PaX/Grsecurity RAP 及其优化

zet

zet (feqin1023@gmail.com), HardenedLinux.org, GuangZhou, China

简介

Return-Oriented Programming (ROP) 是一种比较高级攻击的方式，能够利用现有代码通过找到的 gadgets 串来执行攻击者想进行的任何操作，现有的防御手段只有 Control-flow integrity（CFI）是针对这种攻击进行防御的。此外常规针对存储破坏的防御 canary, NX, ASLR 等，对于这种攻击无效。

PaX/Grsecurity RAP 是一种 CFI 的实现方式，本文描述了 RAP 的实现，以及针对 RAP 的优化，以及重新实现的 hl-cfi 的实现。

1 导引

Return-Oriented Programming (ROP)(Krahmer 2005) 是一种比较高级攻击的方式，是 code-reuse attack 的一种，能够利用现有代码通过找到的 gadgets 串来执行攻击者想进行的任何操作，已经有研究者表明 gadgets 是图灵完备的 (Shacham 2007)。现有的防御手段只有 Control-flow integrity(CFI) 是针对这种攻击进行防御的。此外常规针对存储破坏的防御 canary, NX, ASLR 等，对于这种攻击无效。

自从 CFI(M. Abadi and Ligatti 2005) 提出以来，各种实现层出不群，包括 gcc upstream 里的 vtv 以及 LLVM cfi。不过这些实现有明显的硬伤，vtv 防御 virtual table，LLVM cfi 只是一个 forward cfi 实现，没有 backward 部分。比较而言，PaX/Grsecurity RAP(PaX 2018) 对于 kernel 级别的防护是最理想的选择，不经包括 forward 而且包括 backward 实现，backward cfi 的实现应该是业界第一个也是为数不多的可以进入生产环境的。

贡献

本文主要是贡献是从编译器角度分析 PaX RAP 以及 HardenedLinux hl-cfi(zet 2018) 的改进以及实现。

内容

本文的内容安排包括：2 背景，3 算法与实现，4 结论以及未来工作。

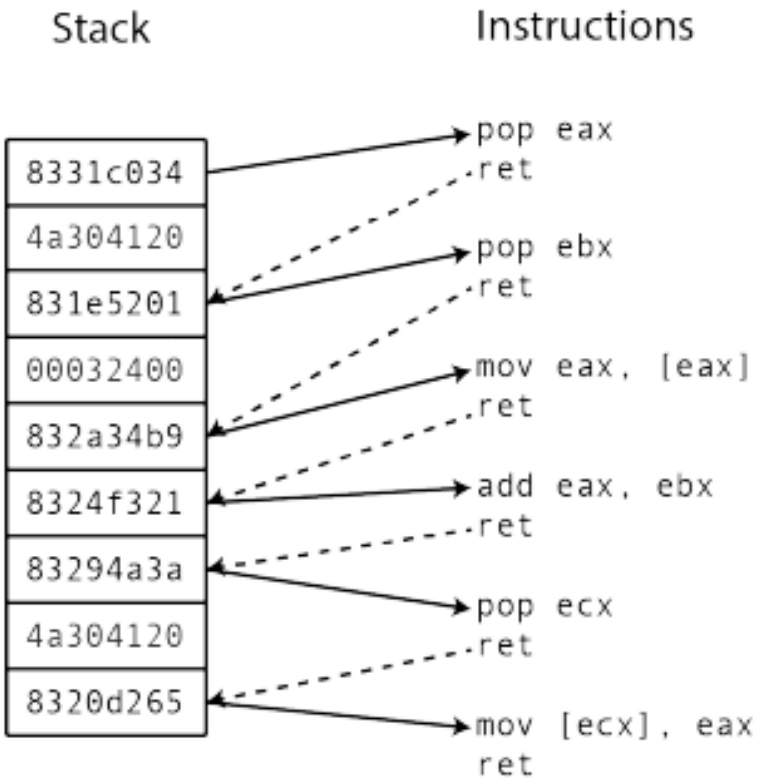
2 背景

Return-Oriented Programming (ROP)(Krahmer 2005) 是 return-into-libc(J. Pincus 2004) 的一种一般形式，也就是最终的目标不是进入 libc。ROP 的攻击由串接执行一连串的 gadgets 进行，gadgets 是一些几条汇编语言的代码片段，一般来说以 **ret** 指令结尾，因为 ROP gadgets 是图灵完备的 (Shacham 2007)，理论上来说,ROP 可以按照攻击者的目的来做任何计算。

下图简要描述了一个常规的 ROP 攻击图示，图片来源于 (N. Carlini 2014)。左边的 Stack 是用户 stack 初始化为 gadgets 的地址，右边的 Instructions 以 ret 结尾就是 gadgets 由左边 stack 处的地址实线指向。这个攻击的结果是将数 0x32400 加到地址 0x4a304120 所存储的值。

由于这个攻击可以看到 ROP 发生的根本原因：就是 **call/jmp/ret** 违反了原始代码控制流。基于这个因素所以有研究者提出了 Control-flow integrity (CFI)(M. Abadi and Ligatti 2005)。原理就是在编译器编译最开始的未经过破坏的源代码的时候就分析控制流，在 **call/jmp/ret** 发生的地方插入检测代码，在代码运行的时候来判断报错是否有攻击行为发生。

关于 ROP 更详细的描述可以参考 (Krahmer 2005)(Shacham 2007)(N. Carlini 2014)。


Stack
Instructions
8331c034
4a304120
831e5201
00032400
832a34b9
8324f321
83294a3a
4a304120
8320d265
pop eax
ret
pop ebx
ret
mov eax, [eax]
ret
add eax, ebx
ret
pop ecx
ret
mov [ecx], eax
ret

ROP 图示

3 算法与实现

根据公开的文献记载，PaX RAP 针对 ROP 的威胁建模和初始设计 (PaX 2003) 在 2003 年时已经存在。下面将描述 PaX RAP 的算法及其实现，分析这种方式带来的性能损耗，和 HardenedLinux 社区所做的优化改进。

由上一节背景的描述可知，ROP 发生的根源就是因为违反了原始代码的控制流。所以对于相应的检测防御也很简单：在原始控制流转移的地方由编译器插入检测代码。这其中就会有一个 trade-off 的考虑: 怎么样在保证一定精度的情况下作出防御?

对于控制流的转移可以分为：

1) 直接调用的 call/jmp
2) 间接调用的 call/jmp - RAP forward cfi
3) 返回 - RAP backward cfi

对于情况 1 来说不需要考虑，因为直接调用都是位于 *.text* 段，只读的 section 不会发生控制流 ROP 的攻击。所以 ROP 的防御就是针对情况 2 和 3。

理论上来说最理想的防御当然是: 对于每个会被间接调用的函数在进入该函数的时候作出检测，对于每个 **ret** 作出检测。对于情况 3 的 **ret** 来说可以确定有 **ret** 就意味着返回，也就是我们必须处理的检测点，也就是 backward cfi 的实现点，也就是对于所有 **ret** 都需要处理。但是对于情况 2 来说复杂了许多，所有优化以及考虑都在于情况 2 里面。对于会被间接调用的函数这个问题是一个 NP 问题，是一个典型的 pointer analysis 的问题，也就是求解一个函数指针的指向范围是什么，相对地也就是求一个函数是不是会被间接调用也就是有没有被函数指针所指向，这个问题在编译器领域大概伴随着优化编译器的最开始研究（大约是 1970 年）一直进行到现在。

PaX RAP 是使用 gcc plugin(GCC-internals 2017) 来实现的，gcc plugin 是 gcc 提供给第三方程序的一个钩子，可以回调第三方的代码，可以在 gcc 内部满足条件的时候输出信息，可以在 gcc 某个 pass 之后输出信息也可以调用第三方的代码作为某个 pass，只要根据规范提供给 gcc 想插入的 pass 点就行了。其他的详细功能请参考 gcc-internals(GCC-internals 2017) 或者 gcc 代码。

RAP forward cfi

RAP 的实现是做为几个 gcc pass 通过 gcc plugin 插入 gcc 来进行的。由上一段的分析知道求解精确的函数是否被间接调用以及函数指针的指向范围是一个 NP 问题，所以学术界和工程界有数不清的学术 paper 以及实现出现，在 gcc 内部的实现来说有一个 flow-insensitive analysis 和一个比较精确的 flow-sensitive 的算法，insensitive 的算法非常简单，就是根据编译器的分析以及函数的声明来判断函数有没有可能被间接调用，代码如下：比如判断是不是 static 有没有取过函数的地址。

flow-insensitive analysis :

```
static inline bool
may_be_aliased (const_tree var) {
  return (TREE_CODE (var) != CONST_DECL
    && !((TREE_STATIC (var) || TREE_PUBLIC (var) || DECL_EXTERNAL (var))
    && TREE_READONLY (var)
    && !TYPE_NEEDS_CONSTRUCTING (TREE_TYPE (var)))
    && (TREE_PUBLIC (var) || DECL_EXTERNAL (var) || TREE_ADDRESSABLE (var)));
}
```

flow-sensitive analysis 算法无数，gcc 的实现基本上是描述于 paper(B. Hardekopf 2009) 里的算法，在 gcc summit 也有开发者的 paper(Berlin 2005) 描述。是基于 Constraint set 的一类算法，只不过是在 gcc 中间代码 tree-ssa 的基础上，针对 ssa 做了改进，这样基于 ssa 的一个稀疏算法。详细的实现参考 (B. Hardekopf 2009)(Berlin 2005)。这是一个流敏感的算法，也就是对于 block 内部的语句有分析，也会进入函数的内部。总是是一个比较精确的算法。

介绍完，RAP 遇到的难点，接着介绍 RAP 对间接函数的处理，RAP 做了一个更粗略的 trade-off，当函数被间接调用的时候当前的调用点，除了函数指针没有任何信息，这里 dereference 函数指针会得到函数类型，函数类型和函数指针的指向是静态分析唯一能够知道的信息，RAP 放弃了函数指针的指向，只抽取了函数的类型，对得到的函数类型做 hash 运算，然后得到一个 hash 值，这个 hash 值就是 RAP 间接检测的标准。hash 值

的计算算法是 SipHash-2-4(JP. Aumasson 2012)，函数类型则是按照标准规范包括返回值和参数 (ISO/IEC 2011)。接着在间接调用的地方插入检测代码。RAP 的检测代码插入在 gcc tree-ssa(Novillo 2003)(Novillo 2004) 的表示层, 在 gcc 内部表示来说 tree-ssa 之前是 tree-ast(Merrill 2003), 而之后是 rtl(GCC-internals 2017)。tree-ssa 是几乎整个 gcc 重要优化 pass 的实现点。

PaX RAP forward cfi alogrithms:

```
for all gimple code of all function
  hashValue = computeHash(currentFunctionType);
  insertHashValueBeforeCurrentFunction(hashValue);
  if (isFunctionPointer(currentPointer))
    functionType = *currentPointer;
    hashValue = computeHash(functionType);
    insertHashValueCheckBeforeCurrentIndirectCall(currentPointer, hashValue);

insertHashValueBeforeCurrentFunction(hashValue)
  fprintf(asm_out_file, "\t.quad %#llx", hashValue);

insertHashValueCheckBeforeCurrentIndirectCall(currentPointer, hashValue);
  value = (long)*((long *)currentPointer - 8);
    if (value == hashValue)
      currentPointer();
    else
      catchAttack();
```

上面给出的是 RAP forward cfi 实现的算法伪代码，在 RAP 的实现上有三个问题:

1) 因为 RAP 没有使用 pointer analysis，所以会对全部函数都输出 hash 值。
2) 上面的伪代码在 RAP 的实现上是在 tree-ssa 表示层，所以其实是几条 gimple 等效的伪代码，但是 RAP 的实现上这些 gimple 代码是作为一个整体插入 gcc 的，而且插入的位置是 gcc 的所有 tree-ssa 优化 pass 之后，也就是 gcc 根本不会感知到这些代码的存在，也就是说 gcc 不会去分析这些代码，也不会去优化这些代码。

3) 安全上的考虑，RAP 的实现是针对函数类型编码 hash，所有函数类型都是同一个 hash 值，所以这是一个可能的漏洞利用平面。

基于以上的三个问题所以我们给出了 Hardenedlinux hl-cfi 的实现 (hl 的意思 high level)。

hl-cfi alogrithms:

```
for all gimple code of all function
  hashValue = computeHash(currentFunctionType);
  if (isCurrentFunctionMaybeAttacked(currentFunction))
    insertHashValueBeforeCurrentFunction(hashValue);
  if (isFunctionPointer(currentPointer))
    functionType = *currentPointer;
      hashValue = computeHash(functionType);
      insertHashValueCheckBeforeCurrentIndirectCall(currentPointer, hashValue);
```

解决 RAP 实现上的问题 1

```
isCurrentFunctionMaybeAttacked(currentFunction)
  lookupPointerAnalysis(currentFunction)
```

解决 RAP 实现上的问题 2

```
insertHashValueCheckBeforeCurrentIndirectCall(currentPointer, hashValue)

  +-------
  stmt1;
  call fptr;
  +-------

  change to =>

  // insertHashValueCheckBeforeCurrentIndirectCall(currentPointer, hashValue)
  +-------
  stmt1;
  lhs_1 = t_;
```

```
ne_expr (lhs_1, s_);  // hsahValueCheck()
// FALLTHRU
<bb ???> # true
cfi_catch();
<bb ???> # false
call fptr;
+--------
```

insertHashValueCheckBeforeCurrentIndirectCall(currentPointer, hashValue) 的插入实现为 block 级别的 gimple code，而且 hl-cfi 的插入点是在 tree-ssa pass 的最开始处，在 pass_build_ssa_passes 之前，所以这里需要实现上的处理，需要模拟 gcc 已经跑过的所有代码对于一个假想的检测 insertHashValueCheckBeforeCurrentIndirectCall() 函数的表示在当前 pass 点是个什么样子，然后插入对应的代码。而且对于 hl-cfi 来说我们需要使用 gcc pointer analysis，所以对于 isCurrentFunctionMaybeAttacked() 的调用又得保证在 pass_ipa_pta(gcc flow-sensitive analysis) 之后。cfi 的检测代码的优化是一个复杂的问题，因为这是一个运行时才能知道的函数指针值，常规静态分析优化 (Gupta 1993) 基本上没有用。所以 hl-cfi 的实现相对来说考虑的只能是尽可能利用 gcc 现有的优化，检测代码的 hash 常数可以被 gcc 做 register promotion(Makarov 2007)，以及分析信息传播到整个 fucntion 里面 (Novillo 2005) 进一步给其他 pass 提供机会。在 hl-cfi 构建 block 以及 edge 的时候给 gcc 提供 profile 信息 (Hubicka 2005)(V. Ramasamy and Hundt 2008)，使用 profile 的相关基础代码。给后续的优化 pass 比如 code placement 以及 cache 优化的 pass 提供信息。

解决 RAP 实现问题 3

// 这部分代码因为复杂度的原因，并没有在hl-cfi里面实现，下面只是提供算法。

```
for all gimple code of all function
  if (isCurrentFunctionMaybeAttacked(currentFunction))
    insertHashValueBeforeCurrentFunction(hashValue);
  if (isFunctionPointer(currentPointer))
    if (pointerAliasSetChanged(currentPointer, currentCodePlace))
      functionType = *currentPointer;
```

```
        // 根据当前的代码分析结果来给一个变化的seed来得到一个
        // 跟当前函数指针和函数类型相关的hash值。
        hashValue = computeHash(functionType, seed);
        insertCurrentHashValue(currentPointer, functionType, currentCodePlace);
    else
        hashValue = lookupHashValue(currentPointer, fucntionType, currentCodePlace);
        insertHashValueCheckBeforeCurrentIndirectCall(currentPointer, hashValue);
```

为了解决问题 3，计算一个动态的 hash 值，根据函数指针的变化而变化。

RAP backward cfi

为了解决 **ret** 的问题，PaX Team 提出了 backward cfi 的软件实现方案。

```
for all rtl code of all function
  if (isCurrentRTLIsRet(currentRTL))
    // 调用gcc内置函数，得到返回地址。
    retAddress = __builtin_return_address();
    // 在gcc里，这个函数类型通过分析可以知道的。
    hashValue = getHash(functionType);
    insertRetCheck(retAddress, hashValue);
```

对于这个检测函数的描述跟 RAP 实现略有差异，RAP 比较的是一个 hashValue 的取反的值。

```
insertRetCheck(retAddress, hashValue)
  if (long*((long *)retAddress - 8) != hashValue)
    catchAttack();
```

对于 backward cfi 来说这大概是业界仅有的几个的实现，而且跟 RAP forward cfi 的一样，也是一串没有经过编译器优化的代码直接出现在输出的 object 里，不过想不到优化的方案，只能期待硬件更新支持 backward cfi，比如类似 ARMv8.3 的 PA(zet 2017)，ARMv8.5 的 MTE(WikiChip 2018) 以及未来的 Intel CET(Intel 2019) 等。

4 结论

对于 hl-cfi 的实现和 RAP 的实现在 spec CPU2017 上做了一个对比测试，因为 RAP 本身是为了 linux kernel 防御，在优化 RAP 以及开发 hl-cfi 的时候，为了方便测试以及开发，我们抽取出了 RAP 作为一个独立的项目，hl-cfi 也是如此，正因为这是两个针对 kernel 的项目，因为应用代码外部 library 依赖的问题，CPU2017 的很多测试跑不过，跑过的数据如下：

RAP	hl-cfi	hl-cfi - RAP / RAP
2.15	2.35	9.3%

大约有 9.3% 的性能提升。

由于 hl-cfi 的实现依赖于 pointer analysis，如果使用 lto(T. Glek 2010) 来编译 linux kernel，相信 hl-cfi 应该会有一个更佳的表现。lto 模式更好的一个特点是只需要修改 linux kernel 的编译构建脚本，对于 hl-cfi 本身不需要做改变。期待未来 lto 的编译能够普及。

引用

Berlin, D. 2005. "Structure aliasing in GCC." *In Proceedings of the 2005 GCC Summit*, i25–i36.

B. Hardekopf, C. Lin. 2009. "Semi-sparse flow-sensitive pointer analysis." *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, i226–i238.

GCC-internals. 2017. "Chapter rtl and plugins." *GNU Compiler Collection (GCC) Internals.*

Gupta, R. 1993. "Optimizing array bound checks using flow analysis." *ACM Letters on Programming Languages and Systems*, i135–i150.

Hubicka, J. 2005. "Profile driven optimisations in GCC." *In Proceedings of the 2005 GCC Summit*, i107–i124.

Intel. 2019. “未来的 Intel CET.” *Https://Software.intel.com/Sites/Default/Files/Managed/4d/2a/Contro Flow-Enforcement-Technology-Preview.pdf*.

ISO/IEC. 2011. “Information technology — Programming languages — C.” *ISO/IEC 9899:2011*, i42.

JP. Aumasson, D. J. Bernstein. 2012. “SipHash: A Fast Short-Input PRF.” *Progress in Cryptology - INDOCRYPT*, i489–i508.

J. Pincus, B. Baker. 2004. “Beyond stack smashing: Recent advances in exploiting buffer overruns.” *Security & Privacy, IEEE.*, i20–i27.

Krahmer, S. 2005. “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique.”

M. Abadi, Ú. Erlingsson, M. Budiu, and J. Ligatti. 2005. “”Control-Flow Integrity: Principles, Implementations, and Applications”.” *Proceedings of the 12th ACM Conference on Computer and Communications Security*, i340–i353.

Makarov, V N. 2007. “The integrated register allocator for GCC.” *In Proceedings of the 2007 GCC Summit*, i77–i90.

Merrill, J. 2003. “GENERIC and GIMPLE: A New Tree Representation for Entire Functions.” *In Proceedings of the 2003 GCC Summit*, i171–i180.

N. Carlini, D. Wagner. 2014. “ROP is still dangerous: Breaking modern defenses.” *In USENIX Security Symposium*, i385–i399.

Novillo, D. 2003. “Tree SSA – A New Optimization Infrastructure for GCC.” *In Proceedings of the 2003 GCC Summit*, i181–i194.

———. 2004. “Design and Implementation of Tree SSA.” *In Proceedings of the 2004 GCC Summit*, i119–i130.

———. 2005. “A Propagation Engine for GCC.” *In Proceedings of the 2005 GCC Summit*, i175–i184.

PaX. 2003. “PaX RAP 针对 ROP 的威胁建模和初始设计.” *Https://Pax.grsecurity.net/Docs/Pax-*

Future.txt.

———. 2018. “Frequently Asked Questions About RAP.”

Shacham, H. 2007. “structural variant discovery by integrated paired-end and split-read analysis.” *Proceedings of the 14th ACM Conference on Computer and Communications Security*.

T. Glek, J. Hubicka. 2010. “Optimizing real-world applications with GCC Link Time Optimization.” *In Proceedings of the 2010 GCC Summit*.

V. Ramasamy, D. Chen, P. Yuan, and R. Hundt. 2008. “Feedback-Directed Optimizations in GCC with Estimated Edge Profiles from Hardware Event Sampling.” *In Proceedings of the 2008 GCC Summit*, i87–i101.

WikiChip. 2018. “ARMv8.5 的 MTE.” *Https://En.wikichip.org/Wiki/Arm/Mte*.

zet. 2017. “ARMv8.3-A PA 在 GCC 里的相关实现.”

———. 2018. “hl-cfi project.” *Https://Github.com/Hardenedlinux/RAP-Optimizations*.