Dark Firmware: A Systematic Approach to Exploring Application Security Risks in the Presence of Untrusted Firmware

Duha Ibdah, Nada Lachtar, Abdulrahman Abu Elkhail, Anys Bacha, Hafiz Malik
University of Michigan, Dearborn, USA
{dhibdah, nlachtar, abdkhail, bacha, hafiz}@umich.edu

Abstract

Compromising lower levels of the computing stack is attractive to attackers since malware that resides in layers that span firmware and hardware are notoriously difficult to detect and remove. This trend raises concerns about the security of the system components that we have grown accustomed to trusting, especially as the number of supply chain attacks continues to rise. In this work, we explore the risks associated with application security in the presence of untrusted firmware. We present a novel firmware attack that leverages system management cycles to covertly collect data from the application layer. We show that system interrupts that are used for managing the platform, can be leveraged to extract sensitive application data from outgoing requests even when the HTTPS protocol is used. We evaluate the robustness of our attack under diverse and stressful application usage conditions running on Ubuntu 18.04 and Android 8.1 operating systems. We conduct a proof-of-concept implementation of the attack using firmware configured to run with the aforementioned OSs and a mix of popular applications without disrupting the normal functionality of the system. Finally, we discuss a possible countermeasure that can be used to defend against firmware attacks.

1 Introduction

Traditionally, firmware has been perceived as the invisible software that breathes life into the digital world around us at the stroke of a button. In addition to firmware's role of silently configuring and initializing platform resources, it serves the purpose of maintaining the health of the underlying hardware through periodic management cycles.

Although firmware is often regarded as a decoupled entity from the application layer, the aforementioned management cycles could be re-purposed to scrape system memory and therefore undermine the confidentiality guarantees systems provide for user data. As a result, it is critical to systematically evaluate the security of systems when faced with adversaries that can harness firmware to maliciously scrape memory for sensitive user data. This is especially important as the number of supply chain attacks that can result in malicious firmware being installed on systems continues to rise [1–3].

A significant body of work has explored various attacks aimed at harvesting sensitive data from system memory [4–13]. Work by Hizver et al. [4] demonstrated the use of introspection attacks through hypervisors to extract credit card data from Point of Sale (PoS) systems running as virtual machines. Other work examined memory scraping techniques that involve physical access [6–13]. For example, Halderman et al. [6] explored how memory modules could be extracted from a machine and then scanned to recover cryptographic keys. Similar work investigated the re-purposing of different hardware ports to collect memory dumps from smartphones once a device has been stolen [8]. Other work [9–13], considered the use of malicious peripherals to collect memory dumps through DMA attacks.

Unlike prior work, we propose a novel firmware-based attack that covertly scrapes memory for sensitive user data. Our attack doesn't require the use of special hardware or physical access. It is not limited to environments that employ virtual machines and can be applied to most computing systems. However, despite such advantages, the low-level nature of this subsystem makes it difficult to gain insight into the application layer and how its data is managed. This lack of insight introduces a layer of complexity for systematically collecting sensitive user data through firmware. Another challenge relates to the limited execution cycles firmware is allocated during runtime. Reclaiming compute resources beyond the intended time slice can cause the system to malfunction and applications to become unresponsive. This is especially true for mobile environments where app responsiveness and energy efficiency are treated as first order constraints for consumers.

In this paper, we explore the possibility of leveraging system management cycles to collect user passwords from outgoing HTTP requests. We show that periodic system interrupts that are used for managing platform events, could be harnessed to reliably and efficiently extract sensitive user data

through the use of malicious firmware. We show that this is possible even when applications employ the secure HTTPS protocol. We evaluate the effectiveness of this approach across desktop and mobile systems by extensively testing popular web services and mobile apps, such as Instagram, Facebook, and Linkedin running on different hardware configurations. We characterize the robustness of our approach under stressful app usage conditions while considering a diverse set of applications from six different categories. We build a proof of concept for our proposed attack using real system firmware that is configured to run with Ubuntu 18.04 Linux and Oreo 8.1 Android operating systems combined with a mix of popular web services and apps. We devise a mechanism that achieves low overhead across both x86-64 and ARMv8 architectures. We accomplish this by limiting memory scans to the user accessible dirty pages in memory that obviates the need for parsing the entire memory subsystem. Finally, we discuss a possible hardware-based countermeasure that can be used to defend against such attacks.

Overall, this paper makes the following contributions:

- Presents a novel attack that leverages platform management cycles to collect sensitive data from HTTP requests even when HTTPS is used.
- Conducts a proof-of-concept implementation of the attack using real firmware configured to run with Ubuntu Linux and Android Oreo operating systems.
- Characterizes the robustness of the proposed approach while running a diverse set of popular web services and mobile apps under different platform configurations.
- Demonstrates how firmware can efficiently leverage page tables to covertly extract sensitive information without disrupting the normal functionality of desktop and mobile systems, and discusses a possible countermeasure that could be employed to defend against such attacks.

The rest of this paper is organized as follows: Section 2 provides background and motivation information for our attack based on experimental data. Section 3 illustrates the threat model. Section 4 details the design and algorithms for the proposed attack. Section 5 presents an experimental evaluation. Section 7 details related work; and Section 8 concludes.

2 Background and Motivation

2.1 System Firmware

System firmware is an essential component that abstracts away hardware details from the OS as a way of providing a common OS agnostic view across multiple platforms. It can be divided into two phases: boot time and runtime. The boot time phase is responsible for initializing the platform's hardware resources during in preparation for launching the OS. Although the main responsibility of system firmware entails discovering and initializing the hardware resources, it also plays a role in providing runtime services to the OS. Since the OS doesn't have enough insight about the details of a given platform, it relies on firmware to perform certain actions on its behalf. Such actions include the gathering and handling of machine state information in response to hardware events, as well as housekeeping tasks that include power management [14, 15] and system authentication [16].

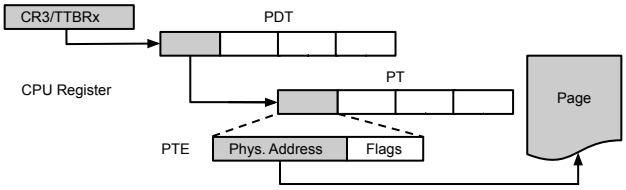


Figure 1: A high level overview of the address translation process from virtual to physical memory.

Transfer of control from the OS layer into system firmware is achieved through a special hardware interrupt. This mechanism enables firmware to conduct platform management tasks transparently from the OS. Therefore, before passing control to the OS, it is incumbent upon firmware to configure the underlying hardware to generate a set of manageability interrupts. This mechanism prompts the interrupted core to invoke firmware execution and put the processor into a special mode that we will refer to as management mode. This mode grants firmware access to system resources. In addition, this mode shadows resources away from the OS and application layers. As such, any transactions issued by firmware would not be visible to upper layers.

2.2 Virtual Address Space

Virtual address space is a fundamental component that provides isolation between processes while presenting each process with the illusion that it has access to the entire address space. The OS accomplishes this through paging, a mechanism that maps process virtual addresses into physical ones. Virtual address transactions issued by a given process result in the kernel walking a series of translation tables that it uses to determine the target physical address of the mapped page. The physical addresses that mark the start of such tables are tracked through dedicated CPU registers. x86 systems make use of a single register (`CR3`) to maintain both user and kernel space addresses, while ARM uses separate registers for mapping user and kernel space addresses (`TTBR0` and `TTBR1`).

Translation tables are organized in a directory like structure often consisting of a page directory table (PDT) followed by page table entries (PTE). However, this directory structure can consist of multiple levels that include intermediate directories. The page table entries serve an important role since they hold

the physical addresses of the pages that represent their virtual address counterparts. In addition, each PTE contains different flags that describe the properties of the page it's associated with. Typical flags include bits that are used to indicate the presence of a page in memory, whether a page is dirty, and if a page is accessible from user space. A high level overview of the address translation process is illustrated in Figure 1.

2.3 The Case for Firmware-based Attacks

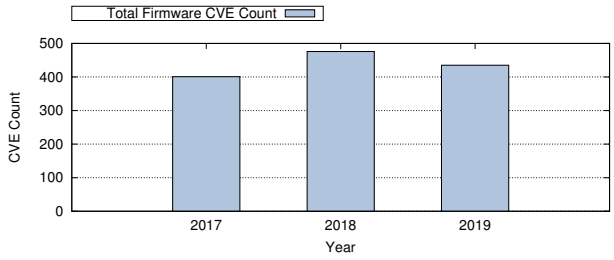
According to Shane Wall, the director of HP Labs [17], compromises at the firmware level are attractive to attackers since malicious firmware remains persistent regardless of OS reinstallations and storage reformatting, often necessitating a hardware replacement. Since our attack relies on the presence of malicious firmware on the system, we discuss possible ways this could be accomplished.

Pushing malicious firmware to systems can be performed through the software supply chain process by compromising live update utilities that are shipped with systems to accommodate future updates. Such an attack was demonstrated by Operation Shadow Hammer that was discovered in 2019 by Kaspersky [3]. Similar attacks can also be performed after deployment by exploiting vulnerabilities in the firmware update process administered by the OS. For instance, prior work explored how vulnerabilities could be leveraged for injecting malicious firmware including the use of remote updates that can be performed from user space while bypassing existing safety measures, such as secure boot [18–24].

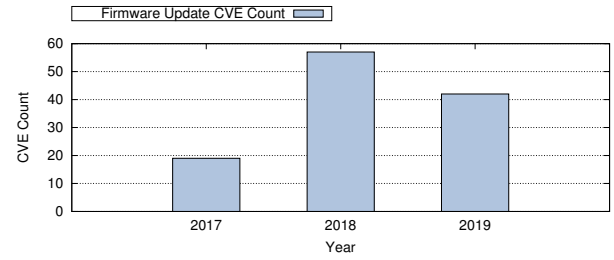
Unfortunately, firmware related vulnerabilities show no sign of abating as this component continues to evolve to promote features such as remote access and over the internet updates. Such features could be harnessed by cybercriminals to enable sensitive data collection across a multitude of platforms that span mobile devices, computer systems, and network infrastructure. Figure 2 summarizes this trend by showing the number of firmware related Common Vulnerabilities and Exposures (CVE) reported by the national vulnerability database [25]. On average, 437 firmware CVEs are reported every year with 39 of such CVEs detailing potential exposure to malicious firmware updates. This trend highlights the increased susceptibility of systems to untrusted firmware and the importance of understanding the impact of such risks.

2.4 The Case for HTTP Attacks

An indispensable technology that pre-enables the delivery of cloud-based services is the Hypertext Transfer Protocol (HTTP). HTTP supports two primary mechanisms for issuing requests: `GET` and `POST`. Data transferred through the `GET` method is accomplished by embedding the information directly within the Uniform Resource Locator (URL). For example, `http://www.mypage.com/form.php?name1=value1&name2=value2` can be used to send input `value1` and `value2`



((a)) Overall Firmware



((b)) Firmware Update

Figure 2: Number of Common Vulnerabilities and Exposures (CVE) per year for (a) all firmware vulnerabilities and (b) vulnerabilities only affecting firmware update.

for form fields `name1` and `name2` to `www.mypage.com`. On the other hand, transferring data through `POST` is achieved through a body section that is appended to the header portion of a given HTTP request.

Service requests initiated by users are typically performed through web browsers and mobile apps using `POST`. Such transactions often involve sending login information over HTTP to allow service providers to authenticate their users. Although HTTP is employed for sending sensitive information such as login credentials, the protocol itself is not designed to provide confidentiality guarantees. Instead, confidentiality guarantees for application data sent over HTTP is accomplished through the Transport Layer Security (TLS) which is implemented between the application and transport layers. TLS is designed to make data exchanged between a client (browser/app) and a server (service provider) cryptographically secure (HTTPS). Unfortunately, this layered approach for securing data is susceptible to adversaries that can intercept sensitive HTTP data before it is encrypted by the underlying TLS layer.

In order to evaluate the feasibility of recovering sensitive data, we characterized two popular services, namely Facebook and Instagram. In the first experiment, we focused on collecting Facebook usernames and passwords after logging into the service through a Chrome web browser that ran on an Ubuntu 18.04 system configured with 4 CPU cores and 8 GB of memory. Upon logging into Facebook, we collected a snapshot of the memory subsystem and analyzed its contents.

During this process, we discovered different occurrences of our Facebook login credentials in memory. For instance, we were able to locate the password embedded in the relevant form fields of the actual Facebook page that was rendered by the local browser. In addition, we found the HTTP request associated with sending our Facebook username and password information over the network. A sample of this HTTP request is shown in Figure 3(a). We observed that the body of the POST request collected from memory consisted of the `email` and `pass` parameters `xyz@gmail.com` and `pass123`, but presented in URL encoding instead.

The second experiment entailed using Instagram. However, this experiment focused on collecting the credentials for this service after using Instagram's mobile app. We downloaded the Instagram app onto an Android Oreo 8.1 system that ran on a system configured with 4 CPU cores and 4GB of memory. Similar to the Facebook experiment, we collected a snapshot of the memory subsystem and analyzed the contents after we had logged into the app. We observed multiple occurrences of our Instagram password in formats that conform to a JavaScript Object Notation (JSON). In addition, we were able to locate the HTTP POST request that was used to send out the authentication information using the JSON format. This information is illustrated in Figure 3(b).

The aforementioned Facebook and Instagram experiments demonstrate the feasibility of extracting authentication information from HTTP requests that are issued through either web browsers or mobile apps. Similar experiments show that we were able to successfully collect usernames and passwords across other popular services such as Gmail, Twitter, Facebook Messenger, and LinkedIn. Such findings underscore the potential for maliciously leveraging system firmware to covertly parse HTTP headers and extract secret data.

3 Threat Model

This study assumes a threat model that is consistent with prior work on firmware attacks. More specifically, we assume the attacker has the ability to install malicious firmware onto a system. A large body of work has demonstrated the ability to inject malicious firmware into a system [18–24]. For example, [21] demonstrated how to update the firmware directly from Windows by exploiting vulnerabilities in the firmware update process. Furthermore, an attacker could compromise the supply chain of the system manufacturer and in turn leverage the manufacturer's live update utility to push malicious firmware across a large number of systems as in the case of Shadow Hammer with ASUS systems [3] which led to 57,000 users having a backdoored version of the live update utility. An attacker could also use spear phishing techniques as was recently done with LoJax [26], a malicious application that runs code that infects the platform's firmware.

```
Host: www.facebook.com
Method: POST
Path: /login/device-based/regular/login
Content-Type:
application/x-www-form-urlencoded

jazoest=2697&lsd=AVrRLRxH&email=xyz%40gm
ail.com&pass=pass123&timezone=300&lgndim=
eyJ3IjoyNTYwLCJoIjoxNDQwLCJhdyI6MjU2MCw
```

((a)) Facebook

```
Host: i.instagram.com
Method: POST
Path: /api/v1/accounts/login
Content-Type: application/json

{"phone_id":"2ecdfcef-30d8-4678-a771-424d
2161f602","username":"xyz@gmail.com","de
vice_id":"android-f74b7d8d404d1cbe",
"password":"pass123","login_attempt_coun
```

((b)) Instagram

Figure 3: Login patterns obtained from HTTP authentication requests for (a) Facebook web service and (b) Instagram Android app.

4 Attack Prototype

Our attack is designed to explore the security risks associated with modern firmware and their impact on computing environments that range from mobile devices to computer systems. To this end, we devise a mechanism that transparently undermines the confidentiality guarantees provided by the upper layers and devise a framework that can systematically collect authentication information from HTTP requests without impacting normal execution. We developed system firmware using the Universal Extensible Framework Interface (`UEFI`). An overview of our prototype's execution flow is outlined in Figure 4. We use a daisy-chained multicore approach for parsing HTTP requests present in memory. This approach allows for efficient streamlining of the search process across multiple cores. It also minimizes the number of cycles each core is taken away from the user.

The execution flow starts by invoking the lowest numbered core (core 0) into firmware through a periodic management interrupt. The periodic interrupt is configured in hardware through our firmware during the platform initialization phase and prior to the OS taking ownership of the system. This is depicted as step ❶ in Figure 4. During this phase, the core consumes a set of predefined rules that it uses for parsing HTTP requests present in memory. HTTP requests are located by parsing each page for a pattern that begins with a valid request-line of an HTTP header (starts with "POST" and ends with CR + LF). The body of the request is then searched for formats that use either key/value form fields or JSON. As a proof of concept, the parameters of the body are parsed against keywords such as username, email, and password.

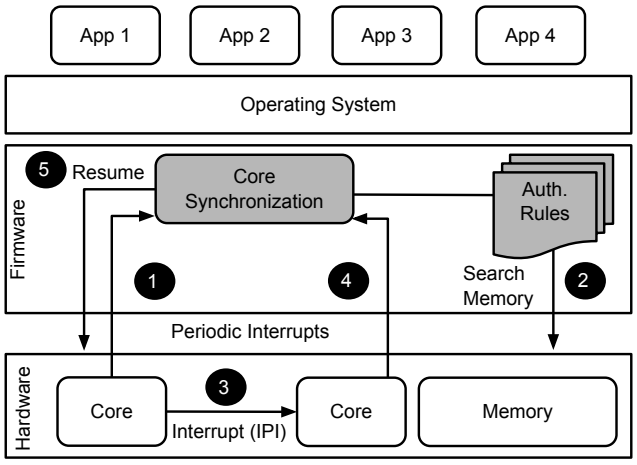


Figure 4: Main components of our firmware prototype for extracting user credentials from HTTP requests.

This is shown as step 2. Once step 2 is started, the core initiates its search for possible user credentials based on the rules it previously consumed. The core continues parsing memory until its time slice expires. When this occurs, the core halts the search and issues an inter-processor interrupt (IPI) to the next core (core 1) that will be responsible for resuming the search (step 3). However, before this newly invoked core (core 1) resumes the search, both cores (cores 0 and 1) undergo a synchronization phase to share information. This includes sharing the address of the last memory location that was parsed by the exiting core (core 0), a pointer to the last PTE that was referenced (both PTE and PDT information), as well as, a pointer to the next data structure that should be used for storing recovered credentials (step 4). Once the synchronization phase is complete, the previous core (core 0) is relinquished by firmware to resume normal execution while the newly invoked core (core 1) resumes the search (step 5). This process of having cores resuming memory searches followed by IPIs to the next available core, continues until the last core in the system is invoked into firmware. Once the last core completes its time slice, system firmware becomes inactive and awaits the next periodic management interrupt to occur. The execution flow from Figure 4 restarts again with step 1 once hardware issues the next periodic management interrupt.

A challenge with re-purposing manageability cycles for extracting user credentials relates to performance. Full memory scans require time to complete and varies as a function of memory capacity. This can lead to user applications to malfunction or become unresponsive. To reduce the performance impact on the overall system, our attack relies on walking the OS's page tables and examining the page table entries before scanning any pages. This process is illustrated in Figure 5. Whenever the initial core (core 0) receives the very first management interrupt, it copies the physical address of the page tables from the CPU register and begins walking the structures. For every encountered PTE, the firmware examines the PTE flags to determine if the page should be parsed. The firmware only parses a given page if its dirty, user/supervisor, and write flags (D/U/W) are set. Otherwise, the page is skipped. This approach greatly reduces the number of manageability cycles used by firmware and makes the design dependent on the number of launched apps instead of the memory capacity of the system.

Furthermore, our design makes use of memory that is available through runtime services to save collected data. Unlike memory allocated through boot time services, the OS doesn't reclaim memory regions that are reserved through runtime services, and therefore, remain available to firmware after the OS has taken ownership of the system. We utilize this memory for buffering recovered passwords from HTTP requests before they are saved to non-volatile memory. The saved data can then be ex-filtrated to a command-and-control (C&C) server through firmware's own network stack during a reboot of the platform. To enable this, we include the UEFI network stack within our firmware and dispatch it during boot to enable communication with a C&C server.

Although our attack relies on reboots for ex-filitrating data, we discuss how our design could be extended to ex-filtrate data while the OS is running. This approach, however, requires additional support across firmware's boot time and runtime phases. The aforementioned entails dispatching the network stack as a runtime module during boot and strictly consuming memory available through runtime services. This ensures that any resources that are consumed by the network stack are not reclaimed after control is relinquished to the OS. Sending data over the network while the OS is running requires an additional step. Specifically, firmware must invoke all CPU cores from the OS to avoid any contention over the network device. This can be achieved by having the master core that receives the management interrupt generate an IPI to the remaining cores in order to synchronize them within firmware. Data can be sent over the network once the cores have been synchronized, then relinquished back to the OS

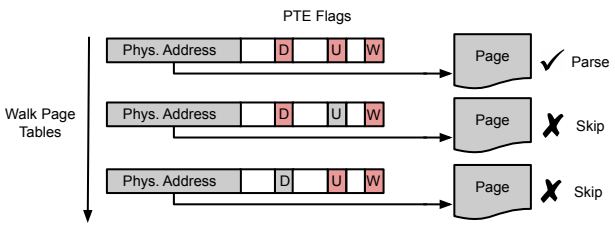


Figure 5: Example of firmware-based attack selecting pages for parsing according to their page table entry (PTE) flags. Pages that do not have their dirty (**D**), user/supervisor (**U**), and write (**W**) flags set are skipped to improve performance.

after the data has been transmitted. This process can repeat as needed over a predefined period (e.g. once every 24 hours).

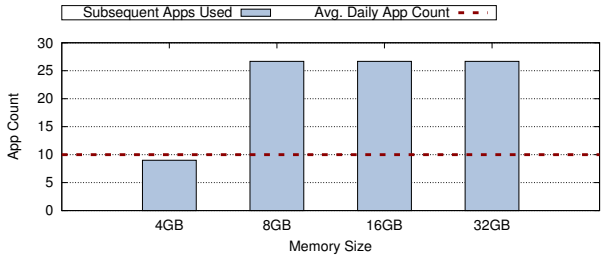
5 Evaluation

5.1 Experimental Framework

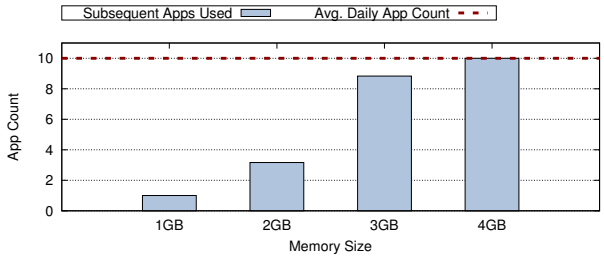
We conducted experiments using several applications across desktop and mobile platforms. To ensure test coverage of our solution under different application requirements, we used a diverse set of applications from six categories: social, communication, productivity, travel & local, health & fitness, and entertainment. The aforementioned categories and their corresponding applications are summarized in Table 1. In order to test our desktop system, we downloaded and installed commonly used applications for each category including productivity programs such as Office suite, Android Studio, and Eclipse that were configured to run on Linux. In addition, we launched common web services such as YouTube, Google Maps, and TripAdvisor through Google Chrome on the same platform. In the case of our mobile system, we downloaded several Android apps from Google Play store. Since computer systems are used in the context of productivity, we tested our desktop environment (Ubuntu) against more productivity applications compared to the Android apps used with our mobile platform (Android). We used a synthetic workload from [27] to serve as a stress application across both desktop and mobile platforms. Each stress application consisted of memory allocations that were used to increase the pressure on the memory subsystem and induce low memory conditions.

We used Ubuntu 18.04 LTS for running our linux-based desktop system and the Android Open Source Project (AOSP) 8.1 release for running Android. We installed `Termux` with `BusyBox` version 1.22.1 on Android in order to launch the stress apps from [27]. We used the `QEMU` 3.0.50 release for building our desktop and mobile platforms. This allowed us to test multiple hardware configurations as part of characterizing the robustness of our attack. In addition, we created a test harness using `Python` 3.6 that collected and analyzed memory snapshots through `QEMU`'s debug monitor at different intervals. These snapshots were used for analyzing the outgoing HTTP requests and constructing the username and password rules for different web-based services and Android apps. Examples of such patterns are shown in Figure 3. The test harness was also responsible for generating management interrupts to invoke firmware using different interrupt rates.

We developed system firmware using the Universal Extensible Framework Interface (`UEFI`) that we built from the Tianocore open source project. Finally, we used the `gem5` simulator [28] to collect cycle accurate information and analyze the performance overhead of our attack. We modeled a 3.7GHz x86-64 processor and 2.1GHz ARMv8 processor with DDR4 memory. The parameters of the hardware configurations we modeled are summarized in Table 2.



((a)) Desktop



((b)) Mobile

Figure 6: Summary of credential information persistence as a function of newly launched apps after user login.

5.2 Persistence vs. Application Usage

We first tested our firmware's ability to extract authentication credentials at runtime from memory under different app usage conditions. To this end, we enabled our firmware with a set of rules based on HTTP requests, Facebook's web service, and Instagram's Android app issue for authenticating their users. We analyzed several memory dumps of the running services and determined a reliable set of rules that could be used. We then characterized the ability of our firmware to retrieve Facebook's credentials on a desktop platform after launching 50 applications. We also conducted a similar experiment with Instagram after launching 20 apps on a mobile platform. The categories used in this experiment are listed in Table 1.

Characterization entailed constructing six app mixes according to Table 3 to simulate the persistence of login credentials under different app use cases. For each test instance, we logged into Facebook/Instagram (desktop/mobile), launched a different sequence of mixes, and then assessed the persistence of the login information after each app launched from a given mix while using our firmware. The experiments consisted of dividing 50 apps (desktop) and 20 apps (mobile) into six app mixes where each app mix consisted of all six categories that were run in a round robin fashion. The order of the apps launched in each category conformed to the same order listed in Table 1. Apps in each mix were used to generate system activity through actions, such as streaming videos, playing games, opening files, getting directions to an address, and reading emails for a duration of 30 seconds.

Figure 6 shows the results of the persistence experiment across our desktop and mobile platforms under different mem-

Category	Desktop Applications (Ubuntu)	Mobile Applications (Android)
Social	*Corebird (Twitter), Reddit, LinkedIn, Pinterest, Ramme (Instagram), Tumblr, Nextdoor, Wattpad*	*Facebook, Twitter, Linkedin, Pinterist*
Communication	*Slack, Skype, Signal, Whatsdesk (WhatsApp), Discord, Viber*	*WhatsApp, Facebook Messenger, Google Chrome*
Productivity	*Calc (Excel), Impress (Power Point), Writer (Word), Draw (Visio), Gimp, PDF, Overleaf, Gmail, Thunderbird, Calendar, Dropbox, Box, Peek (Screen Recorder), Everpad (Evernote), Android Studio, GitKraken, Eclipse, VirtualBox, Toggl, Qualtrics*	*Gmail, Dropbox, Calendar, Todoist*
Travel & Local	*Airbnb, Google Maps, TripAdvisor, Expedia Travel, Uber, Yelp, Lyft, Grubhub*	*Google Maps, TripAdvisor, Uber, Yelp*
Health & Fitness	*WebMD, LiveStrong, MyFitnessPal*	*Google Fit, MyFitnessPal*
Entertainment	*YouTube, Angry Birds, Candy Crush, Spotify, Steam*	*Bitmoji, YouTube, Wordscapes*

Table 1: Summary of desktop (Ubuntu) and mobile (Android) applications tested for each category.

ory configurations and app mixes. A study from [29] suggests that the number of daily apps used by consumers is 10 with such consumers spending over 50% of their digital time on

Hardware Configuration	
Cores	4
ISA	x86-64, ARMv8 (64-bit)
Frequency	3.7GHz (x86-64), 2.1GHz (ARMv8)
IL1/DL1 Size	32KB
IL1/DL1 Block Size	64B
IL1/DL1 Associativity	8-way
IL1/DL1 Latency	2 cycles
Coherence Protocol	MESI
L2 Size	2MB
L2 Block Size	64B
L2 Associativity	16-way
L2 Latency	20 cycles
Memory Type	DDR4-2400 SDRAM
Memory Size	1GB, 2GB, 3GB, 4GB, 8GB, 16GB, 32GB

Table 2: Summary of hardware configurations.

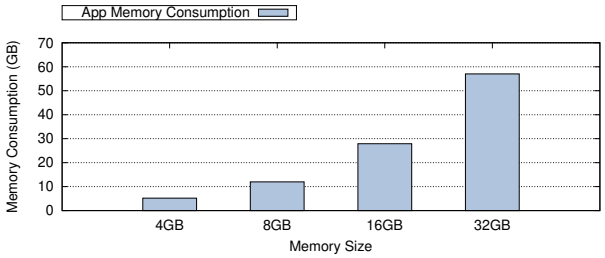
Mix	Category Sequence
1	*social, communication, productivity, travel & local, health & fitness, entertainment*
2	*communication, productivity, travel & local, health & fitness, entertainment, social*
3	*productivity, travel & local, health & fitness, entertainment, social, communication*
4	*travel & local, health & fitness, entertainment, social, communication, productivity*
5	*health & fitness,entertainment, social, communication, productivity, travel & local,*
6	*entertainment, social, communication, productivity, travel & local, health & fitness*

Table 3: Summary of application mixes used for credential persistence testing.

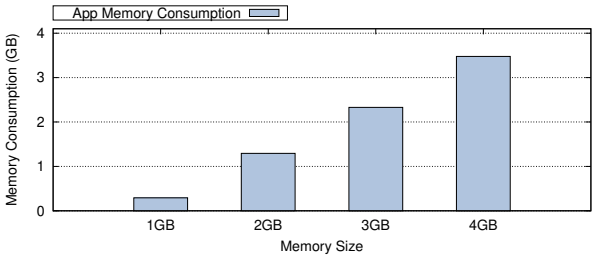
mobile devices [30]. As such, our study assumes a baseline of 10 daily apps per platform for the average user.

On our desktop platform, we observed that the number of apps that could be launched while still having the ability to retrieve credentials ranged between 4 to 49 apps. On average, our firmware could locate login credentials after utilizing 22.3 apps. Figure 6(a) summarizes the persistence of credential information as a function of launched apps averaged across the six mixes from Table 3. Overall, the vulnerability factor (finding credentials) increased by 3x when doubling the memory size from 4GB to 8GB. However, this vulnerability factor plateaued once the memory capacity reached 8GB. For instance, systems with 8GB, 16GB, and 32GB memory capacities had the same average, tolerating 26.7 apps before the login credentials could no longer be found. A closer look at these results revealed that the gaming app, *Steam*, consistently caused our Facebook credentials to be evicted. This behavior was consistent across all the app mixes listed in Table 3. As a result, we conducted an additional experiment using our stress app to better characterize the vulnerability factor as a function of memory capacity. The results of this experiment revealed a strong correlation between memory capacity and the number of apps that could be used before credential eviction. We successfully located our password after launching 120, 279, and 570 apps on platforms configured with 8GB, 16GB, and 32GB, respectively. Even though applications, such as *Steam* can result in passwords being evicted, our overall attack is robust in desktop environments and can reliably retrieve credentials after running a variety of application mixes.

Figure 6(b) illustrates the results of the persistence experiment on our mobile platform. Overall, our attack performs the best against larger systems that have 3GB and 4GB memory capacities. We observed that the number of apps that could be used while still being able to retrieve Instagram's credentials ranged between 1 to 12 apps depending on the memory size. On average, our firmware could locate login credentials after utilizing 5.7 apps. We observed a strong correlation between the device's memory capacity and the number of apps used. This suggests that reasonably modern smartphones from 2015 that have larger capacities, such as Samsung Galaxy S6 and



((a)) Desktop



((b)) Mobile

Figure 7: Summary of credential information persistence as a function of memory consumption.

HTC M9 are more prone to this attack. For instance, we observed that for a 1GB configuration, we could only retrieve the login credentials after 1 app has been used, with the password consistently deleted after a second app is launched. Further analysis shows that the vulnerability factor for a 2GB configuration increases to 3.2x relative to a 1GB configuration. The 2GB system allows for user credentials to still be detected after 2 to 5 apps have been used. The attack performs the best on the 4GB system showing a 10x increase in the relative vulnerability factor. The 4GB system allows for the login information to remain in memory after 7 to 12 apps have been used which is in line with the average number of daily apps consumers use. This is slightly higher than the 3GB configuration which remains vulnerable after using 8.8 apps.

5.3 Persistence vs. Memory Consumption

In order to evaluate the persistence of authentication information under different memory consumption profiles, we tested our attack with a configurable stress app. Using a stress app from [27] allowed us to systematically vary the amount of memory consumed in the system. Each experiment entailed first logging into Facebook/Instagram (Desktop/Mobile) and then launching the stress app. The memory footprint of the stress app was gradually increased in 100MB increments every minute. We then tested for the presence of the user's credentials through firmware after every memory increment. We also ran multiple apps in the background including WhatsApp and Gmail to further stress the memory subsystem.

Figure 7(a) shows the results of the memory consumption experiment for different memory capacities on our desktop platform. Although the number of launched apps has an impact on the persistence of credential information in memory, the persistence of such information strongly depends on the amount of memory a given app consumes. On average, we observed that for the 4GB system, the credentials remained in memory until the stress app consumed 1.2GB beyond the physical memory capacity. Evictions occurred after an additional 50%, 75%, and 78% of memory was consumed beyond the total physical memory of the system when configured with 8GB, 16GB, and 32GB, respectively. In other words, a significant amount of swap space had to be consumed before the credentials were evicted. Figure 8 illustrates this trend by showing the probability of finding credentials as a function of memory activity during our stress experiment on a 4GB system. The credentials on our desktop system remained in memory for 104 mins. Firmware failed to find the credentials after this 104 min duration which occurred after 900K pages have been evicted, as shown in Figure 8(a). We also observed that the system goes through different page-out rates. This is illustrated in Figure 8(c). We can see that at the beginning a steady page-out rate of about 3 pages per second occurred. However, a more aggressive page-out rate was observed before the credentials could no longer be found in memory.

Figure 7(b) shows the results of the memory consumption experiment for our mobile platform. Although the number of launched apps has an impact on the persistence of credential information in memory, the persistence of such information strongly depends on the amount of memory a given app consumes. On average, we observed that for the 1GB system, the credentials remained in memory when the stress app consumed less than 300MB. On average, this threshold increased to 1.3GB for a 2GB capacity, 2.3GB for a 3GB capacity, and 3.5GB for a 4GB capacity before the credentials were evicted. In general, Android's MMU played a significant role in the retention of credentials with the MMU's allocation policy varying as a function of memory capacity. The MMU allocated memory more aggressively for background apps as the capacity was increased from 1GB to 4GB. Furthermore, Android doesn't use swap space. As such, background apps are terminated more frequently and the associated pages are freed. Unlike the desktop case, we can see in Figure 7(b) that password evictions occurred before the amount of physical memory was exhausted. Moreover, Figure 8 illustrates the probability of finding Instagram's credentials as a function of memory activity during our stress experiment on a 4GB system. We observed that the credentials remain in memory for 74 mins. Firmware failed to find the credentials after this 74 min duration which occurred after 68K pages were evicted from memory. We found that Android frees 70K pages within 83 minutes compared to 310K pages in a desktop environment. Although the page-out rate in a desktop environment is 4.4x higher than what we observed for Android's page-out rate, password evictions occurred much sooner on an Android

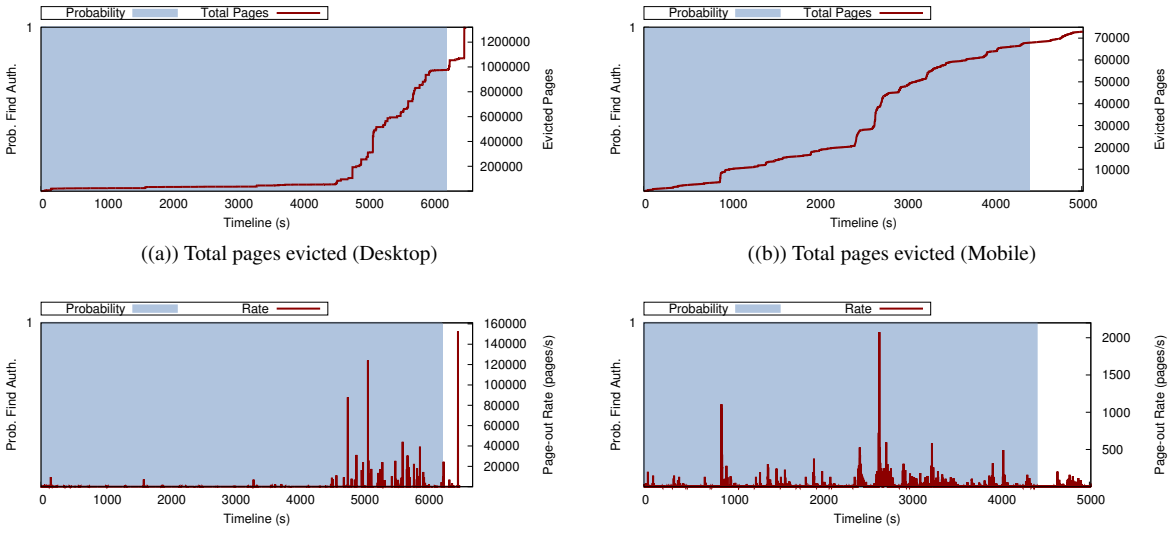


Figure 8: Probability of finding authentication credentials over time as a function of the total number of pages evicted (a) desktop (b) mobile and the page-out rate in pages per second for (c) desktop and (d) mobile.

platform. Figure 8(d) summarizes Android's page-out rate over time. Unlike desktop systems, faster password evictions occurred in Android. We observed that Android initially freed pages at a relatively steady rate that is 3x higher compared to what we observed for a desktop system when excluding abruptly high page-out rates that are in excess of 1000 pages/s.

5.4 Performance Overhead

Our firmware-based attack is designed to selectively examine a limited number of pages in order to covertly extract sensitive data from HTTP requests without disrupting the normal execution of the system. Therefore, to better understand the suitability of our solution for desktops and mobile devices, we conducted runtime experiments across different applications and platform configurations. In this section, we examine the overhead of our solution compared to a solution that parses HTTP requests through full memory scans.

Figure 9 shows the number of searched pages as a function of launched applications and their respective runtimes. Since our previous experiments show that login credentials remain in memory well beyond 1 hour, we set up our management interrupts to ensure the completion of a full memory search cycle over a one hour period. Figures 9(a) and 9(b) summarize the number of user pages as a function of launched applications across our desktop and mobile platforms, respectively. We observe that the total number of pages that have their dirty, user/supervisor, and write flags set (D/U/W), varies as a function of launched apps and platform type. In general, our mobile platform running Android had 1.9x more D/U/W pages compared to an Ubuntu-based desktop platform. On average, our firmware scanned 380 and 725 D/U/W pages for every desktop and mobile application, respectively. This correlates to each core spending a total of 180 µs – 3.1 ms every hour (3 µs – 52 µs per minute) scanning memory when running 10 to 30 desktop apps and between 3.3 ms – 5.1 ms every hour (55 µs – 85 µs per minute) when running 10 to 30 mobile apps. The Ubuntu-based desktop overhead is relatively lower than the Android-based mobile overhead due to the reduced number of PTEs that have their D/U/W flags set. We also observe that by limiting our search to only D/U/W pages instead of all user pages, we eliminate scanning an additional 48K pages which corresponds to a 4.7x performance improvement, on average. Finally, we didn't observe a significant difference between user pages that are writable and those that have become dirty. On average, we recorded across our tests 128 pages that were writable, but not dirty. While this figure is dependent on the application type, scanning pages that have their D/U/W flags as opposed to U/W presents another level of optimization for reducing the overhead of our attack.

Figure 10 illustrates the performance of an attack that uses a full memory scan approach and how it compares to our optimized solution. We measure the performance across different memory capacities, launched applications, and platforms. Figure 10(a) shows that the overhead increased linearly as a function of memory capacity when performing a full search. On our desktop system each core spent 23 s and 180 s scanning memory in order to complete a full search on 4GB and 32GB memory configurations, respectively. As such, this naïve approach doesn't scale well to larger memory capacities. On the other hand, we observe that our page table-based attack sig-

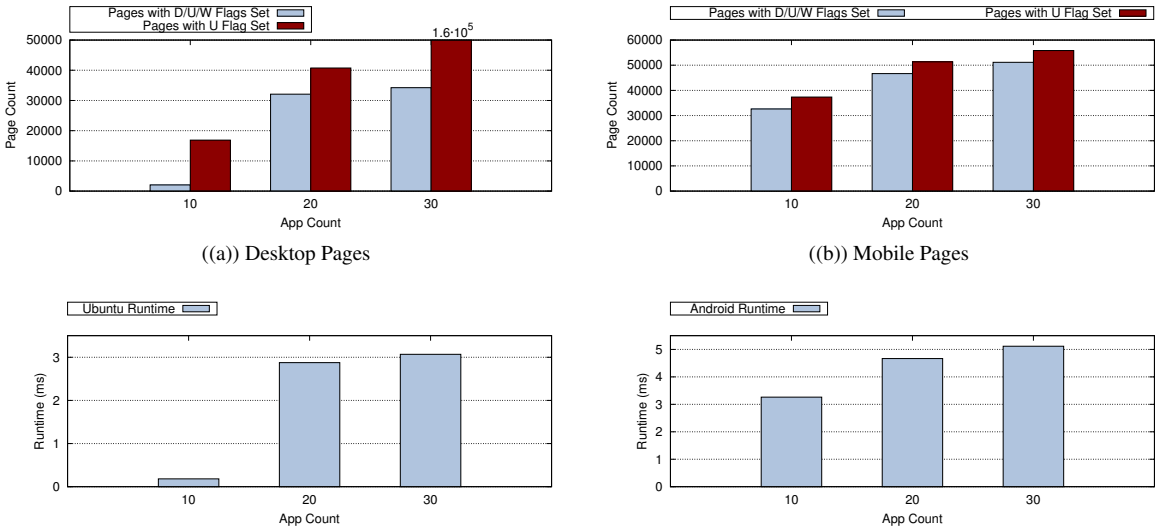


Figure 9: The number of searched pages as a function of launched applications (a) Desktop (Ubuntu), (b) Mobile (Android), and the runtime as a function of launched applications (c) Desktop (Ubuntu) and (d) Mobile (Android).

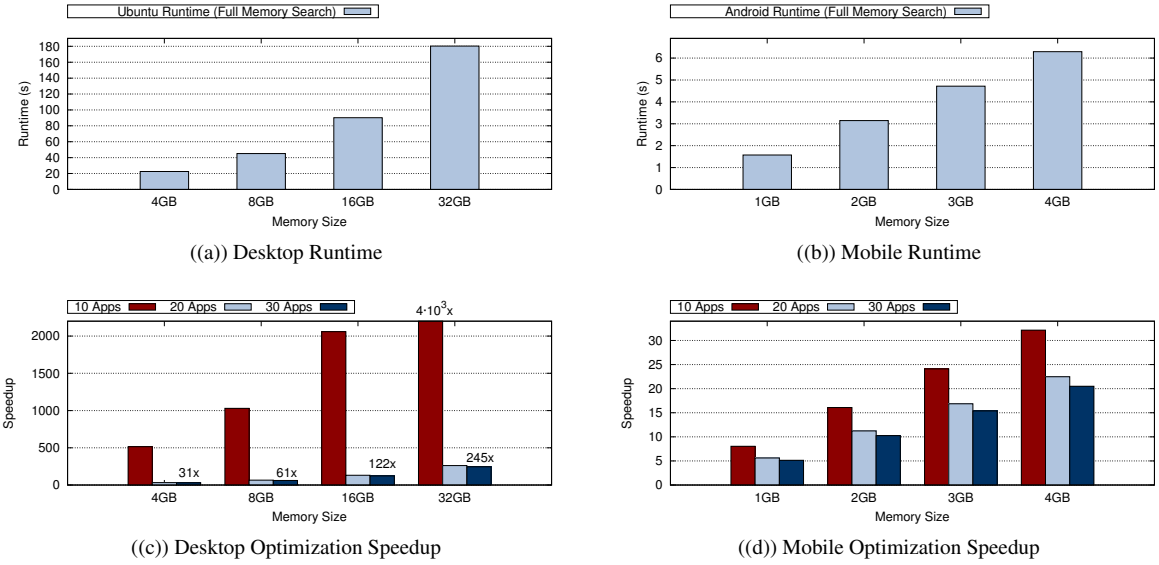


Figure 10: Runtime of full memory search as a function of memory capacity (a) Desktop (Ubuntu), (b) Mobile (Android), and speedup of our attack relative to a full memory search as a function of launched apps (c) Desktop (Ubuntu), (d) Mobile (Android).

nificantly improves the search time relative to a full memory search approach. We observe a relative speedup that ranges between 514x and $4.1 \cdot 10^3$x when running 10 apps across memory capacities 4GB – 32GB on a desktop. The speedup was reduced to 31x and 245x when running 30 apps across memory capacities 4GB – 32GB on the same platform. This reduction was due to an increase in the number of candidate pages our firmware scanned with 30 apps running on a desktop. This is summarized in Figure 10(c). Our results suggest a similar trend for mobile systems. This is shown in Figures 10(b) and 10(d). On our mobile system, each core spent 1.6 s and 6.3 s in order to complete a full search on 1GB and 4GB memory configurations, respectively. On the other hand, the speedup while using our optimization ranged between 8x –

32x when running 10 apps across memory capacities 1GB – 4GB and a speedup of 5x – 20x when running 30 apps across memory capacities 1GB – 4GB. Overall, our results show that our attack is suitable across mobile and desktop systems and shows that the overhead only increases as more applications are launched, irrespective of the memory capacity. The overheads of our optimized solution are shown in Figures 9(c) and 9(d).

5.5 Algorithm Robustness

We conducted several hours of testing on our firmware. The attack ran reliably and consistently under multiple workloads, load levels, and platforms. One of the stress experiments we conducted entailed evaluating the reliability of our firmware algorithm while searching memory in management mode. The experiment included continuously invoking firmware to search memory every one minute over a period of one week while we monitored the system for any crashes. The aforementioned steps were carried out on 8GB and 4GB systems that were already booted to the Ubuntu and Android OS's, respectively. The experiment also included interacting with each booted OS every one minute over an SSH connection. Both systems ran successfully without any crashes until the experiment was stopped.

6 Discussion

In this section, we discuss a possible countermeasure against firmware attacks. Since firmware operates at the lowest layer of the software stack, we consider a hybrid approach that entails the OS and hardware layers cooperatively restricting the processor from accessing non-firmware owned memory. To achieve this, the OS must register the memory ranges it will manage during its boot process, leaving out any regions already claimed by firmware. To support this, the hardware could expose a write-once table that we will refer to as the memory protection table (MPT).

During boot, the OS writes the memory ranges it owns into this table and validates that the intended ranges were properly programmed into the MPT. The OS raises a warning to the user in the event that it is unable to program the table as a possible indication that firmware has already programmed the ranges to purposely overlap with the OS's memory. Furthermore, any memory accesses issued to the ranges present in the MPT while the processor is executing firmware will result in an exception.

A challenge with the aforementioned approach is that the OS often relies on firmware to perform legitimate platform tasks on its behalf through runtime services. Since runtime services are in the form of code that belongs to firmware, it is conceivable that such services could be leveraged to collect secret information from memory. As a result, the OS must program the MPT with ranges that are owned exclusively by the OS, excluding any ranges that are shared between the firmware and OS layers. This is because runtime services often require data to be exchanged with the OS through shared regions. Therefore, to support the countermeasure without breaking any legitimate runtime service functionality executed by firmware, we propose augmenting the processor's load/store queue (LSQ), the entity responsible for issuing memory transactions. In this case, the LSQ would serve the purpose of validating the program counter (PC) corresponding to the load/store instruction being executed and the address of the memory being accessed. To avoid triggering exceptions due to speculative instructions that may be squashed and never visible outside the processor, we propose waiting until a given entry within the LSQ is selected for retirement. At this point, the countermeasure checks the PC and the address of the memory transaction against the MPT containing the ranges originally programmed by the OS. An exception is raised if the PC corresponding to the load/store instruction isn't found in the MPT (not within the OS's exclusively owned range) and the address for the memory transaction is within the MPT (within the OS's exclusively owned range).

Detecting firmware accesses to memory through load/stores while the processor is in management mode would follow a similar approach to that of dealing with runtime services. When a management interrupt is issued to the processor, the mode will be set to reflect that it is in management mode. However, the same solution applies. The code executed in management mode belongs to firmware and as such would fall outside of the range programmed into the MPT. To this end, an exception would be raised in the event that the PC corresponding to the load/store instruction isn't found in the MPT and the address for the memory access is found within the MPT.

7 Related Work

Our related work is divided into: (1) prior work that relates to our attack, namely memory attacks and (2) prior work against the firmware subsystem that our proposed attack builds upon.

Memory Attacks. For many years, system memory has been a prime target for stealing secret information [31–35]. In response to these challenges, researchers have explored various attacks that aim to expose the security risks associated with this subsystem [4–13]. Prior work [4] demonstrated the use of introspection through hypervisors for extracting credit card data from Point of Sale (PoS) systems that run as guest machines. However, such attacks are limited to environments that rely on hypervisors. Firmware attacks, on the other hand, are more dangerous, since firmware exists on every computing device. In addition, unlike hypervisors, firmware doesn't have the ability to trap memory accesses. As a result, our study explores a different set of challenges associated with covertly recovering data belonging to the application layer.

Other work examined the risks of memory scraping through

physical access [6–13]. For example, Halderman et al. [6] proposed using cold boot attacks to recover sensitive data from memory. The authors showed that a cooling agent could be applied to preserve the content of memory modules extracted from a stolen laptop, then later scanned on another machine to recover cryptographic keys. Muller et al. [36] proposed a similar approach against Android smartphones that could recover photos, personal messages, and passwords, in addition to cryptographic keys. Re-purposing available hardware ports is another technique that was used to collect memory dumps from smartphones. For instance, Munro [8] showed that the JTAG port on smartphones could be used to communicate directly with the system-on-chip and read out memory without the use of a cooling agent. Other attacks [9–13], considered installing malicious peripherals for reading arbitrary memory regions through DMA attacks. Virtually, all of these attacks require physical access or the presence of special hardware on the system. Unlike the aforementioned work, our attack is firmware-based and can be carried out remotely through software supply chain attacks or compromised live update utilities [1–3].

Firmware Attacks. Different forms of attacks against firmware have been explored [18–24, 37–53]. This includes crafted remote updates issued from user space to bypass safety measures such as secure boot [18–24]. For example, Kallenberg and Wojczuk [19] demonstrated the ability to overcome flash write-protection mechanisms and overwrite firmware at runtime by exploiting a race condition within the manageability chip. Other work [24], explored leveraging the presence of unused reference code that could be activated through integer overflows to initiate malicious firmware updates. Additional techniques include bypassing secure boot features by exploiting unsigned fragments such as company logo placeholders that are designated for OEM customization. Matrosov and Rodionov [54] demonstrated that malicious firmware could be launched in the presence of security features such as secure boot. Our work leverages the aforementioned attacks to insert untrusted firmware into the system and explore the risks associated with such malware. More importantly, we examine how such a component could be re-purposed to undermine the confidentiality guarantees the system provides to the application layer in the presence of such untrusted firmware.

8 Conclusion

In this work, we propose a novel firmware attack that leverages platform management cycles to extract sensitive data from HTTP requests. We develop a proof-of-concept of our attack using firmware that was configured to run with both Ubuntu 18.04 LTS and the Android Oreo 8.1. We show that a page table-based approach is sufficient to efficiently extract sensitive data from outgoing HTTP requests without disrupting the normal execution of launched apps. Our approach is up to $4 \cdot 10^3$x faster compared to a full memory search implementation. Our attack claims limited execution cycles from the application layer making the attack difficult to detect even in mobile environments where app responsiveness is closely by consumers. Finally, we discuss countermeasures that could be employed to defend against such attacks.

Acknowledgements

The authors would like to thank the anonymous reviewers for their constructive feedback and comments on this work. We extend special thanks to Yaohui Chen for shepherding our paper and working with us on producing the camera ready version of this paper. Finally, the authors would like to thank members of the Security and Systems Lab for all their support and feedback related to this work.

References

[1] A. Greenberg, "Software has a serious supply-chain security problem," 2017, https://www.wired.com/story/ccleaner-malware-supply-chain-software-security.

[2] P. Gralla, "Malware takes aim at the supply chain," 2018, https://symantec-blogs.broadcom.com/blogs/expert-perspectives/malware-takes-aim-supply-chain.

[3] Kaspersky, "Operation shadowhammer," 2019, https://securelist.com/operation-shadowhammer.

[4] J. Hizver and T.-c. Chiueh, "An introspection-based memory scraper attack against virtualized point of sale systems," in *International Conference on Financial Cryptography and Data Security*. Springer, 2011, pp. 55–69.

[5] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[6] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryption Keys," in *USENIX Security Symposium*, July 2008, pp. 45–60.

[7] C. Hilgers, H. Macht, T. Müller, and M. Spreitzenbarth, "Post-mortem memory analysis of cold-booted android devices," in *2014 Eighth International Conference on IT Security Incident Management & IT Forensics*. IEEE, 2014, pp. 62–75.

[8] K. Munro, "Android scraping: accessing personal data on mobile devices," *Network Security*, vol. 2014, no. 11, pp. 5–9, 2014.

[9] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson, "Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals." in *NDSS*, 2019.

[10] U. Frisk, "Dma attacking over usb-c and thunderbolt 3," Blog, October 2016, http://blog.frizk.net/2016/10/dma-attacking-over-usb-c-and.html.

[11] ——, "Direct memory attack the kernel," *Proceedings of DEFCON*, vol. 24, 2016.

[12] J. FitzPatrick and M. Crabill, "Stupid pcie tricks," 2014.

[13] U. Frisk, "Pcileech: Direct memory access attack software," https://github.com/ufrisk/pcileech.

[14] A. Bacha and R. Teodorescu, "Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors," in *International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 297–307.

[15] ——, "Using ECC feedback to guide voltage speculation in low-voltage processors," in *International Symposium on Microarchitecture (MICRO)*, December 2014, pp. 297–307.

[16] ——, "Authenticache: Harnessing cache ECC for system authentication," in *International Symposium on Microarchitecture (MICRO)*, December 2015, pp. 1–12.

[17] S. Wall, "Resiliency for a cyber-physical future," 2017, https://www.slideshare.net.

[18] T. Hudson, X. Kovah, and C. Kallenberg, "Thunderstrike 2: Sith strike," in *Black Hat USA*, 2015.

[19] C. Kallenberg and R. Wojtczuk, "Speed racer: Exploiting an intel flash protection race condition," 2015, white Paper.

[20] R. Wojtczuk and C. Kallenberg, "Attacking UEFI boot script," in *Chaos Communication Congress*, 2014.

[21] C. Kallenberg, X. Kovah, J. Butterworth, and S. Cornwell, "Extreme privilege escalation on windows 8/uefi systems," in *Black Hat USA*, 2014.

[22] Y. Bulygin, A. Furtak, and O. Bazhaniuk, "A tale of one software bypass of windows 8 secure boot," in *Black Hat USA*, 2013.

[23] X. Kovah, C. Kallenberg, J. Butterworth, and S. Cornwell, "Senter sandman: Using intel TXT to attack bioses," in *Hack in the Box*, 2014.

[24] ——, "Bios necromancy: Utilizing "dead code" for bios attacks," in *Hack in the Box*, 2014.

[25] N. Institute of Standards and Technology, "National vulnerability database," 2019, https://nvd.nist.gov.

[26] T. Spring, "First-ever uefi rootkit tied to sednit apt," 2018, https://threatpost.com/uefi-rootkit-sednit/140420/.

[27] A. Waterland, "Stress," 2014, https://people.seas.harvard.edu/ apw/stress/.

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

[29] A. Annie, "Spotlight on consumer app usage," 2017, https://www.appannie.com.

[30] BusinessofApps, "App download and usage statistics," 2019, https://www.businessofapps.com/data/app-statistics.

[31] R. Inocencio, "New blackpos malware emerges in the wild, targets retail accounts," Online, August 2014, https://blog.trendmicro.com/trendlabs-security-intelligence/new-blackpos-malware-emerges-in-the-wild-targets-retail-accounts.

[32] L. Constantin, "Dexter malware infects point-of-sale systems worldwide, researchers say," Online, December 2012, https://www.csoonline.com/article/2132674/dexter-malware-infects-point-of-sale-systems-worldwide–researchers-say.html.

[33] D. Goodin, "Meet "chewbacca," the point-of-sale malware that infected dozens of retailers," Online, January 2014, https://arstechnica.com/information-technology/2014/01/meet-chewbacca-the-point-of-sale-malware-that-infected-dozens-of-retailers/.

[34] S. Gatlan, "Dmsniff point-of-sale malware silently attacked smbs for years," Online, February 2019, https://www.bleepingcomputer.com/news/security/dmsniff-point-of-sale-malware-silently-attacked-smbs-for-years.

[35] N. Huq, "Pos ram scraper malware: Past, present, and future," *A Trend Micro Research Paper*, pp. 1–16, 2014.

[36] T. Muller and M. Spreitzenbarth, "FROST: Forensic Recovery of Scrambled Telephones," in *International Conference on Applied Cryptography and Network Security*, June 2013, pp. 373–388.

[37] F. Cao, Q. Li, and Z. Chen, "Vulnerability model and evaluation of the uefi platform firmware based on improved attack graphs," in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 2018, pp. 225–231.

[38] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 367–378.

[39] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, 2016, pp. 1–16.

[40] J. Taylor, B. Turnbull, and G. Creech, "Volatile memory forensics acquisition efficacy: A comparative study towards analysing firmware-based rootkits," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ACM, 2018, p. 48.

[41] F. Zhang, H. Wang, K. Leach, and A. Stavrou, "A framework to secure peripherals at runtime," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 219–238.

[42] M. LeMay and C. A. Gunter, "Cumulative attestation kernels for embedded systems," *IEEE Transactions on Smart Grid*, vol. 3, no. 2, pp. 744–760, 2012.

[43] D. Schellekens, P. Tuyls, and B. Preneel, "Embedded trusted computing with authenticated non-volatile memory," in *International Conference on Trusted Computing*. Springer, 2008, pp. 60–74.

[44] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham, "Mouse trap: Exploiting firmware updates in {USB} peripherals," in *8th {USENIX} Workshop on Offensive Technologies ({WOOT} 14)*, 2014.

[45] Y. Li, J. M. McCune, and A. Perrig, "Viper: verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 3–16.

[46] D. Peck and D. Peterson, "Leveraging ethernet card vulnerabilities in field devices," in *SCADA security scientific symposium*, 2009, pp. 1–19.

[47] A. M. Garcia Jr, "Firmware modification analysis in programmable logic controllers," AIR FORCE INSTITUTE OF TECHNOLOGY WRIGHT-PATTERSON AFB OH GRADUATE SCHOOL OF . . . , Tech. Rep., 2014.

[48] K. Chen, "Reversing and exploiting an apple firmware update," *Black Hat*, vol. 69, 2009.

[49] S. Hanna, R. Rolles, A. Molina-Markham, P. Poosankam, J. Blocki, K. Fu, and D. Song, "Take two software updates and see me in the morning: The case for software security evaluations of medical devices." in *HealthSec*, 2011.

[50] C. Miller, "Battery firmware hacking," *Black Hat USA*, pp. 3–4, 2011.

[51] B. Jack, "Jackpotting automated teller machines redux," *Black Hat USA*, 2010.

[52] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno *et al.*, "Comprehensive experimental analyses of automotive attack surfaces." in *USENIX Security Symposium*, vol. 4. San Francisco, 2011, pp. 447–462.

[53] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 447–462.

[54] A. Matrosov and E. Rodionov, "Uefi firmware rootkits: Myths and reality," in *Black Hat Asia*, 2017.