Vulnerabilities in the Android driver environment*

Marius Biebel
Hochschule München
University of Applied Sciences
Munich, Germany
mbiebel@hm.edu

ABSTRACT

With a market share of over 80%, Android has become an industry standard not only for mobile devices. Mostly Android is seen as an open-source project where everybody can contribute and review the code. However, in practice, a lot of the Android kernel and kernel-environments code gets heavily customized by manufacturers, carriers, and vendors, which can be hard to access for review. Such customizations can pose a threat to the overall device security. Especially drivers are affected by this customization process and can cause serious vulnerabilities. This paper will show different perspectives on the Android device development process, focusing on device drivers, how security research approaches drivers, and how attackers might exploit them.

KEYWORDS

Android, Android drivers, kernel vulnerabilities, SoC Firmware, Fuzzing

1 INTRODUCTION

Modern smartphones contain a lot of different functionalities. A phone is used to manage nearly every aspect of our modern life. The smartphone-OS market is dominated by two big players, Google's Android and Apple's IOs. While Apples IOs is very restrictive and only gets used on their own devices, Android is open source and used by a wide variety of vendors. Android is not only used for smartphones and tablets but also on other devices, like smart devices or for modern IoT products. With a market share of over 80 % in the smartphone market Android is the biggest market out there, and many manufacturers and vendors try to participate in this market.

To stand out in this market, it is vital to have a unique selling point to attract new customers to sell their products. This unique selling point can be, for example, the price or the features. Unfortunately, both of these aspects often come at the expense of security. As a result, smartphones are often developed under pressure to keep costs down and implement new features as soon as possible so that manufacturers can profit from their new unique selling point as long as possible to have an advantage over their competition.

Today's smartphones already come with many functionalities. They have multitouch- screens, several cameras on the front and the back of the phone, several speakers, microphones, a charging port, a headphone jack, a micro SD card slot, NSC, Bluetooth, Wifi, a cellular antenna, fingerprint sensors ... the list goes on. Some of these features are prerequisites for each new modern smartphone others are uniquely for the device or manufacturers. Many of these features get customized with proprietary software for manufacturer's specific features like Samsung's alternative for Apple's airdrop called Quick Share. All these technologies must be integrated into a device and customized to run appropriately with the used Android version. This process is tediously and can consume a lot of time and effort, especially if the manufacturers and vendors want to do this in a secure and tested fashion.

All these features and customizations come with a price. Each of them opens up potential attack surfaces for vulnerabilities and exploits. Between 2015 and 2018, the Android security bulletin program listed over 2179 vulnerabilities with over 1349 publicly accessible patches. 20% were considered critical, 61% were considered high priority, 18% were considered moderate, and less than one percent were considered low priority. 60% of these patches only required changes in a single file. 50% was even fixable in less than ten lines of code, and 9.5% only needed changes in a single line of code to get patched. This shows that most of these vulnerabilities are integrated with the development phase and overseen by internal testing. 92.75% of the analyzed vulnerabilities in the Android security bulletin program were related to code written in C and C++. Only 7.25% of the vulnerabilities are related to code written in Java or similar languages. The Android kernel accounts for 65.9% of the vulnerabilities, while the native libraries layer accounts for 23.9% of the vulnerabilities. The Android runtime, the application framework, and the system applications only account for 9.45% of the vulnerabilities in the Android security bulletin program. From the 65.9% of the kernel vulnerabilities, more than 54% are associated with hardware drivers alone, which shows how severe the problem is.[25]

2 ANDROID DEVICE REQUIEREMENTS

The development process for a new Android device can be very complicated. The Android- system itself is based on the Linux kernel. Android only uses the LTS- versions of the Linux kernel [11]. First of all, the Linux- kernel has to be adjusted for Android. The Android Open Source Project supplies the base- version, which gets heavily customized by manufacturers and vendors. These customized systems branch heavily not only between different devices but also between different versions and regions. Also, vendors and carriers can customize the products before the product finally finds its way to the consumer. All these steps are time-sensitive and developed under pressure.

While most developers publish their customized code, the configuration with which the devices are shipped is often not published. The manufacturers are forced to publish at last the customized Linux kernel of their Android System because the Linux kernel is licensed by the GNU GPLv2 license, which includes a *copy-left*

*This paper was written as part of the practical exam and course work in "Mobile Application Security" at the HM during the winter term 2022.

clause [1] [3]. This requires that customized code of the Linux kernel also has to be published freely. Because some drivers need to be heavily integrated into the kernel to work correctly, they also could be considered as part of the Linux kernel [12]. This could mean that this *copy-left* license also affects these drivers. Nevertheless, often such drivers are not open source. Especially since Android version 8, where Google introduced project Treble. With project Treble Google introduced a new architecture for the Android system, which aimed to separate the manufacturer's customization from the core resources of the Android kernel [15]. The Hardware abstraction layer defined interfaces for vendor customization. Before these, an update of the Android major version required an update for the device drivers, which are integrated into the kernel. These forced the manufacturers to go back in their supply chain and request updates from their suppliers, which could go back up to the silicon manufacturers. If only one of these parties does not supply updates, the whole system update could be at risk. Because of these reasons, the process was cost-intensive and very unattractive. The introduced vendor interfaces of project Treble are *cross-version* interfaces. That made it much easier to update the Android kernel or install a custom Android version on a device [15]. The drivers get stored, with other customizations, in a separated system partition and are accessible for the Android kernel there [17].

If a manufacturer or vendor only wants to release an Android compatible device, manufacturers must validate their product. Therefore they have to comply with the CDD (Compatibility Definition Document). The CDD lists all requirements that the manufacturers must meet before they have a valid Android device for a specified version of the Android system. Google releases a new CDD for each Android version that gets published [10]. The CDD addresses requirements depending on the Android version, for example, the multimedia framework, hardware interfaces, and security requirements. [17]

If vendors and manufacturers want to brand and market their devices as *Android*, they have to request a certification from Google for that because the brand name *Android* is licensed by *Google LLC* [13]. To qualify for certification, the manufacturers must pass tests to prove their compatibility with AOSP. To ensure that, Google publishes the CTS (Compatibility Test Suite), which aims to check that the AOSP requirements are satisfied. This suite is fully automated, and many of these tests verify that the requirements of the CDD are satisfied. Nevertheless, it does not ensure all requirements in the CDD because some of them are challenging to express as a test suite. [17]

Google's services like Google Maps, Youtube, Gmail, or the PlayStore are part of the GMS (Google Mobile Services). If manufacturers and vendors what to ship their devices with these services included, they also have to satisfy the requirements of the GTS (GMS requirement Test Suite). If the Android version is greater than 8 (which contains the redesigned Android of Project Treble), the vendors and manufacturers must pass another test suite called the VTS (Vendors Test Suite). The VTS aims to ensure that the requirements, which were introduced with the architecture changes of Project Treble, are satisfied. [17]

It is important to note that the Manufacturers execute all these tests and not Google or another instance. If all the tests pass, the device is considered a CDD compliant device. Furthermore, it is considered compliant with all the compatibility and security requirements defined by Google. All these requirements ensure the integrity of the whole system and are also important for driver integration. [17]

3 DRIVER DEVELOPMENT PROCESS

The development of drivers for Android devices is not only a task for the manufacturers of mobile devices. Even if the manufacturers are responsible for the finished product, they often buy components for their devices from other suppliers. The supplier often ships their components with the necessary software and service contracts. Nevertheless, suppliers often buy sub-components or services in the development and design of hardware and software. It can involve several suppliers until the origin source of a silicon manufacturer is reached. These can make it hard for manufacturers to get fast updates and patches for their products. This environment for product development offers opportunities for mistakes which can end up as vulnerability in the finished device. [26]

Also, when the manufacturer sells the finished device, that does not mean that the product and drivers are set and done. The products often get shipped worldwide and distributed by different dealers and vendors. So the installed systems need to be customized and configured for the target market. Also, the drivers can be affected by this customization. For example, Wifi, Bluetooth, and cellular drivers have to use different frequencies in different markets.

This heterogeneous environment with different versions and flavors of Android makes it difficult for researchers to approach these issues. [26]

Project treble improved this situation a lot because drivers, which were directly integrated into AOSP, are now developed for different cross-version interfaces. This makes it a lot easier to update a driver or a part of the Android system because the interfaces ensure that there are no unknown dependencies destroyed. [15]

Also, there is a direct advantage for security. Before Project Treble, the integrated drivers must be integrated into the SELinux rules. This can be a very laborious process in which every use case of the driver needs to be addressed. Often manufacturers bypassed these problems by over privileging the driver's access, which can threaten system security because these drivers could also have high if not root access to system resources. But with project Treble interfaces were introduced for most drivers, and these interfaces were already integrated into the SELinux rules for AOSP. [23]

4 DRIVERS IN ANDROID

Most mobile phones have their components integrated into a single board. Primarily they do not support device discovery like USB or PCI that are mostly used by laptops, computers, and servers. For this reason, the Linux kernel uses a *virtual platform bus* with a device tree file to manage these peripherals. A device tree file describes the configuration for all components on a board. It is loaded with the kernel at boot time and makes the kernel aware that these components exists and how to access them.[18]

Drivers for the Linux kernel can be distinguished into three basic types. A character device (char device), a block device, or a network device. A char device can be accessed with a byte stream.

For example, a file can be accessed as a stream. Such drivers mostly implement the usual behavior like the known system-calls like *open()*, *close()*, *read()* and *write()*. Such devices are represented with stream abstractions like the consol driver or the serial ports. They are accessed similarly to filesystem nodes. Block devices are, like char devices, accessed by filesystem nodes, but a block device is a device that hosts a filesystem itself (Like a hard disk). The interface of block devices is different from that for char devices from a kernel perspective. A network device is a device that can exchange data with other hosts in a network. In regular, they are not using system calls like *read()* or *write()*. Instead, they often use *ioctl* functions that manage the transmission of packages and the network resource management. Network devices are, for example, a Wifi card, LAN, or a VPN like Wireguard that can be integrated into the kernel level. [9]

Linux has defined an interface to communicate with drivers. For easy operation syscalls like *open()*, *close()*, *read()*, *write()* or *seek()* can be used. Input/output control (ioctl) is used to depict more complex functionalities. Ioctl is a system call for device-specific operations. It uses three parameters. Frist an open file descriptor, second a device-dependent request code, and third is an integer or pointer, which will be used to transport data into and out of the driver. Ioctl can be used to communicate with devices attached to a computer, for example, via. USB. But it can also be used for kernel extensions or terminal implementation. Despite standardized system calls, the data used by ioctl calls are not bound to standard interfaces. A driver that uses icotl can design and use their own complex data structure for communication. This can make it very difficult to understand and analyze a ioctl interface. Nevertheless, one critical vulnerability in this data can introduce a critical vulnerability to the kernel. Android also uses this ioctl system from Linux. [2] [8] [9]

5 BINARY HARDENING

Drivers often contain vulnerabilities because they often can interact with the kernel with high privileges (Sometimes even as root). Often these drivers are developed by third parties. The source of these drivers is often considered to be no part of the Linux kernel. For this reason, it does not have to be open source. In Android versions, prior to Android Version 8 the customization of the Android system was not separated from APSO and the core resources. This proprietary code made it difficult to test, and also, vulnerabilities often were highly critical because the drivers are so tightly bound to the kernel.[18] [8]

Android introduced several techniques to ensure the integrity of binaries in different Android versions and made them a required technique. These can also be applied to drivers. While the CDD started discussing binary hardening with Android version 9 in 2018, google already started discussing this topic on der Security Enhancements (SE) webpage since 2009. [17]

NX is a technique used by Processors to differentiate between non-executable data in memory and executable data in memory. Here, the CPU will read the NX bit to decide how to process the data. The feature is branded differently by each manufacturer of CPUs. Intel uses the term XD, AMD uses the term NX, and the ARM architecture uses the term XN. Possemato et al. [17] has analyzed the usage of NX protection in manufacturers and vendors binaries. The feature is used in all Binaries of AOSP since API- version 10. However, even in API- version 28, the manufacturers have coverage for their binaries between 50 and 80 percent.

Stack canaries are a common technique to avoid buffer overflow problems. A Buffer overflow usually occurs if a program writes to a memory space outside the allocated memory for this task. Buffer overflow protection modifies the organization of the stack-allocated data to include a canary value. If destroyed, the value shows that the memory is corrupted, and if the canary value is validated, the affected program can be terminated to prevent further damage. Possemato et al. [17] has shown that AOSP has reached a ubiquitous coverage for this technique of nearly 100 percent in their binaries since SDK version 24 while manufacturers binaries still show coverage of 50 to 80 percent.

PIE is a technique that can be enabled at compile time. A binary using PIE is loaded into different locations in virtual memory each time it is loaded. Also, its dependencies are loaded into different locations each time. This makes Return Oriented Programming (ROP) attacks much more complicated to execute [19]. Google first mentioned this in its security enhancements with SDK- version 16. Nevertheless, it was already used in versions prior to 16 by AOSP and manufacturers. Possemato et al. [17] shows that since version 21, a ubiquitous amount of binaries from manufacturers used this technique.

If binaries use shared libraries, they use the Global Offset Table (GOT) to find them. GOT contains pointers that point to the location of the actual function. GOT is populated dynamically and uses lazy binding. When a function is requested, GOT will request it, so a dynamic linker is called to provide it. Since GOT is stored at a predefined place in memory, an attacker could try to override the pointers to execute arbitrary code with elevated privileges [22]. To avoid this problem, the linker must write the pointers to the functions of the shared libraries at the beginning of execution and then mark them as read-only [22]. This technique was first mentioned in SE and widely used by AOSP with SDK- version 16. AOSP reached a coverage for their binaries of over 90% since then, while manufacturers mostly reached a coverage of 50 to 75 percent [17].

Fortify source is a makro compiler extension, which checks the code at compile time for buffer overflows and logs a warning [21]. Google SE first mentioned this enhancement with SDK version 17. In version 28 AOSP has coverage of roughly 75 percent while manufacturers average between 45 and 65 percent [17].

Setuid executables are binaries that can change the user with whom a task is executed — an executable needs special permission to execute such tasks. Since Android 4.3 with SDK 18, AOSP has removed all setuid binaries to prevent abuse for privilege escalation. However, in the manufacturer's Android binaries for Android SDK versions subsequent to 18, 15% of images contained at least one setuid executable. Binaries that appear most frequently are the su (18%), procmem (17%), netcfg (16%), procrank (12%), and tcpdump (11%). Developers often use these binaries for debugging but should be removed for product images. [17]

Vulnerabilities are often caused in the drivers or with the integration of these drivers. [26] It is especially hard to find such

vulnerabilities because a dynamic code analysis is limited by the dependencies to the necessary Hardware. [18]

6 DEVICE DRIVER FUZZING

As mentioned above, customized ROMs' drivers can be highly heterogeneous and run with different configurations. Often drivers are not open source. Out of the 2179 analyzed vulnerabilities till 2018 in the Android security bulletins program, 22.9% were related to closed-source code [25]. This poses a complex challenge for researchers and scientists in this field. If no code is accessible for static code analysis, other ways have to be found to investigate drivers.

Fuzzing is one of the most used techniques to analyze and test Android device drivers. Fuzzing is a kind of testing where the application is automatically called with semi-valid input data that is randomly generated. The application is then monitored for misbehavior, errors, or crashes. If such a constellation occurs, it can be further investigated with the input that triggered the exception. It might be that it cannot be reproduced, so a record can be very helpful. For example, if a Blackbox test, like fuzzing, leads to a device crash, it might be due to a raise-condition and/or incoming IO interrupts. It can be tough to reproduce such a constellation. However, if the test is recorded and can be replayed, further investigation can take place for this situation [24]. Fuzzers are most effective if the generated input can be optimized for the application so that there is a better chance than only brute force to find errors. For this reason, fuzzers are mostly used for applications that take structured input[24]. This can be a challenge for fuzzing ioctl drivers because of their independent interface design. To effectively fuzz a driver, the interface for calls must be known. The easiest way to do that is to monitor calls to the driver and log them to mutate the inputs that where used. But this approach only works for drivers who do not use pointers in their calls, leading to unexpected side effects. Also, it cannot be ensured that all driver functionalities get tested. There might be calls and parameters that trigger unknown functionalities. To ensure coverage for all functionalities it is better to know the driver interface from documentation or source code [8].

Fuzzing is not a new technique. Before it was used to test the Android kernel, fuzzers were developed for numerous other reasons. With *Iofuzz*, *ioattack*, *ioctlbf* and *ioctlfuzzer* there are several fuzzers for the Windows kernel. There are also projects to fuzz the Mac OS kernels. For Linux kernels, the most known Fuzzers are *Trinity* and *syzkaller*. [14]

What makes fuzzing on Android devices also more difficult is the ARM architecture of the system. ARM is based on the RISC (Reduced Instruction Set Computing) while normal PCs work with x86, which is based on the CISC (Complex Instruction Set Computing) architecture. CISC supports several features that are interesting for fuzzing and debugging that are not available in a RISC architecture. For example, it is an interesting information for fuzzing frameworks to know their code coverage. This is also called Coverage-guided fuzzing. While with x86 this information can be requested, ARM does not support such features right away. [14]

With one of the first papers for driver fuzzing on Android Bojiang, et al. [9] published *ADDFuzzer*. Inspired by previous work on the Linux kernel for driver fuzzing, like the tool trinity, they implemented an approach to fuzz drivers on Android systems. They implemented tests for the standard system-calls *open() close() read()* and *write()*. Also, they tried to implement the ioctl interface. To recover the interface, they introduced the thesis that the driver makes the interpretation of arguments right at the beginning. So they recovered the interface from the reversed function that gets called by an ioctl call. This is mostly true, but there can be arguments that get used in other functions or classes of the driver. They tried their approach on a Google Nexus 5 for one week and found three crashes. All of them were reproducible. [9]

With *DEFUZE* Corina et al. [8] has shown an interesting approach for driver Fuzzing. He uses the source code to recover the ioctl interface to generate data for fuzzing. Using sources is better and easier than trying to reverse the driver, but as already mentioned, sources are often not available for drivers. To increase coverage and compare different fuzzers he used two different fuzzer implementations and added his recovered interface information to them. He tested his approach on seven different platforms and claims to have found 36 new vulnerabilities with his approach on several Android devices. While his approach increased the precision with which the Fuzzer is working, there is still room to improve the input generation. Coverage and IO side effects leave room for further work. Also, his approach is limited by the physical device resources. [8]

Inspired by *DEFUZE* Shuaibing et al. [14] picked up the idea and tried to develop it further. They automated most of the manual work from defuze to recover the interfaces and find the device driver's names in Black- and Whitebox environments. They also developed a technique to reverse closed source (Blackbox) drivers to recover their interfaces. The results are promising. By comparing Blackbox and Whitebox interface recovery, they found that fuzzing with the Whitebox recovered interfaces found one bug not found by the Blackbox interface recovery fuzzing. But on the other hand, fuzzing with the Blackbox recovered interface also found a bug that the Whitebox recovered interface has not found. They could not prove that their recovery methods for Blockbox interface recovery can find all valid commands, but there is still proving to find 98% of the valid driver input commands. Also, they tried to prototype a Code-Coverage fuzzing on an x86-64 based Android kernel to use CISC tools for Coverage-guided fuzzing. So they managed to increase the coverage from 14.5% to 35.08%. However, fuzzing on another system architecture can have side effects that can raise false positives or miss out vulnerabilities on ARM architectures. They tested their approach on 11 different devices and claimed to have found 28 vulnerabilities. Nine of them were confirmed by the manufacturers and have CVEs assigned. Eight of these CVEs are referenced in their paper.

But not only a full recovery of the driver interface is a challenge for driver fuzzing. Also is hard to scale driver fuzzing because of the hardware dependencies. The execution of the fuzzing is in-vivo (on device). With *EASIER* Pusogarov et. al. [18] introduces an ex-vivo (of device) approach. They argue that most drivers depend only superficially on hardware and kernel. So they developed an evasion kernel that satisfies these superficial dependencies and enables driver initialization ex-vivo. By evading and not emulating, resources are saved, and user-space tools can be applied for analysis. This

enables easy scaling for fuzzing. They were able to load 77% of their drivers (48 drivers) from 3 different manufacturers. In these drivers, 21 of 26 known vulnerabilities were found. Also, 29 unknown bugs were detected and reported. Twelve of them were confirmed. On the downside, this approach is not suitable for system call analysis. Vulnerabilities that depend on malicious/compromised hardware can not be found, interrupts are not supported, only a limited set of platform bus architectures are supported, and such a setup can produce false positives. [18]

To exploit a vulnerability in a device driver, like the media framework, basic access to the target device is needed. Nevertheless, some drivers manage hardware components like Wi-Fi, Bluetooth, charging port, or cellular networking that might be exploited from outside the system to infiltrate the device. This is especially concerning because such vulnerabilities are zero-click vulnerabilities where no user interaction is needed to exploit these vulnerabilities. These drivers manage Hardware components that are often implemented as a system on a Chip (SoC) which comes with their own firmware. This firmware can also pose a potential attack surface. With *Frankenstein* Ruge et al. [20] introduced a fuzzing framework for non-wireless fuzzing of such firmware in an emulated environment. With this technique, they were no longer limited by physical resources to fuzz the wireless interfaces over the air, which brought their fuzzing speed to an unprecedented level. It is pointed out that it is sometimes hard to fix a vulnerability in drivers that implement standards like Bluetooth because the correct implementation of the standard must still be ensured. Also, a firmware update might be limited by the ROM storage and the physical capabilities of the chip. [20]

7 DEVICE DRIVER REVERSING

Vulnerabilities in Android device drivers are not uncommon. There are regularly and often highly ranked CVEs for Android device drivers. More the half of the Android security bulletins are due to vulnerabilities in device drivers [25]. This shows that they pose a huge attack surface in the Android ecosystem. [16]

While a good amount of work is done for fuzzing device drivers and governance, it is very challenging to get insights into closed source implementations of drivers or chip firmware. Moreover, systematic approaches to reversing are complex because the vulnerabilities and exploits are very individual and hard to compare. The following case study is described to give insights into the challenging work of reversing firmware and drivers, finding vulnerabilities, and exploiting them.

With CVE-2017-0561 Google's Project Zero [6] [7] found an exploit in the Broadcom wi-fi driver that could be exploited to get remote code execution and escalate the privileges on a target device. Broadcom's wi-fi solutions are widely used, not only in Android but also in Apple devices. Often such chips are designed as System On a Chip (SoC). They come with their own proprietary software that already implements all necessary standards that are required. This makes it easy to integrate them into a host system.

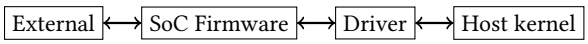
An SoC is like a separate tiny computer with its own storage, memory, and processor (In this case, 640KB of ROM and 768KB of RAM). First, the researchers investigated the device's initialization and found that the initial data loaded into the RAM is not stored in the ROM but provided by the device driver at initialization. Broadcom went to extreme efforts in order to conserve memory. The researchers found several vulnerabilities in the implementation of the 802.11rFT standard for supporting wireless roaming features. Also, they found a vulnerability in the CCKM implementation, which is a proprietor standard from Cisco. Both could be used the trigger a stack overflow. Furthermore, they found a bug in the implementation of the 802.11z for TDLS (tunnel direct link setup). TDLS can be used for communication between two devices without passing it through an AP (access point). This makes an attack even more interesting because an attacker does not have to pose as AP. Also, the standard is open source. The Broadcom implementation of the initialization and the teardown of the TDLS connection are vulnerable due to buffer overflows. However, a buffer overflow does not imply that it can be abused right away. So they first had to evaluate what buffer overflows can be triggered, which fields in the wi-fi frames are under their control and can be used for a buffer overflow, where and how much memory can be overflown, and what targets are interesting to attack in memory. They crafted a wi-fi attack frame, which triggered the overflow, and checked the behavior in the SoC memory to see that the overflow is working as expected. After evaluation, the researchers exploited a memory overflow in the teardown phase of TDLS, which allowed an overflow of 25 bytes. They used this to create two overlapping free chunks in memory. So they used the smaller chunk to manipulate the pointer to a timer that controlled a regularly executed job that searches for other available networks in the area. [6] [5]

From there on, the researchers tried to escalate their privileges out of the wi-fi SoC to the host kernel. Their first approach was to investigate how the SoC communicates with the wi-fi driver on the host system. SoC and host use the same channel to communicate commands and transport the operative data. The only difference is that command frames are marked with a unique identifier. The EtherType for these frames is *0x886C*. To avoid that incoming wi-fi frames trying to abuse this fact, wi-fi frames get validated for this EtherType on both sides, the host driver and the SoC. If an external frame tries to pose as such a command frame, it gets discarded. But with the exploit on the SoC they were able to deactivate this validation in memory. So they can send packages with the EtherType *0x886C* that does not get discarded by the SoC and get forwarded to the host driver where they get interpreted. The host driver contains lots of logic that can be abused with this kind of frames. The researchers even were able to trigger a buffer overflow but found no way to further escalate their privileges from thereon. The only thing they were able to do was to crash the device. Nevertheless, exploiting the driver was not the only approach to attack the host system. The communication between SoC and the host system is running on the PCIe standard. PCIe is a very popular standard because it offers support for Direct Memory Access (DMA) and enables the system to achieve high throughput. PCIe is not regular PCI because it is based on a point-to-point topology. DMA-capable components can be very dangerous for the host system memory. To avoid that an external device can freely access the host memory, the system is protecting itself with additional memory mapping units (IOMMUs). The ARM architecture uses its IOMMU implementation — the System Memory Mapping Unit (SMMU). The SMMU ensures

that a device can only access an assigned memory space to perform operations. Unfortunately, most mobile SoC are proprietary, making it harder to validate if SMMUs are actually in place to protect the host memory. By verifying these dependencies for the SMMU for the Broadcom wi-fi SoC, they found that no SMMU is in place for the SoC or it is configured to access the host's memory directly. This enables them to access the whole host memory from the wi-fi SoC to insert their own arbitrary exploits. From this point, with full access to the whole host memory, known exploits and techniques can be used to infiltrate the target device. They tested their exploit on a Nexus 6P and a Samsung Exynos 8890 SoC. [7]

The researchers from Googles project Zero started their work in December 2016 and published their results in April 2017. This advanced reversing, analyzing, and exploiting shows how *deep* the attack surface can be. Such vulnerabilities are out of the range of common fuzzing frameworks but still offer vast opportunities for attackers. Even if this is done by top researchers from one of the most prestigious tech companies out there, it is hard to believe that there aren't other parties that try to find and use such exploits for their own advantage.

To systemize the problem. From a Linux kernel perspective, a device driver is a component that connects to a resource. However, a driver is just application logic that provides an interface for interacting with another resource like an SoC or application logic like multimedia frameworks. This resource might then also interact with other devices like an AP. Fuzzing the device driver from userspace with syscalls is a well-known technique even on Android systems. Also, there are approaches for analyzing and fuzzing from outside the system. But the connection between the firmware and the driver is very hard to analyze, especially on closed source platforms that do not allow further access to this communication. But still, this is important as the researchers of project zero show because the firmware can give access to resources of the host system due to bad integration or further exploits that can be triggered from this perspective. In theory, each connection and each direction in this architecture should be tested because each perspective could be the point of view of an attacker trying to exploit his way through the system.



8 RELATED AND FURTHER WORK

Not only device drivers are a security problem on the Android platform. Possemato et al. [17] show in their paper "Trust, But Verify" that there overall structural problems in the design process of new Android devices. With the CDD, CTS, and VTS, google tried to define basic requirements that must be satisfied by manufacturers to brand their devices as *Android* and to use the Google Mobile Services (GMS). Google has the juristic tools to enforce these requirements. Nevertheless, Possento et al. have found that 20% of their investigated images (579 images) violate at least one requirement of the CDD. Even 11 devices that are branded by google themselves violate the requirements of the CDD.

In the kernel and system environment, not only device drivers are a problem. Also, compliance issues are problematic. Most of these compliance violations introduce vulnerabilities because best practices are violated, which should be enforced by the requirements defined by google. Next to drivers, there are often bad SELinux policies or vulnerabilities in Initscripts. Their analysis shows that the problems regarding the compliance violations are not improving over time with newer versions. [17]

Also, the overall customization of AOSP from vendors, manufacturers, carriers, and suppliers can introduce band configurations that can be exploited, as shown by Aafer et al. [4] who have analyzed several ROMs from different manufacturers with a differential approach. They implemented several exploits for different devices tampering with system settings, triggering emergency broadcasts, sending SMS, and stealing emails. [4]

9 CONCLUSION

Drivers on the Android platform offer an incredible attack surface for exploits. With more than 54% of the issues in the Android security bulletins, they take a lion share [25]. The supply chain that develops obstructs and distributes these drivers do not make it easy for third parties to investigate their security. The ecosystem of Android and device drivers looks very heterogenous with hundreds of different devices and thousands of different versions. But this appearance is deceptive. Often chips and modules for smartphones are used in several devices from several manufacturers. Drivers are similar, if not the same, for different devices.

It is shown how these problems are seen from different perspectives. First, a management perspective focuses on the requirements for developing a new Android product. Requierements and recommendations from the CDD, CTS, GTS or the VTS are discussed. And best practices are explaint. Often requirements are not met, and also recommendations are discarded. Appropriate and secure driver integration is often seen as a cost case that can be bypassed.

From a research perspective, it is hard to find systematic ways to analyse this heterogeneous environment that is often closed source. Mostly Blackbox approaches are used to investigate these areas. Unfortunately, work in this sector is still slowed down by issues that block access to these components. It is understandable that companies keep their code private in order to protect their intellectual property. Nevertheless, this also hinders security research that is also for the benefit of these companies. For this reason, a lot of resources in security research are spent on interface recovery and emulation/simulation in device fuzzing. Such efforts could be saved if interfaces for drivers were documented. In best case as machine-readable interface documentation.

From an attacker's perspective, they can take advantage of the fact that they might know their target device. As shown in the case study, it is possible to find exploits in target devices with enough resources. This might be a valid method for nation-state actors to intrude a target.

REFERENCES

[1] 2021. GNU General Public Licence Version 2. (14 December 2021). https://www.gnu.org/licenses/old-licenses/gpl-2.0.html

[2] 2021. ioctl(2) — Linux manual page. (21 December 2021). https://man7.org/linux/man-pages/man2/ioctl.2.html

[3] 2021. Linux Kernel Licence Rules. (14 December 2021). https://www.kernel.org/doc/html/v4.16/process/license-rules.html

[4] Yousra Aafer, Xiao Zhang, and Wenliang Du. 2016. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 1153–1168. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aafer

[5] Gal Beniamini. 2017. CVE-2017-0561. (11 April 2017). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0561

[6] Gal Beniamini. 2017. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1). (04 April 2017). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

[7] Gal Beniamini. 2017. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 2). (11 April 2017). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html

[8] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. DOI: http://dx.doi.org/10.1145/3133956.3134069

[9] Baojiang Cui, Yunze Ni, and Yilun Fu. 2015. ADDFuzzer: A New Fuzzing Framework of Android Device Drivers. In *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*. 88–91. DOI: http://dx.doi.org/10.1109/BWCCA.2015.57

[10] Android Source Documentation. 2021. Android Compatibility Definition Document. (25 October 2021). https://source.android.com/compatibility/cdd?hl=en

[11] Andoid Source Documentation. 2022. Android Common Kernels. (10 January 2022). https://source.android.com/devices/architecture/kernel/android-common?hl=en

[12] Sudesh Kumar, Lakshmi Jayant Kittur, and Alwyn Roshan Pais. 2020. Attacks on Android-Based Smartphones and Impact of Vendor Customization on Android OS Security. In *Information Systems Security*, Salil Kanhere, Vishwas T. Patil, Shamik Sural, and Manoj S. Gaur (Eds.). Springer International Publishing, Cham, 241–252. DOI: http://dx.doi.org/10.1109/SP40001.2021.00074

[13] Google LLC. 2007. Auskunft zu einer Unionsmarke 006410856. (06 November 2007). https://register.dpma.de/DPMAregister/marke/registerHABM?AKZ=006410856&CURSOR=0

[14] Shuaibing Lu, Xiaohui Kuang, Yuanping Nie, and Zhechao Lin. 2020. A Hybrid Interface Recovery Method for Android Kernels Fuzzing. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. 335–346. DOI: http://dx.doi.org/10.1109/QRS51102.2020.00052

[15] Iliyan Malchev. 2017. Here comes Treble: A modular base for Android. (12 May 2017). https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html

[16] Huasong Meng, Vrizlynn L.L. Thing, Yao Cheng, Zhongmin Dai, and Li Zhang. 2018. A survey of Android exploits in the wild. *Computers Security* 76 (2018), 71–91. DOI: http://dx.doi.org/10.1016/j.cose.2018.02.019

[17] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. 2021. Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In *2021 IEEE Symposium on Security and Privacy (SP)*. 87–102. DOI: http://dx.doi.org/10.1109/SP40001.2021.00074

[18] Ivan Pustogarov, Qian Wu, and David Lie. 2020. Ex-vivo dynamic analysis framework for Android device drivers. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1088–1105. DOI: http://dx.doi.org/10.1109/SP40000.2020.00094

[19] Redhead. 2012. Position Independent Executables (PIE). (28 November 2012). https://www.redhat.com/en/blog/position-independent-executables-pie

[20] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. 2020. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 19–36. https://www.usenix.org/conference/usenixsecurity20/presentation/ruge

[21] Siddharth Sharma. 2014. Enhance application security with FORTIFY SOURCE. (26 March 2014). https://www.redhat.com/en/blog/enhance-application-security-fortifysource

[22] Hutzaifa Sidhpurwala. 2019. Hardening ELF binaries using Relocation Read-Only (RELRO). (28 January 2019). https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro

[23] Android Sources. 2018. SELinux for Android 8.0. (13 February 2018). https://source.android.com/security/selinux/images/SELinux_Treble.pdf

[24] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. 2018. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 291–307. https://www.usenix.org/conference/usenixsecurity18/presentation/talebi

[25] Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. 2019. Towards Understanding Android System Vulnerabilities: Techniques and Insights. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*. Association for Computing Machinery, New York, NY, USA, 295–306. DOI: http://dx.doi.org/10.1145/3321705.3329831

[26] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *2014 IEEE Symposium on Security and Privacy*. 409–423. DOI: http://dx.doi.org/10.1109/SP.2014.33