Hooking linux内核函数（二）：如何使用Ftrace hook函数

Edvison / 2018-10-21 21:56:09 / 浏览数 2739 技术文章 技术文章 顶(0) 踩(0)

本文是《Hooking Linux Kernel Functions, Part 2: How to Hook Functions with Ftrace》的翻译文章

前言

Ftrace是一个用于跟踪Linux内核函数的Linux内核框架。
但是，当我们尝试启用系统活动监控以阻止可疑进程时，我们的团队设法找到了一种使用ftrace的新方法。
事实证明，ftrace允许你从可加载的GPL模块安装钩子而无需重建内核。 此方法适用于x86_64体系结构的Linux内核版本3.19和更高版本。

这是我们关于Hooking Linux内核函数调用的三部分系列的第二部分。 在本文中，我们将解释如何使用ftrace来hook Linux内核中的关键函数调用。
你可以阅读本系列的第一部分，以了解有关可用于完成此任务的其他方法的更多信息。

一种新方法：使用ftrace进行Linux内核hooking

什么是ftrace？ 基本上，ftrace是一个用于在函数级别跟踪内核的框架。 该框架自2008年以来一直在开发中，具有相当令人印象深刻的函数集。
使用ftrace跟踪内核函数时，通常可以获得哪些数据？ Linux ftrace显示调用图，跟踪函数调用的频率和长度，按模板过滤特定函数等。
在本文的下面，可以找到对官方文档和资源的引用，你可以使用它们来了解有关ftrace函数的更多信息。

ftrace的实现基于编译器选项-pg和-mfentry。 这些内核选项在每个函数的开头插入一个特殊跟踪函数的调用 —— mcount()或fentry ()。
在用户程序中，分析器使用此编译器功能来跟踪所有函数的调用。 但是，在内核中，这些函数用于实现ftrace框架。

当然，从每个函数调用ftrace都是非常昂贵的。这就是为什么有一种针对流行架构的优化——动态ftrace。如果没有使用ftrace，它几乎不会影响系统，因为内核知道调用mc

必要函数说明

下面的结构可以用来描述每个钩子函数:

```
/**
* struct ftrace_hook describes the hooked function
*
* @name: the name of the hooked function
*
* @function: the address of the wrapper function that will be called instead
* of the hooked function
*
* @original: a pointer to the place where the address
* of the hooked function should be stored, filled out during installation
* of the hook
*
* @address: the address of the hooked function, filled out during installation
* of the hook
*
* @ops: ftrace service information, initialized by zeros;
* initialization is finished during installation of the hook
*/
struct ftrace_hook {
       const char *name;
       void *function;
       void *original;

       unsigned long address;
       struct ftrace_ops ops;
};
```

用户只需要填写三个字段:name、function和original。其余字段被认为是实现细节。你可以把所有Hook函数的描述放在一起，并使用宏使代码更紧凑:

```
#define HOOK(_name, _function, _original) \
       { \
           .name = (_name), \
           .function = (_function), \
           .original = (_original), \
       }
```

```
static struct ftrace_hook hooked_functions[] = {
       HOOK("sys_clone", fh_sys_clone, &real_sys_clone),
       HOOK("sys_execve", fh_sys_execve, &real_sys_execve),
};
```

下面是钩子函数包装的结构：

```
/*
* It’s a pointer to the original system call handler execve().
* It can be called from the wrapper. It’s extremely important to keep the function signature
* without any changes: the order, types of arguments, returned value,
* and ABI specifier (pay attention to “asmlinkage”).
*/
static asmlinkage long (*real_sys_execve)(const char __user *filename,
               const char __user *const __user *argv,
               const char __user *const __user *envp);

/*
* This function will be called instead of the hooked one. Its arguments are
* the arguments of the original function. Its return value will be passed on to
* the calling function. This function can execute arbitrary code before, after,
* or instead of the original function.
*/
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;
}
```

现在，钩子函数有最少的额外代码。唯一需要特别注意的是函数签名。它们必须完全相同;否则，参数就会被错误地传递，一切都会出错。不过，对于hooking系统调用来说

初始化ftrace

我们的第一步是查找和保存钩子函数地址。你可能知道，在使用ftrace时，Linux内核跟踪可以通过函数名执行。但是，我们仍然需要知道原始函数的地址才能调用它。

您可以使用kallsyms（所有内核符号的列表）来获取所需函数的地址。 此列表不仅包括为模块导出的符号，实际上还包括所有的符号。
获取钩子函数地址的过程如下所示：

```
static int resolve_hook_address (struct ftrace_hook *hook)

       hook->address = kallsyms_lookup_name(hook->name);

       if (!hook->address) {
               pr_debug("unresolved symbol: %s\n", hook->name);
               return -ENOENT;
       }

       *((unsigned long*) hook->original) = hook->address;

       return 0;
}
```

接下来，我们需要初始化ftrace_ops结构。 这里我们有一个必要的字段func，指向回调。 但是，需要一些关键flags：

```
int fh_install_hook (struct ftrace_hook *hook)

       int err;

       err = resolve_hook_address(hook);
       if (err)
```

```
                return err;

        hook->ops.func = fh_ftrace_thunk;
        hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                        | FTRACE_OPS_FL_IPMODIFY;

        /* ... */
}
```

fh_ftrace_thunk()特性是ftrace在跟踪函数时调用的回调函数。我们稍后将讨论这个回调。hooking需要这些flags——它们命令ftrace保存和恢复处理器寄存器，我们可以在

现在我们准备好开始hook了。首先，我们使用ftrace_set_filter_ip()为所需的函数打开ftrace实用程序。其次，我们使用register_ftrace_function()给ftrace权限来调用我们的

```
int fh_install_hook (struct ftrace_hook *hook)
{
        /* ... */

        err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
        if (err) {
                pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
                return err;
        }

        err = register_ftrace_function(&hook->ops);
        if (err) {
                pr_debug("register_ftrace_function() failed: %d\n", err);

                /* Don’t forget to turn off ftrace in case of an error. */
                ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);

                return err;
        }

        return 0;
}
```

要关闭钩子，我们只需反向重复相同的操作：

```
void fh_remove_hook (struct ftrace_hook *hook)
{
        int err;

        err = unregister_ftrace_function(&hook->ops);
        if (err)
                pr_debug("unregister_ftrace_function() failed: %d\n", err);
        }

        err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
        if (err) {
                pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
        }
}
```

当unregister_ftrace_function()调用结束时，可以保证系统中不会激活已安装的回调函数或包装器。我们可以卸载hook模块，而不用担心我们的函数仍然在系统的某个地方

用ftrace hook函数

那么如何配置内核函数hook呢?这个过程非常简单:ftrace能够在退出回调后更改注册状态。通过改变寄存器%rip——一个指向下一个执行指令的指针——我们可以改变处理

这是ftrace回调的样子:

```
static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long parent_ip,
                struct ftrace_ops *ops, struct pt_regs *regs)
{
        struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);

        regs->ip = (unsigned long) hook->function;
}
```

我们使用宏container_of()和struct ftrace_hook中嵌入的struct ftrace_ops的地址为我们的函数获取struct ftrace_hook的地址。
接下来，我们使用处理程序的地址替换struct pt_regs结构中的寄存器%rip的值。 对于x86_64以外的体系结构，此寄存器可以具有不同的名称（如PC或IP）。
但基本思想仍然适用。

请注意，为回调添加的notrace说明符需要特别注意。 此说明符可用于标记Linux内核跟踪中禁止使用ftrace的函数。
例如，你可以标记跟踪过程中使用的ftrace函数。
通过使用这个说明符，如果不小心从ftrace回调中调用了一个函数，系统就不会挂起，因为ftrace正在跟踪这个函数。

ftrace回调经常使用禁用抢占来调用(就像kprobes一样)，尽管可能有一些例外。但是在我们的例子中，这个限制并不重要，因为我们只需要替换pt_regs结构中%rip值的8个

由于包装函数和原始函数在相同的上下文中执行，因此两个函数具有相同的限制。
例如，如果你hook一个中断处理程序，那么在包装函数中休眠仍然是不可能的。

防止递归调用

在我们之前给出的代码中有一个问题:当包装函数调用原始函数时，原始函数将被ftrace再次跟踪，从而导致无穷无尽的递归。通过使用parent_ip——ftrace回调参数之一，

差异非常显著:在第一次调用期间，参数parent_ip将指向内核中的某个位置，而在重复调用期间，它只指向包装函数内部。你应该只在第一个函数调用期间传递控制。所有其

我们可以通过将地址与当前模块的边界与我们的函数进行比较来运行入口测试。
但是，只有当模块不包含调用钩子函数的包装函数以外的任何内容时，此方法才有效。 否则，你需要更挑剔。

这是一个正确ftrace回调的样子：

```
static void notrace fh_ftrace_thunk (unsigned long ip, unsigned long parent_ip,
               struct ftrace_ops *ops, struct pt_regs *regs)
{
       struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
       /* Skip the function calls from the current module. */
       if (!within_module(parent_ip, THIS_MODULE))
               regs->ip = (unsigned long) hook->function;
}
```

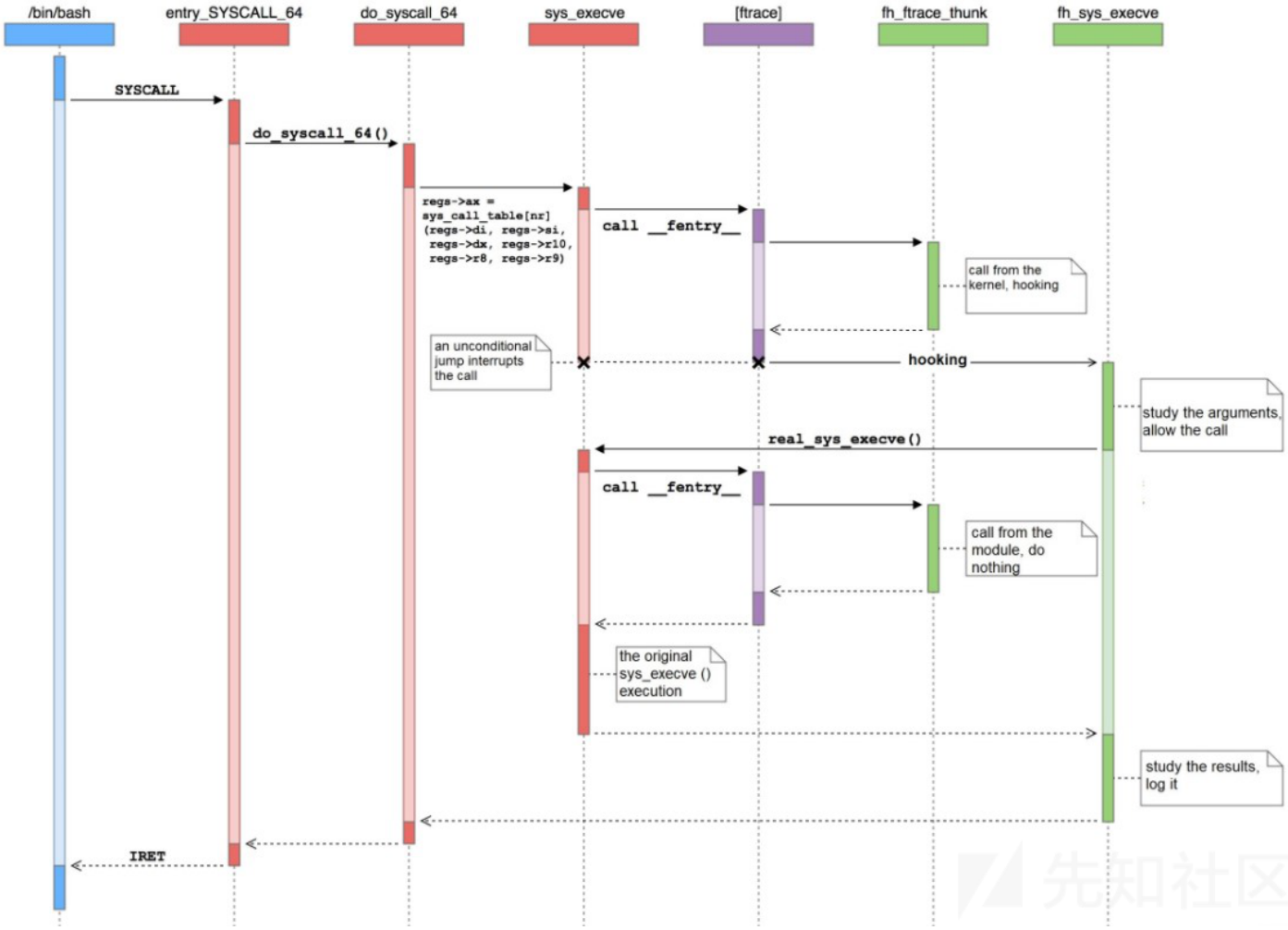
这种方法有三个主要优点:

- 较低的开销。只需要执行几个比较和减法，而不需要获取任何自旋锁或遍历列表。
- 它不必是全局的。由于没有同步，这种方法与抢占是兼容的，并且不绑定到全局进程列表。因此，你甚至可以跟踪中断处理程序。
- 函数没有限制。这种方法没有主要的kretprobes缺点，可以支持开箱即用的任何数量的跟踪函数激活(包括递归)。在递归调用期间，返回地址仍然位于模块外部，因此回
  在下一节中，我们将更详细地了解hook过程，并描述ftrace是如何工作的。

hooking程序的方案

那么，ftrace是如何工作的呢?让我们来看一个简单的示例:你在终端中键入了命令，以查看当前目录中的文件列表。命令行解释器(比如Bash)使用标准C库中的常用函数fork()
execve()系统调用，以获得启动新进程的控制权。

下面的图给出了一个ftrace示例，并说明了hooking处理函数的过程。



在此图中，我们可以看到用户进程（蓝色）如何执行对内核（红色）的系统调用，其中ftrace框架（紫色）从我们的模块（绿色）调用函数。

下面，我们详细描述了这个过程的每一步:

SYSCALL指令由用户进程执行。该指令允许切换到内核模式，并让低级系统调用处理程序entry_SYSCALL_64()负责。此处理程序负责64位内核上64位程序的所有系统调

一个特定的处理器接收控制。内核快速完成汇编程序上实现的所有低级任务，并将控制权移交给高级的do_syscall_64()函数，该函数使用c语言编写。该函数到达系统调用

调用ftrace。在每个内核函数的开头都有一个fentry()函数调用。该函数由ftrace框架实现。在不需要跟踪的函数中，这个调用被替换为nop指令。然而，对于sys_execve

Ftrace调用我们的回调。Ftrace调用所有注册的跟踪回调，包括我们的。其他回调不会干扰，因为在每个特定的位置，只能安装一个回调来更改%rip寄存器的值。

回调函数执行hooking。这个回调函数查看在do_syscall_64()函数内部的parent_ip引导的值——因为它是调用sys_execve()处理程序的特定函数——并决定hook函数，

Ftrace恢复寄存器的状态。在FTRACE_SAVE_REGS标志之后，框架在调用处理程序之前将注册状态保存在pt_regs结构中。当处理结束时，从相同的结构恢复寄存器。我

包装函数接收控制。无条件跳转使它看起来像sys_execve()函数的激活已经终止。不是这个函数，而是fh_sys_execve()函数。同时，处理器和内存的状态保持不变，因此

原函数是由包装函数调用的。现在，系统调用在我们的控制之下。在分析系统调用的上下文和参数之后，fh_sys_execve()函数可以允许或禁止执行。如果禁止执行，函数

回调获得控制权。就像在sys_execve()的第一次调用期间，控件通过ftrace到我们的回调。但这一次，这个过程以不同的方式结束。

回调什么也不做。sys_execve()函数不是由内核从do_syscall_64()调用的，而是由我们的fh_sys_execve()函数调用的。因此，寄存器保持不变，sys_execve()函数照常执

包装函数获得控制权。系统调用处理程序sys_execve()第二次将控制权交给我们的fh_sys_execve()函数。现在，一个新进程的启动已经接近完成。我们可以看到execve()

内核接收控制。最后，运行完fh_sys_execve()函数，并返回do_syscall_64()函数。该函数将调用视为正常完成的调用，而内核照常运行。

控制权转交给用户进程。最后，内核执行IRET指令(或SYSRET，但对于execve()只能执行IRET)，为新用户进程安装寄存器，并将处理器切换到用户代码执行模式。系统调
如你所见，用ftrace hooking Linux内核函数调用的过程并不复杂。

结论

尽管ftrace的主要目的是跟踪Linux内核函数调用，而不是hook它们，但我们的创新方法被证明既简单又有效。但是，我们上面描述的方法只适用于内核版本3.19或更高版本

在本系列的第三部分(也是最后一部分)中，我们将介绍ftrace的主要优点和缺点，以及如果你决定实现这种方法，可能会遇到的一些意外惊喜。与此同时，你还可以了解安装

上一篇：Hooking linux内核函数... 下一篇：Hooking linux内核函数...

1. 0 条回复
   - 动动手指，沙发就是你的了！