Raspberry Pi 4 Malicious Bootloader

Andrew Rust
Columbia University
acr2213@columbia.edu

Emily Soto
Columbia University
es4060@columbia.edu

ABSTRACT

Hardware and rmware of computer systems pose critical yet often overlooked attack vectors when compared to mainstream concerns around software vulnerabilities. Systems exploited at the hardware level potentially allow for root-level access, remote code execution, and perhaps entire network access without privilege escalation required in software exploits. This paper details a procedure exploiting one such vector through a modied bootloader (U-Boot bootloader) on a Raspberry Pi 4 that perform a memory exltration attack. By manipulating the bootloaders hardware initialization and rmware code, we demonstrate the potential to covertly extract memory dumps over a network, revealing sensitive data before device initialization completes. This hypothetical attack outlines the security risk that naive bootloader congurations allow for, particularly among Internet of Things (IoT) devices, which typically lack robust security features like Secure Boot. The ndings underscore the necessity for thoughtful security measures at the hardware level, such as secure boot mechanisms or hardened bootloader congurations, especially as increasingly ubiquitous "smart devices" are produced without security in mind.

ACM Reference Format:
Andrew Rust and Emily Soto. 2024. Raspberry Pi 4 Malicious Bootloader. In *Proceedings of* . ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

In the scene of cybersecurity, hardware or rmware attacks are generally more dicult to implement but are particularly dangerous because they can give attackers root access (level 0, kernel level) to devices usurping any software security in place. These attacks require knowledge of a device's hardware and booting software to properly interject any malicious code used as an exploit. System-on-chip (SoC) devices or Field-Programmable Gate Arrays (FPGAs) that use common bootloaders, like U-Boot or GRUB, are especially at risk because they need exibility for their specic hardware, but often lack adequate scrutiny from a hardware security perspective that would be present in more prominent technologies like smartphones.

Embedded systems like these (SoCs and FPGAs), present an niche, but accessible attack surface given knowledge of their startup process. U-Boot, a widely utilized bootloader in these systems, is responsible for the initialization of hardware components and is the vector used in this paper alongside a nominally-used SoC: The Raspberry Pi 4 (RPi4). The U-Boot codebase, as well as documentation on the Raspberry Pi's hardware/rmware initialization is freely accessible and allows for our hypothetical attack to take place. The code used comprises of mostly assembly and C code as well as initialization scripts for a the RPi4. Given a niche product line using these technologies, and a lax security team, we demonstrate an attack where malicious code is hidden in seemingly benign, but obscure header and environment les, which allows for data exltration.

1.1 SCOPE & ARCHITECTURAL DETAILS

This research is conducted using a Raspberry Pi 4 with 8GB of memory, included with a modied U-Boot bootloader that exltrates memory on startup over the local network. The exploit take place temporally during the early stages of the boot process before the Linux kernel [LKM] takes over, we exltrate said data over the local network but also provide alternative means of exltration given dierent circumstances. The memory dumps themselves, expose potentially sensitive information, which we also analyse and categorize in this paper. Alongside benign application data(likely related to setup, telemetry, or background processes), sensitive les (.txt, .png) and encryption (ssh) keys are among what we show can be covertly accessed and transmitted in this attack. This attack mostly serves as a blueprint for other attacks following similar vectors of targeting hardware and rmware as opposed to exclusively software. We also provide directions for future research along these lines in an eort to advocate for robust security measures in and around the bootloader especially for devices less often under analysis.

2 RELATED WORK & OTHER ATTACKS

Embedded systems such at the Raspberry Pi often use specialized bootloaders due to their customized hardware congurations and limited resources. These bootloaders are designed to be lightweight and ecient, but they can sometimes lack the advanced security features found in more sophisticated computing environments. Consequently, vulnerabilities in these bootloaders can leave devices open to exploitation, potentially allowing attackers to control or disrupt critical functions. One signicant area of focus in embedded system security involves defending against cold boot attacks. These attacks exploit the tendency of RAM to retain data for a brief period after power has been removed. By quickly rebooting a system and dumping the contents of memory, an attacker can potentially recover sensitive information, including cryptographic keys and passwords, directly from RAM. Notably, in Won et. al, SDRAM information is recovered from the Raspberry Pi 4B+ with this particular methodology. While R considered the possibility of an "evil maid" attack–a type of security threat where an attacker gains physical access to a target device, such as a laptop, smartphone, or hard drive. The attacker typically inltrates the device by either breaking into the target's room or gaining temporary access when the target is away. Once inside, the attacker can install malware or manipulate the device's hardware to gain unauthorized access to sensitive information. We took inspiration from IOT Bootloader devices-mention any vulns or research on GRUB, Secure Boot, even TPM or PSP stu rowhammer warm, cold-boot attacks evil maid attacks attac,kse, and perform a memory dump after initializing a network connection with a local device. In this case, the local IP address is known, but in a practical instance there might need to be additional forensic measures taken place. We also outline several di erent alternative methods for ex ltration in the following section, including an ssh tunnel, simple https hando or writing to a more permanent location in storage (SD card, SSD) to be accessed later.

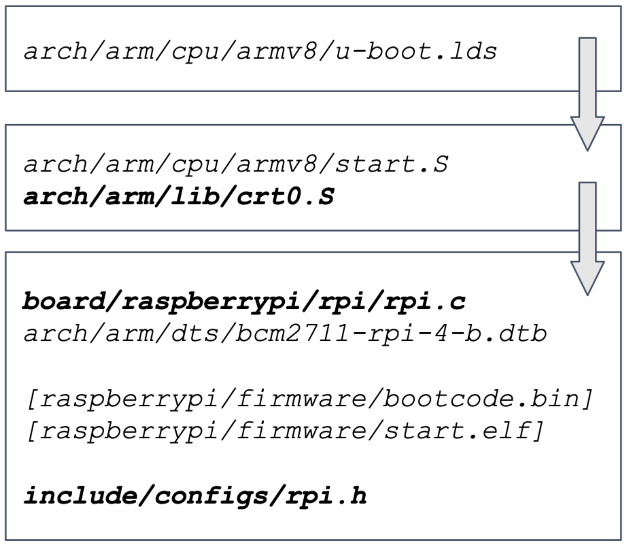
arch/arm/cpu/armv8/u-boot.lds
arch/arm/cpu/armv8/start.S
arch/arm/lib/crt0.S
board/raspberrypi/rpi/rpi.c
arch/arm/dts/bcm2711-rpi-4-b.dtb
[raspberrypi/firmware/bootcode.bin]
[raspberrypi/firmware/start.elf]
include/configs/rpi.h

Figure 1: High-level booting process of U-Boot on Raspberry Pi 4, bolded les are modi ed in the malicious con guration

4.1 ALTERNATIVE EXFILTRATION METHODS

While the ex ltration method demonstrated here is transmitting the memory dump over the local network, alternative approaches can be employed depending on the attacker's resources, the target environment's security con guration, and any desire for stealth. The straightforward method used in this paper is the manipulation of environment variables within the bootloader to establish a connection to another device on the network and transmit using TFTP. This instead could redirect critical data to a benign-looking, but attacker-controlled, le in a speci c folder. This le could be stored on more permanent media like an SD card or any attached storage (SSD/HDD).

Another technique would be to wait for the device to establish internet connectivity (via a successful DNS query or HTTPS connection), then pass the data over HTTPS to a attacker-controlled address. an HTTPS connection during its normal operation. At this point, ex ltrated data could be subtly intermixed with other outgoing tra c, depending on the level of suspected network monitoring done on the device. Augmenting this strategy, an attacker could use an SSH tunnel(s) to pipe out data to a remote server, especially if that is already the common means of transmission on the device, which is possible for IoT devices and the like.

5 DATA ANALYSIS

We found that the GPU-related code on the Raspberry Pi 4b+ was constructed with a proprietary assembly language Videocore IV– luckily, some work toward open-sourcing VC4 exists, and allowed us to analyze more thoroughly.

Continuing our exploration further into the Raspberry Pi 4B+'s proprietary systems, especially regarding the Videocore IV GPU-related code, it's evident that the developmental shift towards partial open-sourcing has been bene cial. This openness has provided us with clearer insights into the often opaque world of embedded GPU functionalities. The assembly language used, although proprietary, has started to reveal its structure thanks to the ongoing e orts towards open-sourcing. This allowed us a more in-depth glimpse into the crucial elements of GPU processing, particularly how encoding and decoding tasks are managed at a low level.

Within the les we analyzed, the transition of certain portions of the boot sequence storage to EEPROM in the Raspberry Pi 4 marks a design evolution from earlier models. This transition potentially enhances the robustness and exibility of the rmware updates and management. Moreover, the ability to retain and execute bootcode independently of external storage media underscores a more resilient approach to maintaining critical operations, which are essential for reliability in varied application environments.

Using the binwalk -E utility, we created a histogram of the entropy in the two proprietary les of note, start4.elf and *bootcode.bin*, and we utilized a python script to output a histogram of the bytes of our dumped code. We did not nd any encrypted sections of note. The absence of encrypted sections within these primary les—while facilitating easier exploration and academic critique—also points towards prospective security risks, speci cally in environments where sensitive operations are conducted using the Raspberry Pi.

A particularly interesting string of bytes that appeared consistently across boots was loading "Micron PoP 44nm 4Gbit," which seems to con rm that our particular Raspberry Pi 4B+ was out tted with this particular SDRAM. We were unable to nd information readily available about the particular SDRAM model the Raspberry Pi 4B+ used anywhere else.

Most of our interesting extracted information, however, related to the setup of the graphics drivers. Encoding and decoding in particular.

An interesting section of note was .guard1 of start4.elf. It seemed that .guard1 contained, guard variables or canaries—special values placed in memory adjacent to bu ers to detect and prevent bu er over ows. Ostensibly, these values are checked during boot to ensure they haven't been altered during execution; any change might indicate that a bu er over ow has occurred, which would seemingly prompt the system to halt operations/boot.

5.1 RESULTS & INTERPRETATIONS

Our investigation into the Raspberry Pi 4B+'s systems yielded substantial details that enhance our comprehension of its internal operations, particularly focusing on the proprietary

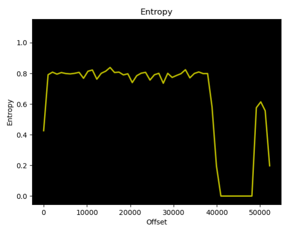

Figure 2: Entropy graph of bootcode.bin

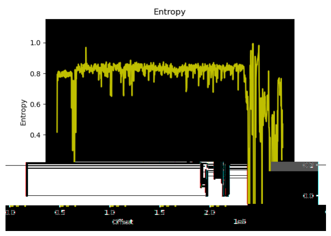

Figure 3: Entropy graph for start4.elf

les involved in its boot process and GPU-related functionalities. From our entropy analysis using the binwalk -E utility, the lack of encryption in the les *start4.elf* and *bootcode.bin* suggests these components are more accessible for analysis than if they were encrypted. This potentially exposes them to higher security risks as it might allow malicious entities easier insights and manipulations, but it also simplies the process for legitimate developmental modications or academic scrutiny. The examination also brought to light substantial insights into the loading mechanisms and behaviors during the boot process. For instance, examining the entropy and strings within these les helped identify regular patterns and key operations occurring at boot, such as initial hardware checks and driver setups. Moreover, our analysis uncovered the robust nature of the Raspberry Pi 4's architecture concerning potential buer overow attacks. The presence of mechanisms like guard variables suggests a deliberate eort to incorporate security at the foundational level of rmware architecture. This is a crucial feature, particularly in a widely-used educational and developmental platform like the Raspberry Pi, which often handles diverse and potentially unsecured code. Continuing our analysis, the ndings from our examination have signicant implications for both the maintenance and advancement of Raspberry Pi systems. The straightforward nature of the bootcode and its lack of robust encryption can provide an accessible platform for educators and developers to teach and experiment with low-level programming and system design. However, this openness also necessitates the implementation of enhanced security measures if the device is deployed in sensitive environments. Further, the absence of sophisticated encryption in key operational les might imply lower processing overhead, leading to quicker boot times and less computational strain during startup. This characteristic is especially advantageous for educational purposes and experimental projects where simplicity and accessibility are priorities. Nevertheless, it also suggests the potential need for additional security measures when the Raspberry Pi is implemented in more sophisticated, security-sensitive projects, i.e, one which nation-state actors may take an interest in.

5.2 MAIN CONCLUSIONS

Bootloader attacks by way of core dumps are quite possible on SoCs where there is not robust booting procedures in place. It is not to say that open bootloaders like U-Boot are worse than secure boot, but simply that bootloaders are a viable entry point for attackers with any intent, potentially even manufactoeres themselves with propreitary booting procedures. It is to show what is possible, not that any specic technology is any more or less vulnerable. At the very least, U-boot's code is available and auditable for security researchers to conrm or any suspicions that they may have. In our case, the use of an open source bootloader made extracting the proprietary code far more ecient, but given that none of it was encrypted, grabbing the assembly was not overly dicult. On the contrary, the biggest hurdle to analyzing proprietary RPI code was Broadcom's VC4 assembly, which has not been entirely reverse engineered yet. Open-source software comes with both costs and benets. By gaining a deeper understanding of the rmware, open-source driver developers may work to enhance the performance and capabilities of the Raspberry Pi 4B+. Enhanced knowledge of hardware-specic functions allows for more ecient translations between the software applications and the physical components of the device, signicantly reducing compatibility issues that can arise from miscommunication between hardware instructions and custom driver operations. However, given their early and integral stage in the boot process, malicious versions of start4.elf and bootcode.bin hold the potential to be disastrous. With detailed knowledge of rmware operations, particularly if undocumented features or vulnerabilities are uncovered, malicious actors can craft targeted attacks that exploit specic weaknesses in the system. For example, they could exploit loopholes in the boot process or memory management functionalities identied through the analysis. These could be used to insert malicious code, create backdoors, or initiate privilege escalation attacks, where the attacker gains higher access privileges than originally intended. Given the widespread use of Raspberry Pi devices in various applications, including sensitive and critical environments, such exploitation could have far-reaching consequences.

5.3 FUTURE RESEARCH

To conclude our work, we would like to present related areas upon which further study is possible and invited. Firstly related to data analysis, a plethora of reverse engineering and binary analysis tools currently exist but perhaps additional focus on tools or methodologies for low-level initialization code or proprietary binaries for custom SoCs is of evergreen interest. Related to the attack outlined here, an improved in□ltration technique or obfuscation strategy of the malicious bootloader may provide a more holistic red-team attack for device manufacturers looking to secure systems from cold boot attacks, malicious bootloaders, or any hardware/□rmware-targeted vectors. Our approach was reductive, making assumptions about the security processes in place, so perhaps analysis on a system deployed in a more realistic, yet still a research environment would net a more substantive attack. In the wider context of secure booting, penetration-testing hardware-based security measures on more mainstream boards like trusted platform module (TPM) on Intel chips and platform security processors (PSP) on AMD chips would be substantive spaces for future research to be done. More speci□c to IoT devices would be security analysis on other bootloaders or hardware initialization code. A very lightweight bootloader for low-spec IoT devices might be a more secure solution than a bootloader for a whole operating system like U-Boot or GRUB. Lastly, smaller batch CPUs or smart-device speci□c processors could be potential ex□ltration targets too instead of just physical memory as was the focus here. An addendum to our e□orts on memory ex□ltration could be CPU registers, caches, or colder storage locations like SD cards.

Y. -S. Won, J. -Y. Park, D. -G. Han and S. Bhasin, "Practical Cold boot attack on IoT device - Case study on Raspberry Pi -," 2020 IEEE International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA), Singapore, 2020, pp. 1-4, doi: 10.1109/IPFA49335.2020.9260613.

Boursalian A, Stamp M. BootBandit: A macOS bootloader attack. Engineering Reports. 2019; 1:e12032. https://doi.org/10.1002/eng2.12032

references