Hooking linux内核函数（三）：Ftrace的主要优缺点

Edvison / 2018-10-21 21:57:13 / 浏览数 1724

本文是《Hooking Linux Kernel Functions, Part 3: What Are the Main Pros and Cons of Ftrace?》的翻译文章。

前言

Ftrace是一个Linux实例程序，通常用于跟踪内核函数。 但是，当我们寻找一个有用的解决方案，允许我们启用系统活动监控和阻止可疑进程时，发现Linux ftrace也可用于钩子函数调用。

这是本系列文章的最后一部分，本系列共分三部分，主要讨论如何使用ftrace来hook Linux内核函数。在本文中，我们将重点讨论ftrace的主要优缺点，并描述在用ftrace hooking Linux内核函数时所遇到的一些意外情况。阅读本系列文章的第一部分，了解其他四种可用于hooking Linux内核函数调用的方法。你在想:什么是ftrace?ftrace是如何工作的?那么请参阅本系列的第二部分，以获得这些问题的答案，并了解关于如何使用ftrace来hooking Linux内核函数。

使用ftrace的利弊

Ftrace使Linux内核函数更容易hook，并具有几个关键优势：

- 一个成熟的API和简单的代码。在内核中利用现成的接口大大降低了代码的复杂性。只需要进行两个函数调用，填充两个结构字段，并在回调中添加一些magic，就可以用
- 能够根据名称跟踪任何函数。使用ftrace跟踪Linux内核是一个相当简单的过程——用常规字符串编写函数名就足够指向你需要的函数名了。不需要纠结于链接器、扫描内

但就像我们在本系列中描述的其他方法一样，ftrace有一些缺点。

- 内核配置要求。 确保成功进行Linux内核跟踪需要几个内核要求：
  -- 用于按名称搜索功能的kallsyms符号列表
  -- 用于执行跟踪的整个ftrace框架
  -- Ftrace选项对钩子函数来说至关重要

所有这些功能都可以在内核配置中禁用，因为它们对系统的运行并不重要。
但是，通常流行发行版使用的内核仍然包含所有这些内核选项，因为它们不会显着影响系统性能，并且可能对调试很有用。
但是，如果你需要支持某些特定的内核，最好还是记住这些要求。

- 开销成本。因为ftrace不使用断点，所以它的开销比kprobes低。但是，这种方法的开销比手工拼接要高。实际上，动态ftrace是拼接的变体，它执行了不必要的ftrace代
- 函数被包装成一个整体。与通常的拼接一样，ftrace将函数包装为一个整体。虽然从技术上讲，拼接可以在函数的任何部分执行，但ftrace只在入口点工作。你可以将这种
- 双重调用ftrace。正如我们之前解释过的，使用parent_ip指针进行分析会导致对同一个钩子函数调用两次ftrace。这增加了一些间接成本，并可能干扰其他跟踪的读取，

让我们仔细分析这些缺点。

内核配置要求

内核必须同时支持ftrace和kallsyms。这需要启用两个配置选项:

- CONFIG_FTRACE
- CONFIG_KALLSYMS

接下来，ftrace必须支持动态寄存器修改，开启以下选项:

- CONFIG_DYNAMIC_FTRACE_WITH_REGS

要访问FTRACE_OPS_FL_IPMODIFY标志，你使用的内核必须基于版本3.19或更高版本。旧的内核版本仍然可以修改%rip寄存器，但是在版本3.19中，只有在设置标志之后才 hook。

最后但并非最不重要的是，我们需要注意函数内部的ftrace调用位置。 ftrace调用必须位于函数的开头，在函数序言之前（形成堆栈帧并分配局部变量的空间）。
以下选项考虑了此功能：

- CONFIG_HAVE_FENTRY

虽然x86_64架构支持这个选项，但i386架构不支持。由于i386架构的ABI限制，编译器不能在函数序言之前插入ftrace调用。因此，当你执行ftrace调用时，函数堆栈已经被

这就是为什么ftrace函数hooking不支持32位x86体系结构。从理论上讲，你仍然可以通过生成和执行反序言来实现此方法，但是它将显著提高技术复杂性。

使用ftrace时的意外情况

在测试阶段，我们面临一个特殊的特性:在某些发行版上hook函数会导致系统永久挂起。当然，这个问题只发生在与开发人员使用的系统不同的系统上。我们也无法在任何发

经调试，系统断在了钩子函数里。由于一些未知的原因，当在ftrace回调中调用原函数时，parent_ip仍然指向内核而不是函数包装器。这就启动了一个死循环，ftrace一遍又

幸运的是，我们有错误的和有效的代码，最终发现了问题的原因。我们统一了代码并去掉了我们现在不需要的部分，并使包装器函数代码的两个版本之间的差异缩小了。

这是稳定的代码：

```
static asmlinkage long fh_sys_execve(const char __user *filename,
                const char __user *const __user *argv,
                const char __user *const __user *envp)
{
        long ret;

        pr_debug("execve() called: filename=%p argv=%p envp=%p\n",
                filename, argv, envp);

        ret = real_sys_execve(filename, argv, envp);

        pr_debug("execve() returns: %ld\n", ret);

        return ret;
}
```

这是导致系统挂起的代码:

```
static asmlinkage long fh_sys_execve(const char __user *filename,
                const char __user *const __user *argv,
                const char __user *const __user *envp)
{
        long ret;

        pr_devel("execve() called: filename=%p argv=%p envp=%p\n",
                filename, argv, envp);

        ret = real_sys_execve(filename, argv, envp);

        pr_devel("execve() returns: %ld\n", ret);

        return ret;
}
```

日志级别如何影响系统行为？ 令人惊讶的是，当我们仔细研究这两个函数的机器代码时，我们发现这些问题背后的原因是编译器。

结果是，pr_devel()调用被扩展为no-op。这个printk-macro版本用于开发阶段的日志记录。由于这些日志在操作阶段没有任何意义，系统会自动将它们从代码中删除，除非

```
static asmlinkage long fh_sys_execve(const char __user *filename,
                const char __user *const __user *argv,
                const char __user *const __user *envp)
{
        return real_sys_execve(filename, argv, envp);
}
```

这就是优化的阶段。在我们的示例中，激活了所谓的尾部调用优化。如果一个函数调用另一个函数并立即返回它的值，这种优化让编译器可以用更直接的跳转到函数的主体来

```
0000000000000000 <fh_sys_execve>:
  0: e8 00 00 00 00 callq 5 <fh_sys_execve+0x5>
  5: ff 15 00 00 00 00 callq *0x0(%rip)
  b: f3 c3 repz retq </fh_sys_execve>
```

这是一个失败调用的例子：

```
0000000000000000 <fh_sys_execve>:
  0: e8 00 00 00 00 callq 5 <fh_sys_execve+0x5>
  5: 48 8b 05 00 00 00 00 mov 0x0(%rip),%rax
  c: ff e0 jmpq *%rax </fh_sys_execve>
```

第一个调用指令与编译器在所有函数的开头插入的fentry()调用完全相同。但在那之后，坏代码和稳定代码的行为就不同了。在稳定的代码中，我们可以看到调用指令执行的

尾部调用优化允许你通过不分配包含call指令存储在堆栈中的返回地址的无用堆栈帧来节省一些时间。但是，由于我们使用parent_ip来决定是否需要hook，因此返回地址的

这也是问题仅出现在某些发行版上的主要原因。 不同的发行版使用不同的编译标志集来编译模块。 在所有问题分布中，尾调用优化默认是开启的。

我们通过使用包装器函数关闭整个文件的尾部调用优化来解决这个问题:

```
# pragma GCC■■(“-fno-optimize-sibling-calls”)
```

至于进一步的hooking实验，你可以使用GitHub的完整内核模块代码。

结论

虽然开发人员通常使用ftrace来跟踪Linux内核函数调用，但这个实例程序本身对于Hooking
Linux内核函数也非常有用。尽管这种方法有一些缺点，但它给了你一个关键的好处:代码和hook过程的整体简单性。

上一篇：Hooking linux内核函数... 下一篇：如何打造我们自己的Android ...

1. 0 条回复
   - 动动手指，沙发就是你的了！