「驭龙」Linux执行命令监控驱动实现解析

YSRC / 2018-04-03 12:56:27 / 浏览数 2084 技术文章 技术文章 顶(0) 踩(0)

本文是「驭龙」系列的第三篇文章，对照代码解析了驭龙在Linux执行命令监控驱动这块的实现方式。在正式宣布驭龙项目开源之前，YSRC已经发了一篇关于驭龙EventLog读取模块迭代历程的文章。

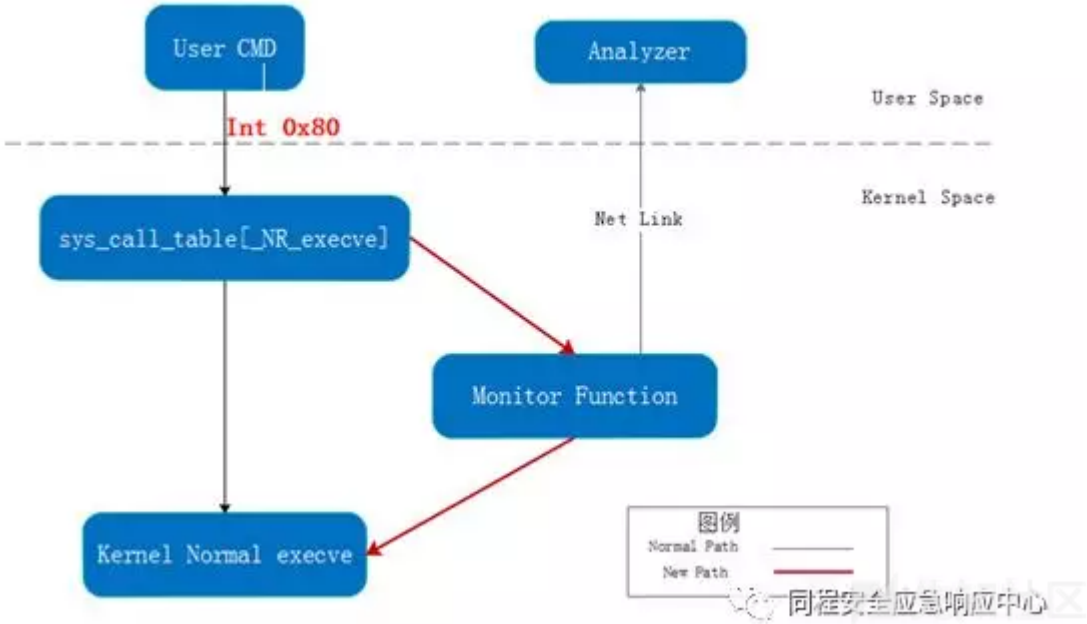
0x00 背景介绍
Linux上的HIDS需要实时对执行的命令进行监控，分析异常或入侵行为，有助于安全事件的发现和预防。为了获取执行命令，大致有如下方法：

1. 遍历/proc目录，无法捕获瞬间结束的进程。
2. Linux kprobes调试技术，并非所有Linux都有此特性，需要编译内核时配置。
3. 修改glic库中的execve函数，但是可通过int0x80绕过glic库，这个之前360 A-TEAM一篇文章有写到过。
4. 修改sys_call_table，通过LKM(loadable kernel module)实时安装和卸载监控模块，但是内核模块需要适配内核版本。

综合上面方案的优缺点，我们选择修改sys_call_table中的execve系统调用，虽然要适配内核版本，但是能100%监控执行的命令。

0x01 总体架构

首先sys_execve监控模块，需要替换原有的execve系统调用。在执行命令时，首先会进入监控函数，将日志通过NetLink发送到用户态分析程序（如想在此处进行命令拦截，



0x02 获取sys_call_table地址
获取sys_call_table的数组地址，可以通过/boot目录下的System.map文件中查找。

命令如下：

```
cat  /boot/System.map-`uname-r` | grep sys_call_table
```

这种方式比较麻烦，在每次insmod内核模块的时候，需要将获取到的地址通过内核模块传参的方式传入。而且System.map并不是每个系统都有的，删除System.map对于系

我们通过假设加偏移的方法获取到sys_call_table地址，首先假设sys_call_tale地址为sys_close，然后判断sys_call_table[__NR_close]是否等于sys_close，如果不等于则将网
*)这么多字节，直到满足之前的判断条件，则说明找到正确的sys_call_table的地址了。

代码如下：

```
unsigned long **find_sys_call_table(void) {

   unsigned long ptr;

   unsigned long *p;

   pr_err("Start foundsys_call_table.\n");

       for (ptr = (unsignedlong)sys_close;

        ptr < (unsignedlong)&loops_per_jiffy;
```

```
        ptr += sizeof(void*)) {
       p = (unsigned long*)ptr;
       if (p[__NR_close] ==(unsigned long)sys_close) {
           pr_err("Foundthe sys_call_table!!! __NR_close[%d] sys_close[%lx]\n"
                   "__NR_execve[%d] sct[__NR_execve][0x%lx]\n",
                  __NR_close,
                  (unsigned long)sys_close,
                  __NR_execve,
                   p[__NR_execve]);
           return (unsignedlong **)p;
       }
   }
       return NULL;
}
```

0x03 修改__NR_execve地址
即使获取到了sys_call_table也无法修改其中的值，因为sys_call_table是一个const类型，在修改时会报错。因此需要将寄存器cr0中的写保护位关掉，wp写保护的对应的bit

代码如下：

```
unsigned long original_cr0;
original_cr0 = read_cr0();
write_cr0(original_cr0 & ~0x00010000);  #■■■■■
orig_stub_execve = (void *)(sys_call_table_ptr[__NR_execve]);
sys_call_table_ptr[__NR_execve]= (void *)monitor_stub_execve_hook;
write_cr0(original_cr0);        #■■■■■
```

在修改sys_call_hook[__NR_execve]中的地址时，不只是保存原始的execve的地址，同时把所有原始的系统调用全部保存下载。

```
void *orig_sys_call_table [NR_syscalls];
for(i = 0; i < NR_syscalls - 1; i ++) {
   orig_sys_call_table[i] =sys_call_table_ptr[i];
}
```

0x04 Execve进行栈平衡
除了execve之外的其他系统调用，基本只要自定义函数例如：my_sys_write函数，在此函数中预先执行我们的逻辑，然后再执行orig_sys_write函数，参数原模原样传入即可
Panic。

需要进行一下栈平衡，操作如下：

1.义替换原始execve函数的函数monitor_stub_execve_hook

```
.text
.global monitor_stub_execve_hook
monitor_stub_execve_hook:
```

2.在执行execve监控函数之前，将原始的寄存器进行入栈操作：

```
pushq   %rbx

pushq   %rdi

pushq   %rsi

pushq   %rdx

pushq   %rcx

pushq   %rax

pushq   %r8

pushq   %r9

pushq   %r10

pushq   %r11
```

3.执行监控函数并Netlink上报操作：

```
call monitor_execve_hook
```

4.入栈的寄存器值进行出栈操作

```
pop     %r11

pop     %r10

pop     %r9

pop     %r8

pop     %rax

pop     %rcx

pop     %rdx

pop     %rsi

pop     %rdi

pushq   %rbx
```

5.执行系统的execve函数

```
jmp    *orig_sys_call_table(, %rax, 8)
```

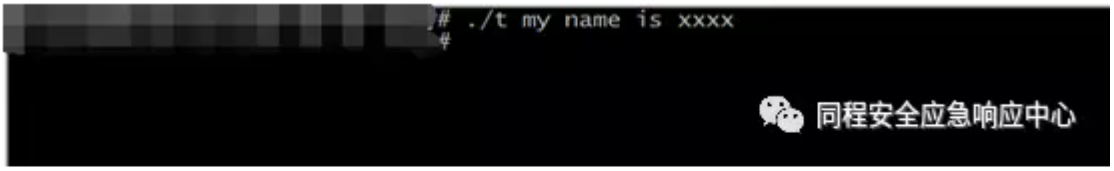
0x05 执行命令信息获取

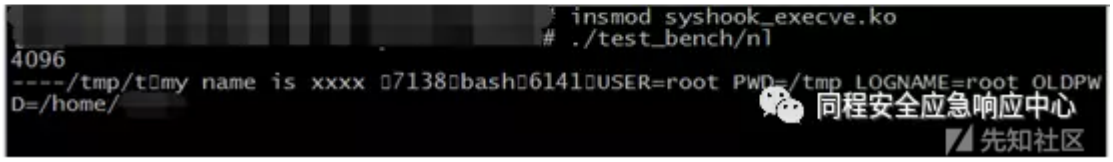
监控执行命令，如果用户态使用的是相对路径执行，此模块也需要获取出全路径。通过getname()函数获取执行文件名，通过open_exec()和d_path()获取出执行文件全路径

最终将获取到的数据组装成字符串，用ascii码值为0x1作为分隔符，通过netlink_broadcast()发送到到用户态分析程序处理。

0x06 监控效果

在加载内核模块，在用户态执行netlink消息接收程序。然后使用相对路径执行命令./t my name is xxxx，然后查看用户态测试程序获取的数据。





0x07 版本支持及代码

支持内核版本：2.6.32, >=3.10.0

源代码路径：https://github.com/ysrc/yulong-hids/tree/master/syscall_hook

关注公众号后回复 驭龙，加入驭龙讨论群。

点击收藏 | 0 关注 | 1

上一篇：0CTF 2018 EZDOOR(... 下一篇：Coding art in she...

1. 1 条回复

worm 2018-04-03 16:59:22

https://www.ibm.com/developerworks/cn/linux/l-connector/

进程事件连接器的使用

进程监控不能直接使用这个特性吗？

Netlink的进程事件和你们这种syscall_hook的区别，优缺点可以分享下吗？

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

目录