疑似critical gadget。

结合这些gadget判定以及相应的ROP攻击判定策略，我们的原型系统会在发生疑似ROP攻击时及时终止受攻击的用户程序。

* 1. 阈值选择

PerfCFI在检测过程中需要设置两个阈值：gadget指令片段最长长度(maxGadgetLength)以及gadget链最短长度(minChainLength)。

maxGadgetLength阈值的选择需要满足以下条件：

1. 该阈值需要大于绝大部分理想的（状态可控、副作用较小的）gadget长度;
2. 该阈值需要尽量小，以减少误报（避免将正常指令片段识别成gadget）。

综合考虑，本文选取30作为该阈值，经验证，该阈值大于ROPgadget工具在常用软件及库中查找出的绝大多数常用的gadget长度(当前测试中发现最长的状态可控、副作用较小的gadget为27，指令片段如图3所示)。

409f17: 66 0f 1f 84 00 00 00 nopw 0x0(%rax,%rax,1)

409f1e: 00 00

409f20: 48 83 c4 28 add $0x28,%rsp

409f24: 48 89 d8 mov %rbx,%rax

409f27: 5b pop %rbx

409f28: 5d pop %rbp

409f29: 41 5c pop %r12

409f2b: 41 5d pop %r13

409f2d: 41 5e pop %r14

409f2f: 41 5f pop %r15

409f31: c3 retq

Fig.3 The longgest ideal gadget found in the test

图3 测试中发现的最长的“理想”gadget

maxChainLength阈值的选择需要满足以下条件：

1. 该阈值需要大于正常指令执行流中出现的最长疑似gadget链;
2. 该阈值需要尽量小，以减少漏报（避免将ROP攻击链识别成了正常指令流）。

本文针对Spec2006程序以及一些常用的Linux程序进行了采样，识别的最大gadget链长度如下表所示：

**Table 3** Max Length of gadget chain on partial spec2006 and some linux application

表**3** 部分spec2006程序以及一些常用的linux程序中的最大gadget链长度

|  |  |
| --- | --- |
| 程序 | 最大gadget链长度 |
| spec2006(部分) | 3 |
| ls | 3 |
| dmesg | 1 |
| ps | 3 |
| cat | 1 |
| mkdir | 3 |
| ping | 2 |
| stat | 3 |
| passwd | 2 |
| gdb | 4 |
| tar | 2(错误输入强制退出时包含了5个) |
| vi | 4 |

由上表可知，正常运行的程序包含的最大gadget链一般不超过5个。已有研究中对于最大gadget链的长度限定一般在5-20[1,10]。本文采用较为保守的方案，将该阈值设置成10。