B.Tech Project Final Report
Android Security

Akshay Anand (EE16B046)

June 28, 2020

Abstract

In this project, we first took a look at kernel modifications in the Linux kernel, both when booting into a desktop and into a Raspberry PI 3. Custom proc entries and system calls were also added to the kernel. Then, Android was compiled from source and booted in a Raspberry PI 3 with kernel modification after which its performance counters were recorded. Due to unavailability of lab equipment at this point due to institute closure, other methods like kernel emulation and implementing TrustZone were then explored.

1 Problem Statement

The main aim of this project was to figure out a way to boot Android 10 on a Raspberry PI 3 and modify its kernel to add security features to it. To do this, hardware level performance counters had to be read so as to detect attacks, and using this information, security features could be added.
We also looked at TrustZone implementation in a custom Android build.

2 Background and Key Idea

2.1 Definitions

Performance Monitoring Counters or PMCs are special purpose hardware counters (separate registers at the hardware level) that can be accessed via processor registers, and enabled and read via certain instructions. PMCs provide low-level CPU performance statistics that aren't available anywhere else. Since they are directly connected to the CPU at the hardware level,

they can be recorded and read to know the CPU state without using up a lot of CPU cycles and thus not adding a high performance overhead.

As part of performance monitoring, the main metric used to determine performance in IPC or Instructions Per Cycle and is measured by counting the instruction count and cycle count PMCs. The more instructions that can be completed with fixed cycles resource, the better.

2.2 Use of PMCs in security

With the measurement of PMCs, in Intel machines (taken as an example to demonstrate its usefulness in security), we would be able to detect attacks that exploit Meltdown and Spectre, as explained below:

Meltdown and Spectre are attacks that take advantage of the speculative execution and branch predictions features of the processors to fetch higher privilege data without the proper access rights and store it in cache. At this point, the CPU would find out the program does not have enough rights and exit from it, but the data is already in cache. Once, it is present there, a separate program can probe the cache for the malicious data and then read it.

PMCs can be used to detect this attack. In the last step, a separate program probes the cache to find the location in cache with a cache hit. During this process, there is bound to be a lot of cache misses. Since cache misses involve fetching data from RAM, which is slower, we can measure/detect these cache misses via cache related performance counters. So, if we find unusual behaviour in these counters, we can detect these attacks.

Thus, as is seen from the example, in regular x86 systems, PMCs can be used to detect hardware level attacks, without much performance overhead to the system.

2.3 Extension to ARM

These PMCs are present on almost all devices, including smartphones. For this project, we decided to read those of a Raspberry PI 3, which uses ARM

processors very similar to that of modern day smartphones, and then in case this needs to be extended to smartphones, just the location/command to get the value from PMC registers need to be changed.

2.4 Raspberry Pi 3

To emulate an Android smartphone, we used a Raspberry PI 3. Raspberry Pis are single board computers that run on ARM cortex processors, similar to modern smartphones. They boot directly from an SD card, and has video output as well. So, if we build Android from source, with the target as Raspberry Pi 3, we would have an OS image that is compatible with it, and which it would boot. Connecting it to a display would also be useful for getting information like IP address and kernel logs from it.
It also has built in WiFi and LAN connectivity which are also essential. As will be explained in the Design section, we used a utility called Android Debug Bridge to get access to the PI's terminal from an external device which requires both of these to be connected to the same network.

2.5 TrustZone

ARM's TrustZone technology offers an efficient, system-wide approach to security with hardware-enforced isolation built into the CPU. Using this, we can split CPU execution into a secure world and normal world. This is compatible with Cortex-A processors as well, which are the ones used in the Raspberry PI 3.

The secure and normal world are hardware separated in the processor, so that they cannot access the other's data which is the key idea for it. To utilize this, the OS has to be built for it, i.e the OS running on it should be a 'Trusted OS' which can make use of the 2 worlds to handle sensitive tasks, like entering and storing passwords and biometric data. The combination of TrustZone based hardware isolation, trusted boot and a trusted OS make up a Trusted Execution Environment (TEE), which can be used alongside other security technology.

This is also useful (as mentioned above) to handle a secure boot process as well, since there can be attacks that exploit vulnerabilities in the boot process, which cannot be detected any software running on the system as it wouldn't be initialized yet.

The main use of TrustZone is in embedded devices, since security is a big concern for embedded hardware, especially for connected devices. Connection to the internet provides a venue for hacking, which can range from Distributed Denial of Service (DDoS) attacks to unauthorized access to internal networks. The vulnerability of the Internet of Things (IoT) is especially concerning since DDoS attacks have used millions of unsecured internet-connected devices like Closed Circuit Televisions (CCTVs) to launch massive attacks, like the Mirai attack.

TrustZone can secure a software library or an entire OS to run in the secure area. Non-secure software is blocked from access to the secure side and resources that reside there. TrustZone is based on the principle of least privilege, which means that system modules like drivers and applications do not have access to a resource unless necessary. Software runs in the secure or the non-secure environment. Work that must transpire between the secure and non-secure environments takes place via software called the secure monitor. According to Arm, "This concept of secure (trusted) and non-secure (non-trusted) worlds extends beyond the processor to encompass memory, software, bus transactions, interrupts and peripherals within an SoC. By creating a security subsystem, assets can be protected from software attacks and common hardware attacks".

Cryptographic operations execute in the secure world. Not even Linux kernel operations have access to security features or keys that are isolated in the secure world. Awareness of the TEE is not obvious for end users with kernel access and rights.

Having said all these, TrustZone is not the be-all and end-all of embedded systems security since to implement it correctly requires a lot of development work and any mistake by the developer in implementing it, is potentially a vulnerability that can be exploited.

3 Design and Process Followed

3.1 Desktop Linux Kernel modification

To get familiar with the Linux kernel and the code associated with it, the basic structure of which wouldn't vary much going from an x86 device to an

ARM device, a Linux kernel was built from source on an Ubuntu machine and installed. Then, the machine was restarted into the newly built kernel image, which was verified by running 'uname -r'.
At this point, the kernel was slightly modified and 'printk' statements were added in various location of the boot code of the kernel, to print custom messages and to get familiar with kernel modification.

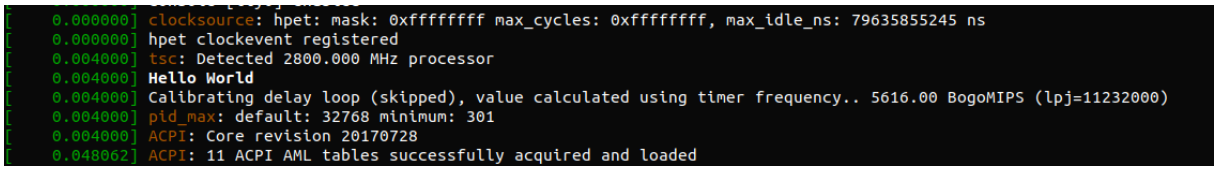

Figure 1: printk output

Now, the kernel was modified to add proc entries to read from and write to, a variable. For this, a proc entry called "sample process" was created and used, to read and write data (in this case, a string) to a variable. Then, "printk" statements were added to this process and the kernel logs checked, to verify its working.

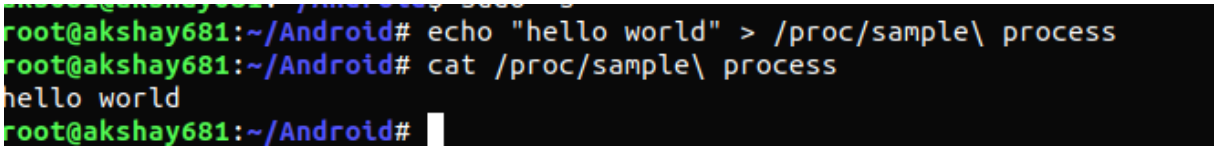

Figure 2: Writing to and reading from a variable

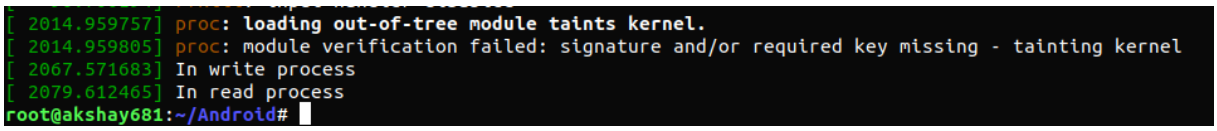

Figure 3: Kernel logs

Apart from proc entries, a new system call was also added to the kernel ('new_call_read') that reads a value passed into it and logs it in the kernel logs.

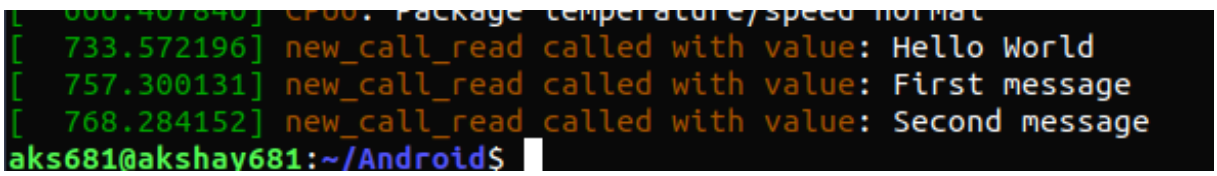

Figure 4: Kernel log with syscall with different values

The output above was obtained by writing a C code that called the system call that was created using the 'syscall' command.

3.2 Booting an OS in Raspberry PI

At this stage, the PI board was booted with a normal cross-compiled linux kernel image (not Android) to test if everything in the board worked fine. For this method, the steps mentioned in this official raspberry website were followed. So, first the Raspberry linux github repo was cloned and then, the steps were followed to run the various 'make' commands.

Then. once the image was created as arch/arm/boot/bzImage, the SD card that had the precompiled Raspbian OS was inserted into the laptop (this was downloaded separately). Then, the image file and the various library files were copied onto the correct locations in the SD card. For Pi 3, the fat32/kernel7.img in the precompiled SD card was overwritten by the compiled bzImage.img (renamed to kernel7.img).

As a minor edit, in the initial source code, before any compilation, the init/main.c file was modified by adding 'pr_notice("Hello World, this is a compiled kernel.")'. The 'pr_notice()' function just runs 'printk()' with the given message. Then, the kernel was compiled again, the Raspberry PI was booted with this and 'dmesg' was run to check the kernel message.

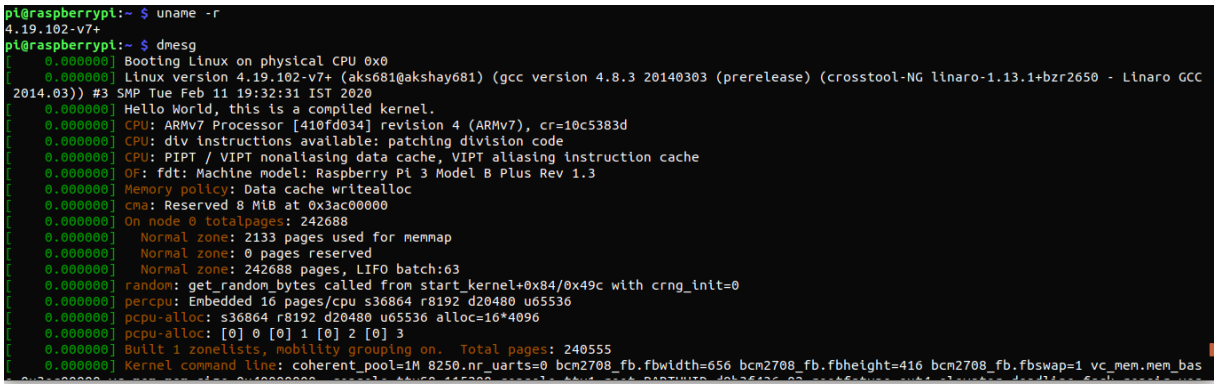

Figure 5: Custom message displayed on the PI

We can also see the source from which the kernel was compiled is shown in the second line where (aks681@akshay681) is shown which is the username and laptop name of the used Ubuntu OS.

3.3 Booting Android on Raspberry PI 3

Now, Android was downloaded and built from source, using the target as 'rpi3-eng' (an engineering build for raspberry pi 3) for it to be compatible with the Raspberry PI. This was over a 100GB download and therefore took some time to download and then to build.

There were some issues here, like the build not completing on Ubuntu 18.04. Then, it was run on an Ubuntu 16.04 machine at which point, it completed. After this, the file system format given in the instructions wasn't right as the PI didn't boot with it, which was later fixed as well.

Now, once Android was running on the PI 3, we needed a way to connect to it from an external device so as to monitor kernel logs and install external apks. For this, we used the Android Debug Bridge utility. Using this, knowing the IP address of the Android device (in this case, by connecting the Raspberry PI 3 to a LAN and using the UI to find the IP address), we were able to connect to it remotely and monitor kernel logs and install apps.

Now, the kernel code was edited to add a 'printk' statement, and call an existing function to display a kernel message (similar to that done in the desktop kernel). The kernel itself was then recompiled and booted to view the kernel messages:

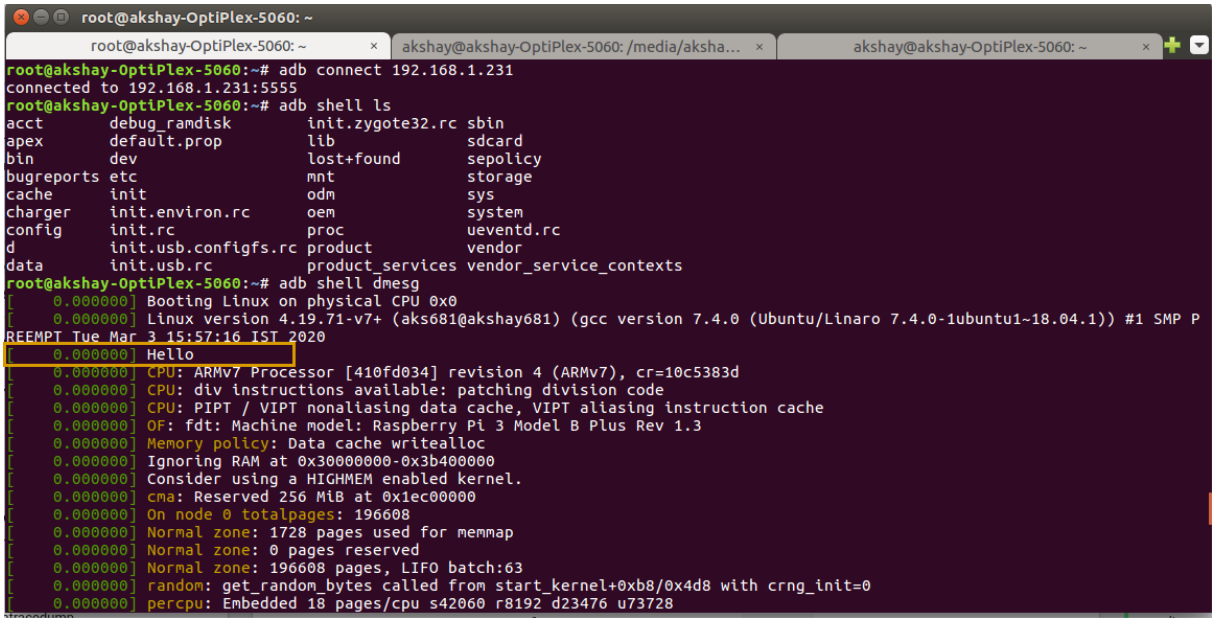

Figure 6: Custom message displayed on the PI

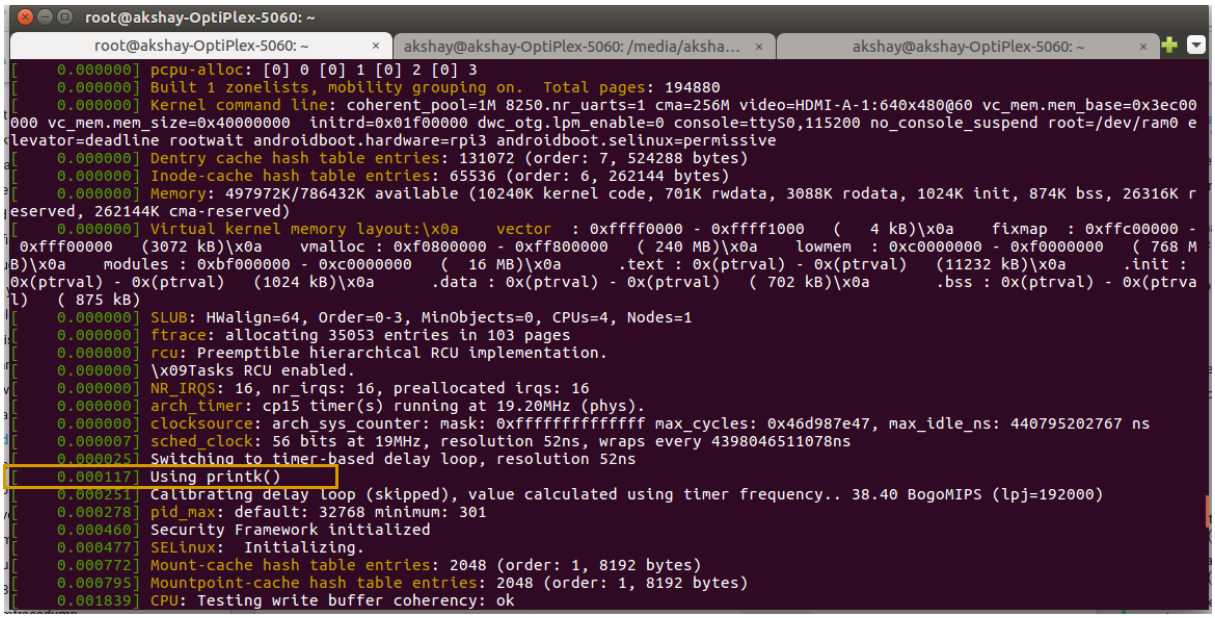

Figure 7: Custom message displayed on the PI

3.4 Exploring Performance Monitoring on PI 3

Right before institute closure, we were working on performance monitoring on the PI board and had some progress. We found the code location, in kernel that had the variables related to performance monitoring. Though just displaying those variables didn't show anything (showed the default 0 value), as it, in its default state didn't monitor anything. This was because it was found that the counters were initialized only after a certain stage during the Android boot process, so we would have to record activities, only after it was initialized. So, we tried incrementing it and recording some values, by setting manual time delays and calling other functions.

It was at this point, that the institute closed. Thus, without access to the PI board any more, kernel emulation using QEMU was tried.

3.5 Kernel emulation using QEMU

To do this, the earlier, already built, version of Android could not be used, as QEMU required the device target to be 'goldfish' instead of 'rpi3-eng' since 'goldfish' had certain features that allowed process emulation on x86 devices, which was what was being done.

Though we were able to use QEMU to boot the image and kernel modi-

fications did work (like in the PI 3), since the kernel itself was different and it was being emulated, the results with performance counters wouldn't translate to real hardware, and so this was not taken further.

The reason for that is that the performance counters are hardware level registers, so when emulating a kernel, those registers would also be emulated using either locations in RAM of the host device or something similar, but would not correspond to the actual address in the device, and hence we wouldn't be able to get reliable values from the counters, which are required for when we use it to detect attacks.

3.6 Other Areas Explored - TrustZone

Since, hardware performance monitoring couldn't be done any longer due to the unavailablity of the device itself, we then tried looking into TrustZone and whether a secure app that uses TrustZone could be implemented.

We found Google's Trusty API which can be used by apps to communicate with the secure world via ports, similar to a network connection, but this also required hardware to compile the OS, with the secure world enabled, to test it, which wasn't available. Since, as mentioned earlier, in the background section, TrustZone requires the OS to be compiled in such a way that it can make use of it. It also uses hardware separation between the normal and secure worlds. So, emulating it was also not an option.

4 Results

Thus, from our work, we were able to:

- Compile a custom linux kernel and boot it in a desktop.
- Modify the above compiled kernel, by adding custom proc entries and system calls.
- Compile and modify a custom linux kernel and boot it in a Raspberry Pi 3.
- Build Android from source, for Raspberry Pi 3.

- Modify the Android kernel and read performance counters in the Raspberry Pi 3, verifying it by connecting to it via Android Debug Bridge.
- Emulate the Android kernel on an x86 machine using QEMU.

As an extension, We also found out that using a custom build of Android OS, with the secure mode enabled, TrustZone could be leveraged, via the Trusty API, to create secure apps for Android.

5 Conclusion

Thus, we have seen that, just like in x86 devices, we can read performance counters in ARM devices as well. This can be very useful to detect attacks which we may not be able to detect, using software only approaches.
Apart from this, TrustZone is a very useful technology implemented by ARM in their processors which can be used to create secure apps for an operating system configured to use it.

References

[1] ARM TrustZone website
https://developer.arm.com/ip-products/security-ip/trustzone

[2] Kernel building in Raspberry Pi 3
https://www.raspberrypi.org/documentation/linux/kernel/building.md

[3] Building Android for Raspberry Pi 3
https://github.com/android-rpi/device$_b$rcm$_r$pi3

[4] Building Android for QEMU: instructions

[5] Using Trusty API for Android
https://source.android.com/security/trusty/trusty-ref

[6] Using Android Debug Bridge
https://developer.android.com/studio/command-line/adb

[7] Brendan Gregg's blogs on performance counters.

[8] Code created/modified for the project, corresponding to some of the screenshots:
BitBucket repo