BootKitty: A Stealthy Bootkit-Rootkit Against Modern Operating Systems

Junho Lee, *Mokpo National University;* Jihoon Kwon, *Korea University;* HyunA Seo, *Sungshin Women's University;* Myeongyeol Lee, *Chosun University;* Hyungyu Seo, *Keimyung University;* Jinho Jung, *Ministry of National Defense;* Hyungjoon Koo, *Sungkyunkwan University*

https://www.usenix.org/conference/woot25/presentation/lee

This paper is included in the Proceedings of the 19th USENIX WOOT Conference on Offensive Technologies.

August 11–12, 2025 • Seattle, WA, USA

ISBN 978-1-939133-50-2

BOOTKITTY: A Stealthy Bootkit-Rootkit Against Modern Operating Systems

Junho Lee
Mokpo National University

Jihoon Kwon
Korea University

HyunA Seo
Sungshin Women's University

Myeongyeol Lee
Chosun University

Hyungyu Seo
Keimyung University

Jinho Jung
Ministry of National Defense

Hyungjoon Koo
Sungkyunkwan University

Abstract

Bootkits and rootkits are among the most elusive and persistent forms of malware, subverting system defenses by operating at the lowest levels of system architecture. Bootkits compromise the firmware or bootloader, allowing them to manipulate the boot sequence and gain control before security mechanisms initialize. Meanwhile, rootkits embed themselves within the OS kernel, stealthily conceal malicious activities, and maintain long-term persistence. Despite their critical implications for security, these threats remain underexplored due to the technical complexity involved in their study, the scarcity of real-world samples, and the challenges posed by defense-in-depth security in modern OSes.

In this paper, we introduce BOOTKITTY, a hybrid bootkit-rootkit capable of circumventing modern security features in multiple OS platforms, across Windows, Linux, and Android. We explore critical firmware and bootloader vulnerabilities that can lead to a low-level compromise, demonstrating techniques that bypass advanced security protections by breaking the chain of trust. Our study addresses technical challenges such as exploiting UEFI drivers, manipulating kernel memory, and evading advanced mitigations in the boot process, and provides actionable insights. Our systematic evaluations show that BOOTKITTY reveals critical weaknesses in contemporary security mechanisms, highlighting the need for better security design that offers holistic (low-level) protection.

1 Introduction

Malware infiltration presents a persistent and evolving threat in modern computing environments. Among the most elusive forms of malware are bootkits and rootkits, which operate at the lowest levels of system architecture to evade detection, persist stealthily, and thwart conventional security mechanisms [24, 36, 47, 52, 61]. Bootkits embed themselves within firmware (*e.g.,* UEFI [106]) or the bootloader, allowing them to manipulate the boot sequence and execute before any security mechanisms are initialized. On the other hand, rootkits run themselves within the OS kernel, obscuring malicious activities by concealing processes, files, and network packets. Both classes of malware can remain undetected because of their privileged execution and deep system penetration, systematically subverting even defense-in-depth security for modern OSes. Conducted by advanced adversaries, such as Nation-State Actors and Advanced Persistent Threat (APT) groups, the objectives of these attacks range from data exfiltration and espionage to system sabotage. Besides, the stealthy nature and long-term exploitation (*i.e.,* survival after system reboots) of bootkits and rootkits make them particularly riskier than conventional malware.

Despite the evolving security mechanisms such as UEFI Secure Boot [104] on PCs and Verified Boot [10, 83] on mobile devices, bootkits and rootkits continue to exploit low-level vulnerabilities in firmware and OS kernel components to undermine these protections. Bootkits leverage firmware weaknesses to tamper with the boot process, undermining the system's foundational security. Rootkits evade kernel-level protections, including PatchGuard [62] in Windows and Kernel Lockdown [53] in Linux, allowing them to operate undetected. Hence, an in-depth understanding of such threats and infiltration routes can assist in designing holistic defense strategies and providing critical insights to harden modern systems.

Although these threats pose a severe risk, studying bootkits and rootkits remains limited for several reasons: ① **(Lack of Real-world Samples)** Unlike widespread malware campaigns, bootkits and rootkits are primarily used in targeted attacks [46, 84], limiting public research opportunities. ② **(Technical Complexity)** Studying these threats requires deep expertise in UEFI internals, OS kernel operations, driver interactions, memory management, and hardware initialization. ③ **(Advanced Protections and Debugging Challenges)** Modern OSes adopt secure bootchains, mandatory driver signing [54, 67], hypervisor-based monitoring [68], and memory encryption, rendering both the analyses and attack demonstrations more challenging. ④ **(Hardware and Firmware Variability)** Differences across hardware platforms and firmware implementations complicate the development of universal bootkits and rootkits, requiring custom exploits and extensive hardware-based testing.

These technical barriers explain why bootkits and rootkits receive less attention compared to more accessible threats like ransomware and botnets. Security teams often prioritize higher-level malware due to the steep learning curve, diverse firmware architectures, limited debugging tools, and the necessity of hardware-based testing (as well as limited resources). As a result, comprehensive and reproducible case studies on low-level intrusion remain scarce.

In this paper, we bridge these gaps by constructing and analyzing a hybrid bootkit-rootkit, dubbed BOOTKITTY, which demonstrates how firmware-level vulnerabilities can be exploited to compromise modern OSes across multiple platforms (*e.g.,* Windows, Linux, and Android). We identify and exploit boot-level vulnerabilities that allow early-boot code execution and chain these weaknesses to achieve OS-level compromise at the kernel layer. Next, we provide a detailed technical breakdown of the steps required to circumvent key security mechanisms, including UEFI Secure Boot (firmware protection), Kernel Lockdown [53] (kernel integrity), and Driver Signature Enforcement (DSE) [67] (kernel module verification). To this end, our experimental setup for instrumenting bootkits and rootkits includes debugging UEFI firmware and pre-OS execution environments, hooking kernel structures for stealthy persistence, and analyzing OS-level interactions across different platforms. Lastly, we outline actionable insights for defense in firmware security by strengthening secure boot enforcement, kernel protection by improving detection, and OS-level integrity monitoring by runtime verification of boot and kernel processes. BOOTKITTY highlights the potency of low-level malware and the critical need for the better security design that offers holistic protection from the bare-metal startup process to OS execution.

In summary, we make the following contributions:

- We identify critical challenges in deploying bootkit-rootkit attacks across modern OSes.
- We design and implement BOOTKITTY, a hybrid bootkit-rootkit that exploits firmware and bootloader vulnerabilities, achieving persistent compromise with minimal adversary interaction.
- We demonstrate practical techniques to evade advanced security defenses for modern OSes.
- We highlight boot process weaknesses and provide actionable insights into fortifying system integrity in a trust chain.

2 Background

Bootkits and Rootkits. Bootkits are a class of malware that operate at the earliest stages of the system boot, targeting components such as UEFI, Basic Input/Output System (BIOS) [25], the Master Boot Record (MBR) [4], or the bootloader. By executing malicious code before the OS is loaded, they achieve persistence covertly. As a result, bootkits are notoriously difficult to detect and remove, often surviving OS reinstallations and full disk reinitializations. Meanwhile, rootkits are a class of malware designed to stealthily obtain top-level privileges within an OS, allowing an attacker to maintain long-term control and surveillance. They evade detection by concealing their presence from critical security components, including file systems, process lists, and system logs. Once active, a rootkit can install backdoors, harvest sensitive information, manipulate kernel drivers, intercept system functions. By remaining hidden for extended periods, rootkits serve as a persistent foothold for further attacks or data exfiltration, posing a critical security threat. This work introduces BOOTKITTY, a bootkit-rootkit hybrid.

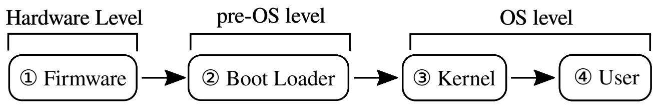

Figure 1: **Modern OS Boot Process (§2).**

Boot Process in Modern OS. A modern OS (*e.g.,* Windows, Linux, Android) follows a similar boot process, although their implementation varies by architecture. Figure 1 illustrates the general boot process of modern OSes. ① When the power is turned on, the firmware initializes the hardware first. Windows and Linux use UEFI, while Android uses a Boot ROM [94]. ② After that, the process moves on to the bootloader stage. In Windows, the boot manager runs first, and then the OS Loader in sequence. In Linux, Shim [100] is executed first, then GRUB2 [37], and in Android, the Little Kernel (LK) [56] is launched. ③ Once the bootloader finishes, the kernel is loaded and initializes hardware such as the CPU, memory, and peripheral devices, setting up internal data structures and scheduling. At this time, if Virtualization-based Security (VBS) [64] is enabled in Windows, the hypervisor is also initialized during kernel loading. ④ Finally, each kernel initializes user-space processes and services to complete the boot process, bringing the system to a fully operational state.

Secure Boot. Secure Boot is a security feature of the UEFI that ensures a secure boot process by verifying only trusted software through cryptographic signatures. Such trusted software includes bootloaders like the Windows boot manager and Linux's Shim [100], both of which are trusted within the UEFI boot chain due to their official Microsoft signatures. In the case of the Shim bootloader, it incorporates an additional mechanism known as the Machine Owner Key (MOK) [103], which allows users to manage trusted keys for loading custom or third-party bootloaders, kernel modules, or drivers without breaking the Secure Boot chain. Together, UEFI and MOK assist in verifying boot components, prevent unauthorized modifications, and preserve system integrity throughout the boot sequence. Note that Secure Boot and MOK are independent of modern OSes, but supported by multiple OS environments, including Windows and major Linux distributions.

Advanced Protection Mechanisms for Booting. While OSes differ in architecture, modern boot chains incorporate shared security measures to ensure system integrity and prevent unauthorized modifications. First, at the firmware level, *Secure Boot* ensures that only trusted and digitally signed software components are allowed to execute during the boot process. The firmware establishes a chain of trust with the Platform Key (PK) [107], which ensures each subsequent component (*e.g.,* boot manager or bootloader) is verified. Second, the OS kernel enforces code integrity during loading, using DSE in Windows and Kernel Module Signing [54] in Linux. Besides, kernel integrity protection mechanisms, such as PatchGuard and Kernel Lockdown, help safeguard critical system data from unauthorized modifications. Third, Mandatory Access Control (MAC) [42] enforces strict security policies to protect the bootloader and critical configuration files. For instance, SELinux [93] restricts file access, mitigating tampering by permitting modifications or execution only by authorized processes. Fourth, VBS leverages a hypervisor to isolate critical system processes and enforce security policies. A good example is Credential Guard [66] and Hypervisor-Protected Code Integrity (HVCI) [68] in Windows, and Protected Kernel Virtual Machine (pKVM) [11] and Android Virtualization Framework (AVF) [9] in Android. Note that BOOTKITTY is designed to circumvent varying protection mechanisms in modern OSes.

3 Threat Model and Challenges

This section describes our threat model and obstacles we encounter when implementing a bootkit-rootkit against modern OSes equipped with advanced security mechanisms.

3.1 Threat Model

We assume an adversary with physical access to the target system (victim), capable of initiating rapid infection within minutes. The victim system is running Windows or Linux under a normal user account, or Android with standard user privileges. For the infection, the adversary may leverage known vulnerabilities to escalate privileges if exploitable conditions are met. For both Windows and Linux, if an adversary gains the ability to execute arbitrary commands with SYSTEM privileges on Windows or root privileges on Linux, they could remotely deploy bootkit or rootkit modules without requiring physical access. However, our current implementation necessitates physical access for the initial infection, as we currently lack viable remote code execution exploits for either platform. For Android, strict platform restrictions make remote flashing infeasible, therefore physical access is required for the initial compromise. Additionally, we exclude remote exploits from our threat model (§8), as achieving remote code execution and evading antivirus is beyond the scope of this work. Nevertheless, if such an exploit were available, our system could support remote deployment. Once installed, the bootkit and rootkit are capable of bypassing antivirus measures. BOOTKITTY targets multiple testbed platforms, including (but not limited to) Windows 24H2 (October 2024), Linux Ubuntu 24.04.01 (kernel 6.8.0-31), and Android 4.14.186 on Galaxy A325N devices where the target systems hold several 1-day vulnerabilities that BOOTKITTY should harness for exploitation. As a final note, a system reboot is required to evade runtime security mitigations during an infection phase.

3.2 Attack Overview

Initial Compromise. To infect a target system, we establish a USB connection that requires physical access to the target device. For Windows and Linux, we emulate a keyboard (*e.g.,* BadUSB [22, 27]) to interact with the system automatically. Upon connection, this virtual device is recognized as a keyboard, running predefined commands to download and launch BOOTKITTY from a remote server. For Android, the adversary forces the device into a special boot mode (*e.g.,* Odin [1]) for flashing our payload to bypass pre-OS authentication.

BOOTKITTY Injection. A bootkit-rootkit can be injected across Windows, Linux, and Android. Despite variations in boot processes, BOOTKITTY follows four common key steps:

- **Infection:** An attacker exploits firmware vulnerabilities, allowing the overwriting of firmware objects (*e.g.,* boot logo images). Then, BOOTKITTY can execute the attacker-injected code at the firmware level (*e.g.,* Secure Boot bypass), breaking the chain of trust in a subsequent booting.
- **Bootkit Operation:** BOOTKITTY implants its custom modules and initiates code execution at the pre-OS stage, thereby establishing control before the OS loads.
- **Rootkit Installation:** The compromised bootloader patches critical routines to disable kernel defenses allowing for rootkit installation by loading arbitrary kernel drivers.
- **Rootkit Operation:** The surreptitious rootkit exfiltrates victim's sensitive information in an undetectable fashion while communicating with a remote server.

3.3 Challenges

Understanding Low-level Code. Constructing bootkits and rootkits is inherently challenging due to the need to exploit vulnerabilities in an operating system's low-level architecture. These types of malware must operate within the firmware, bootloader, or kernel, requiring specialized knowledge and precise execution. A bootkit executes before the OS is loaded into memory, necessitating a deep understanding of the boot

[1] Odin mode allows users to flash firmware, recovery images, and low-level system files on Samsung Android devices.

process and the ability to bypass security mechanisms such as Secure Boot that ensures boot integrity. In contrast, a rootkit operates within kernel space, directly interacting with the OS core functions. Both must evade security mitigations designed to secure the boot process and kernel operations, rendering their implementation highly complex.

Undocumented Routines. UEFI firmware and many OS components lack appropriate documentation, making reverse engineering a critical step in bootkit and rootkit implementation. Such absence of documentation introduces significant time and complexity, forcing a malware author to rely on trial-and-error approaches to understand internal routines. Any inappropriate interaction with the firmware or OS kernel can bring about system instability or crashes, with no clear guidelines available for troubleshooting or resolving such issues.

Limited Samples. Bootkits and rootkits typically target high-value systems, leveraging stealth techniques to evade detection while bypassing modern OS security features. Due to their complexity and the risks associated with their exposure, publicly available samples are rare in the wild.

Debugging Challenges. Debugging bootkits and rootkits presents significant technical challenges. Unlike rootkits that can utilize kernel debugging tools provided by most OSes, firmware and bootloader-level debugging is particularly challenging due to the absence of standardized tools. Besides, firmware testing on actual hardware (rather than on emulated environments) is non-trivial because it may introduce a substantial risk; *i.e.,* any failure during testing can result in permanent hardware damage.

Architectural Variations. The diversity of modern computing environments further complicates bootkit development. Fragmentation across hardware platforms, firmware implementations, and kernel architecture demand substantial engineering effort to create universally compatible malware.

4 BootKitty on Windows

This section delineates varying mitigation techniques to safeguard a boot process in Windows, followed by the design and implementation of BootKitty on Windows.

4.1 Boot Process Safeguards in Windows

DSE. DSE has been introduced since Windows Vista to ensure that only digitally signed drivers trusted by Microsoft can be loaded into the kernel [63]. Note that BootKitty nullifies DSE, which loads an unsigned rootkit driver.

VBS. VBS in Windows utilizes hardware virtualization and the hypervisor to create an isolated environment that protects critical system resources and security assets.

HVCI. HVCI is a security technique within the VBS environment that protects the kernel from tampering by preventing unsigned or modified code execution. Notably, BootKitty circumvents VBS and HVCI by disabling them.

PatchGuard. Windows OS detects and prevents unauthorized modifications to kernel structures through PatchGuard, which periodically verifies the integrity of critical kernel data and code sections. BootKitty disables PatchGuard to forge essential internal structures within the kernel space, such as the System Service Descriptor Table (SSDT) [113] and the Interrupt Descriptor Table (IDT) [110].

4.2 Bootkit Infection on Windows

Exploitation Overview. BootKitty aims to circumvent Secure Boot's security mechanisms to establish a persistent foothold within the system's boot chain, enabling the execution of malicious code during startup. At a high level, this process exploits a one-day vulnerability (CVE-2022-21894 [72]) by creating a legitimate system utility, `mcupdate`, with a malicious version. Once compromised, it interacts with the MOK installer to enroll a rogue MOK, allowing BootKitty to bypass Secure Boot verification. Figure 2 depicts the overall process of BootKitty on Windows, compromising the layered security mechanisms in the firmware, bootloader, and kernel in sequence.

Launching BootKitty. As described in §3.1, the installer (*i.e.,* `bootkitty.exe`) initiates the infection on a running Windows system. It disables critical security mechanisms (*e.g.,* HVCI or BitLocker [65]).

$\boxed{\text{I1}}$ **Manipulating Boot Configuration Data.** The Windows boot manager (`bootmgfw.efi`) plays a crucial role in booting by reading a database, Boot Configuration Data (BCD) [70], that contains essential boot-related settings and configuration options to load the OS properly. As with BlackLotus bootkit [72], BootKitty drops a vulnerable (but legitimately signed) boot manager, hypervisor loader, and libraries. This allows an adversary to add several options (*e.g.,* `avoidlowmemory`, `trucatememory`, `nointegritychecks`, `testsigning`) to BCD for further exploitation. To trigger the above vulnerability, BootKitty creates a malicious BCD that uses the previously dropped boot manager and hypervisor loader, and then applies it to the system. Once completed, BootKitty forces the Windows system to reboot. During startup, BootKitty executes the payload with the replaced EFI files. Bypassing Secure Boot is essential as it is usually enabled by default.

$\boxed{\text{I2}}$ **Bypassing Secure Boot Verification.** After rebooting, BootKitty can bypass the Secure Boot security feature by disabling Secure Boot verification with the manipulated BCD. This process involves shifting Secure Boot to a higher memory region (with the `avoidlowmemory` option) and removing its memory block (with the `truncatememory` option), effectively disabling enforcement. Although CVE-2022-21894 (also known as Baton Drop) was patched, a new vulnerability

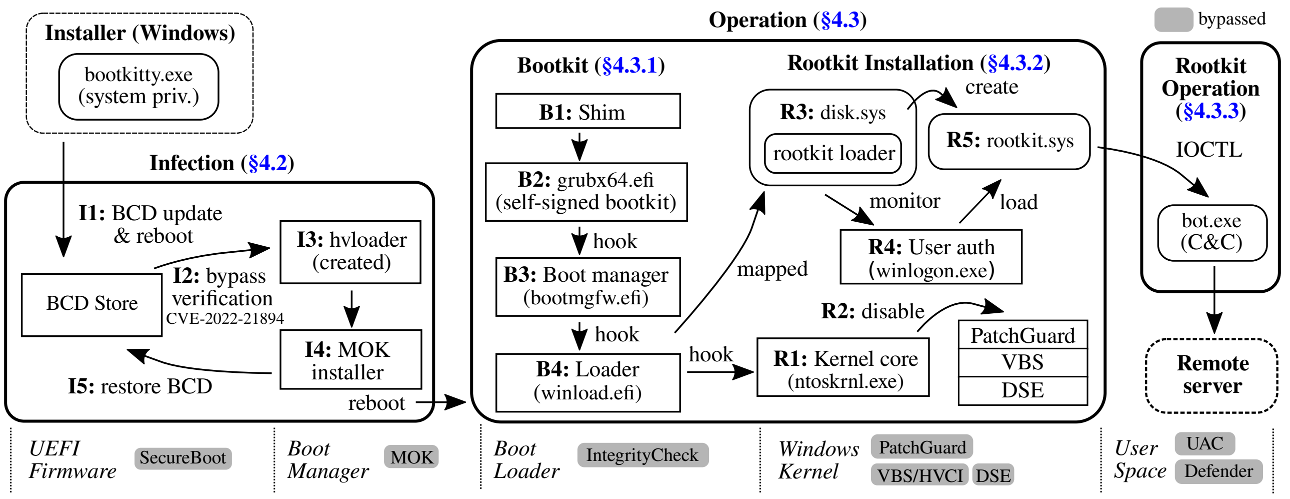

Figure 2: **Overview of BootKitty on Windows.** The infection phase corrupts the BCD and bypasses Secure Boot using a 1-day vulnerability. Then, the bootkit loads a malicious bootloader to gain kernel control, while the rootkit operates stealthily by disabling security features. The rootkit finally communicates with a remote server for Command-and-Control (C&C).

was discovered that bypasses the fix by allowing attackers to replace the patched boot manager with a vulnerable version. In response, Microsoft released additional patches [71] [2]. However, the issue persists widely, as the mitigation requires a complex, manual installation process, leaving many systems still vulnerable to the exploit. Interested readers refer to the exploit in detail [1, 95].

I3 Preparing MOK Registration. During the Secure Boot bypass, BootKitty forces UEFI to load a self-signed DLL. At this stage, BootKitty loads the compromised Windows hypervisor loader to load a custom-signed dll file (*e.g.,* `mcupdate.dll`). The file allocates a memory region and maps it to the code for MOK registration. A careful reader may note that DLLs are specific to Windows and not directly related to UEFI. However, executing code through a DLL is feasible because it is loaded before UEFI invokes a function that terminates UEFI services (`ExitBootServices()`).

I4 Enrolling Custom MOK. Next, BootKitty registers a custom MOK in NVRAM and sets a Microsoft-signed Shim binary as the default bootloader. Recall the Shim is a first-stage UEFI bootloader, followed by loading GRand Unified Bootloader 2 (GRUB2) [37] to launch the system. Since Microsoft only signs the Shim and delegates boot control to it, inserting custom keys into a user-managed key database is possible to circumvent Secure Boot. This step ensures the persistent control over the system by BootKitty after reboots.

I5 Restoring BCD. After enrolling the MOK, BootKitty restores the original BCD to avoid user detection (*i.e.,* returning to a seemingly normal boot process). At this point, the attacker breaks the chain of trust by leveraging the MOK to load a self-signed bootloader without restriction.

[2]This update addresses CVE-2023-24932 [71], which affects CVE-2022-21894 [72].

4.3 BootKitty Operations on Windows

4.3.1 Bootkit Operation

Overview. The bootkit operates at both the bootloader and kernel levels, using a multi-stage approach to disable security mechanisms and prepare for rootkit installation. At the bootloader level, BootKitty sequentially intercepts key components to modify kernel loading and allocate memory for rootkits. At the kernel level, BootKitty hooks NT kernel functions and initial drivers to disable security features while injecting and executing malicious code. Such a carefully orchestrated operation compromises the boot process so that BootKitty persistently can reside in a malicious environment after system reboots. Figure 2 (**B1** ∼ **B4**) illustrates the Bootkit operation as the following step-by-step process.

Exploitation. The bootkit intercepts multiple components to evade Windows' security mitigation. We deploy hooks in key Windows boot components, including the Windows boot manager (`bootmgfw.efi`), the Windows OS loader (`winload.efi`), and the Windows kernel (`Ntoskrnl.exe`), to disable security mechanisms during the boot process. **B1** BootKitty leverages a Microsoft-signed Shim to load custom code without triggering Secure Boot alerts. With a custom MOK, the Shim passes control to the next stage, allowing malicious components to load under the guise of legitimate boot files. **B2** Instead of loading the genuine Windows boot manager, BootKitty directs the Shim to load a self-signed GRUB2 image (`grubx64.efi`), which functions as the *bootkit*. Because the Shim is designed to load boot files with the specific name (`grubx64.efi`), we follow the naming convention for seamless execution within the Secure Boot framework. This bootkit intervenes in the normal boot process, manipulates the boot sequence, and ultimately enables kernel patching, leading to the installation and loading of the rootkit. $\boxed{\textbf{B3}}$ At this point, the bootkit bypassed Secure Boot by using a custom MOK. BootKitty hooks the original Windows boot manager (`bootmgfw.efi`) to intercept loading routines and allocate memory for subsequent components (*i.e.,* `winload.efi`). Compromising `winload.efi` in memory, it reserves space for rootkits and ensures persistence. $\boxed{\textbf{B4}}$ Finally, the compromised boot manager calls the Windows OS loader (`winload.efi`). At this stage, BootKitty injects hooks into the kernel-loading process, targeting functions that validate the memory map and prepare boot data for kernel execution.

4.3.2 Rootkit Installation

Figure 2 (middle) depicts the rootkit installation phases. $\boxed{\textbf{R1}}$ The Windows OS loader transfers control to the Windows kernel (`Ntoskrnl.exe`). During kernel initialization, the OS loader locates the kernel memory layout, applying memory patches to evade Windows mitigation mechanisms. $\boxed{\textbf{R2}}$ By hooking the kernel, BootKitty disables core security features such as PathGuard, VBS, and DSE. These hooks also inject a rootkit loader into an allocated memory buffer, enabling subsequent execution of malicious code in the kernel. $\boxed{\textbf{R3}}$ After thwarting security mechanisms on Windows, we intercept the disk driver (`Disk.sys`) to execute the rootkit loader by handling read/write operations. This loader installs the main rootkit payload into the kernel memory by creating the rootkit driver (`rootkit.sys`). Since the file system is not fully initialized during the early boot stages, the creation of the rootkit driver is postponed until the user authentication process (`winlogon.exe`) is loaded. $\boxed{\textbf{R4}}$ Once the authentication is complete, the pre-defined callback functions (via `PsSetLoadImageNotifyRoutine()`) at the rootkit loader create the rootkit driver file. This phase registers a service key (*e.g.,* `ImagePath`, `Type`, `Start`, `ErrorControl`) in the registry, followed by installing the driver. $\boxed{\textbf{R5}}$ Finally, the main rootkit payload (`rootkit.sys`) is loaded into the kernel memory. The rootkit maintains stealth, provides persistent control, and communicates covertly with user space modules and remote servers through input/output control operations.

4.3.3 Rootkit Operation

Rootkit Features. Table 1 outlines seven key features that BootKitty can provide as a rootkit. First, the rootkit conceals processes by unlinking their `EPROCESS` structures from the system's process list, rendering them invisible to tools like Task Manager. Second, it modifies flags in the object header associated with `EPROCESS` to prevent system shutdown, effectively blocking termination attempts. Third, for registry hiding, the rootkit intercepts operations through callback routines, allowing it to exclude specific keys from queries. Fourth, network sockets are concealed by hooking driver handlers in the NSI Proxy driver, filtering connection tables to remove targeted entries. Fifth, files can be hidden using SSDT hooks to intercept directory queries and manipulate directory information structures, masking specific files. Sixth, location data can be manipulated by hooking a Registry function, enabling the modification of the Windows location service registry key and providing access to the service. This allows the rootkit to use the service uninterrupted, even when GPS services are disabled. Lastly, keylogging is implemented by hooking keyboard drivers to intercept and log input while maintaining normal functionality. These techniques demonstrate how effectively the BootKitty's rootkit can subvert core system operations. To implement these functions, we leveraged widely available code and established techniques [3].

Functionality	**Interception Means**	**Exploitation**
Hiding processes	Unlink kernel structures	`EPROCESS`
Preventing a shutdown	Modify object headers	`EPROCESS`
Hiding registry keys	Hinder registry operations	Registry
Hiding sockets	Hook driver handlers	NSI Proxy driver
Hiding files	Intercept queries	SSDT hooks
Leaking GPS	Manipulate service keys	Registry
Keylogging	Capture keyboard inputs	Keyboard driver

Table 1: **Rootkit Features of BootKitty on Windows.**

5 BootKitty on Linux

This section details varying mitigation techniques to safeguard a boot process in Linux, followed by the design and implementation of BootKitty on Linux.

5.1 Boot Process Safeguards in Linux

Shim-based Protection. Shim [100] is a lightweight pre-bootloader for UEFI systems, primarily designed to act as an intermediary between UEFI (that enforces Secure Boot) and a second-stage bootloader like GRUB2. A majority of Linux distributions (*e.g.,* Ubuntu, Fedora, Debian, SUSE, Redhat) use GRUB2 that is not signed by Microsoft, and cannot run directly under Secure Boot. Shim resolves this issue by serving as a trusted bridge to securely load GRUB2 and the Linux kernel. In Linux, Shim implements a UEFI protocol (`shim_lock_protocol` [38]) that provides secure communication between Shim and GRUB2. This protocol locks boot parameters, prevents unauthorized modifications, and verifies the signatures of loaded components. Besides, the protocol facilitates state communication to ensure that only authorized kernel images are executed. By rigorously validating signatures, the protocol maintains a continuous chain of trust, effectively blocking untrusted binaries from running.

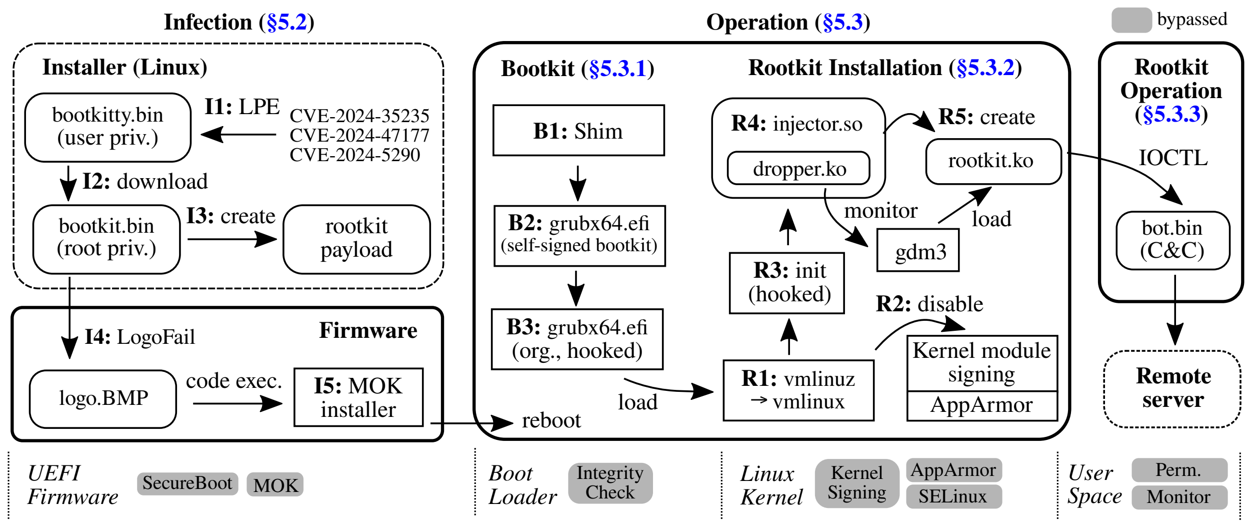

Figure 3: **Overview of BOOTKITTY on Linux.** The infection phase exploits Local Privilege Escalation (LPE) vulnerabilities to install a malicious bootkit and execute arbitrary code through a fabricated boot logo. The bootkit loads a tampered bootloader to bypass integrity checks, while the rootkit disables security mechanisms and injects itself into critical system processes.

Linux Security Module. The Linux Security Module (LSM) [111] enhances kernel security by integrating hooks that monitor and control critical operations. Prominent implementations, such as SELinux and AppArmor [12], are based on MAC. ① SELinux regulates the interactions between processes and files, mitigating the risk of privilege escalation and unauthorized access. It supports three operational modes: enforcing, permissive, and disabled, offering flexibility in security management. ② AppArmor enforces path-based access control through application-specific profiles. These profiles restrict resource access, prevent unauthorized actions, and strengthen security through compliance enforcement.

Kernel Module Signing. Kernel module signing ensures that only trusted modules are loaded into the system, protecting kernel integrity from malicious modifications. Each module is digitally signed during compilation using a private key. When loaded, the kernel verifies the signature with a trusted public key and rejects any unsigned or altered modules.

5.2 Bootkit Infection on Linux

Exploitation Overview. BOOTKITTY initiates its operation by chaining CVEs to escalate privileges. Then it downloads and executes a payload to inject a malformed boot image to NVRAM. By exploiting the UEFI's LogoFAIL [17] vulnerability, BOOTKITTY bypasses Secure Boot, deploying a rootkit by enrolling a MOK to load self-signed EFI files. This ensures persistence by loading the rootkit during boot. Figure 3 illustrates the overview of BOOTKITTY on Linux, where the following phases describe its infection.

$\boxed{\text{I1}}$ **Local Privilege Escalation.** First, BOOTKITTY obtains a root permission with LPE by chaining three 1-day Proof-of-Concept (PoC)s: CVE-2024-35235 [76], CVE-2024-47177 [77], and CVE-2024-5290 [78]. These exploits leverage vulnerabilities in the Common Unix Printing System (CUPS) [98] and the D-Bus interface to execute arbitrary code with root privileges.

$\boxed{\text{I2}}$ **Payload Preparation.** Next, BOOTKITTY downloads the payload binary (`bootkit.bin`), which contains exploitable code embedded in a BMP image to trigger the UEFI's LogoFAIL vulnerability, along with a rootkit payload.

$\boxed{\text{I3}}$ **Rootkit Generation.** First, it generates the rootkit payload, which will be loaded into the kernel at a later stage. Note that this payload is stored as `injector.so` for rootkit driver installation ($\boxed{\text{R4}}$). Second, BOOTKITTY overwrites the malformed image into NVRAM, which is used as the boot logo during startup.

$\boxed{\text{I4}}$ **LogoFAIL Vulnerability Trigger for Secure Boot Bypass.** BOOTKITTY then exploits the LogoFAIL vulnerability (CVE-2023-40238 [73]), a UEFI firmware flaw disclosed back in 2023, to bypass Secure Boot. LogoFAIL is a critical vulnerability that affects image parsers in UEFI firmware from the EFI Development Kit 2 (EDK2) [101] open-source project. In essence, this exploit allows the firmware to trigger an out-of-bounds write for arbitrary code execution during the Driver Execution Environment (DXE) [115] phase in UEFI. This can be done by writing a malformed BMP image (*e.g.,* `logo.bmp`) to the EFI partition and modifying NVRAM settings to enable the custom boot logo. We elaborate the exploitation process for interested readers in Appendix (Appendix A).

$\boxed{\text{I5}}$ **Enrolling Custom MOK.** Triggering LogoFAIL, BOOTKITTY injects a well-crafted shellcode that enrolls a custom MOK for maintaining persistent control over the boot process. Note that this process entails the loading of self-signed (unauthorized) EFI files.

5.3 BootKitty Operations on Linux

5.3.1 Bootkit Operation

Figure 3 (middle) illustrates the initialization process of the BootKitty's Bootkit operation. $\boxed{\text{B1}}$ A standard Linux booting begins with a signed bootloader, Shim, which enables Secure Boot by loading a verified OS bootloader (*e.g.,* GRUB2). BootKitty leverages a trusted Shim to load the next stage without raising any security alarms. $\boxed{\text{B2}}$ Next, BootKitty loads a self-signed GRUB2 bootloader (`grubx64.efi`) and set up function hooking on the original GRUB2. It is worth noting that BootKitty can bypass Secure Boot because of the custom signature that is enrolled in the MOK database, $\boxed{\text{B3}}$ The self-signed GRUB2 subsequently loads the original GRUB2 bootloader with multiple code patches applied. Notably, the GRUB2 verifier is thwarted at this stage. It also hooks the `start_image()` function responsible for loading the compressed Linux kernel (*i.e.,* `vmlinuz`) for further control.

5.3.2 Rootkit Installation

As depicted in Figure 3 (right), BootKitty installs a series of hooks within the kernel process to facilitate rootkit installation on the system. $\boxed{\text{R1}}$ Once the Linux system transitions to kernel space, the kernel decompresses and resumes execution. Because hooking the compressed image (*i.e.,* applying patches that take effect after decompression) is complex, we instead wait for the decompression process to complete. Specifically, BootKitty hooks the decompression function (`decompress_kernel()`) in `vmlinuz` and waits for the OS to generate a raw Linux kernel image (`vmlinux`), at which point we install another hook. $\boxed{\text{R2}}$ At this stage, BootKitty patches critical kernel functions to disable Linux security mitigations. In particular, it modifies `aa_audit_file()` to always return 0, effectively disabling AppArmor, and patches the kernel module signing code to circumvent signature enforcement. $\boxed{\text{R3}}$ To prepare for the rootkit installation, BootKitty targets `init`, the first user-space process. It locates the address of `envp_init()` and modifies an environment variable for the `init` process. Specifically, we configure the kernel to launch `init` with the `LD_PRELOAD` setting, ensuring that the rootkit installer loads the implanted library (`injector.so`) alongside the initial process. $\boxed{\text{R4}}$ When the rootkit injector takes control from the `init` process, the file system is not yet fully initialized, preventing the creation of the rootkit driver (`rootkit.ko`). To handle this, the dropper module (`dropper.ko`) monitors the login process (`gdm3`) as an indicator that system initialization is complete. $\boxed{\text{R5}}$ When the `gdm3` becomes active, it creates and loads the rootkit kernel module (`rootkit.ko`), completing the rootkit installation. Since kernel module signing and AppArmor mitigations are successfully invalidated, BootKitty can now load the custom rootkit driver without restriction.

Functionality	**Interception Means**	**Exploitation**
Hiding modules	Unlink kernel structures	Module list
Hiding processes	Hook `getdents64()`	Kernel output
Hiding sockets	Hook `tcp4_seq_show()`	Kernel output
Hiding files	Hook `getdents64()`	Kernel output
Keylogging	Capture keyboard inputs	Device

Table 2: **Rootkit Features of BootKitty on Linux.**

5.3.3 Rootkit Operation

Table 2 summarizes five key features that BootKitty can offer as a rootkit. Similar to Windows, the rootkit conceals a certain module by unlinking its corresponding kernel structure, thereby disappearing the module name in the system. To hide a particular process, a network socket, or a file, the rootkit intercepts kernel functions such as `getdents64()` that retrieves process or file information and `tcp4_seq_show()` that displays network socket information, which are responsible for generating an output. Lastly, the rootkit provides the keylogging feature by reading inputs directly from a keyboard device and storing them into a local file.

6 BootKitty on Android

Android devices may vary significantly across manufacturers, resulting in fragmentation issues, such as differences in software and hardware implementations. This section focuses on the multi-layered security features of Samsung-manufactured Android mobile devices, running on the ARM architecture.

6.1 Boot Process Safeguards in Android

6.1.1 Secure Boot Chain

ARM Exception Levels. ARM Exception Levels (EL) [13] define execution privileges. EL3 (Secure Monitor Mode) manages transitions between the Secure World, for sensitive operations like the Trusted Execution Environment (TEE) [8, 21], and the Normal World running Android. EL1 (Kernel Mode) is used for the OS kernel, while EL0 (User Mode) runs user-space applications. This hierarchy ensures secure isolation of critical processes.

Secure Boot. The Android security framework is tightly coupled with hardware to provide a chain of trust. The Secure Boot in Android [5] is analogous to those in Windows and Linux, which ensures that only signed (*i.e.,* trusted) firmware and boot components are loaded. The secure boot chain begins with the Boot ROM running at EL3 in the Secure World. The Boot ROM initializes hardware and verifies the

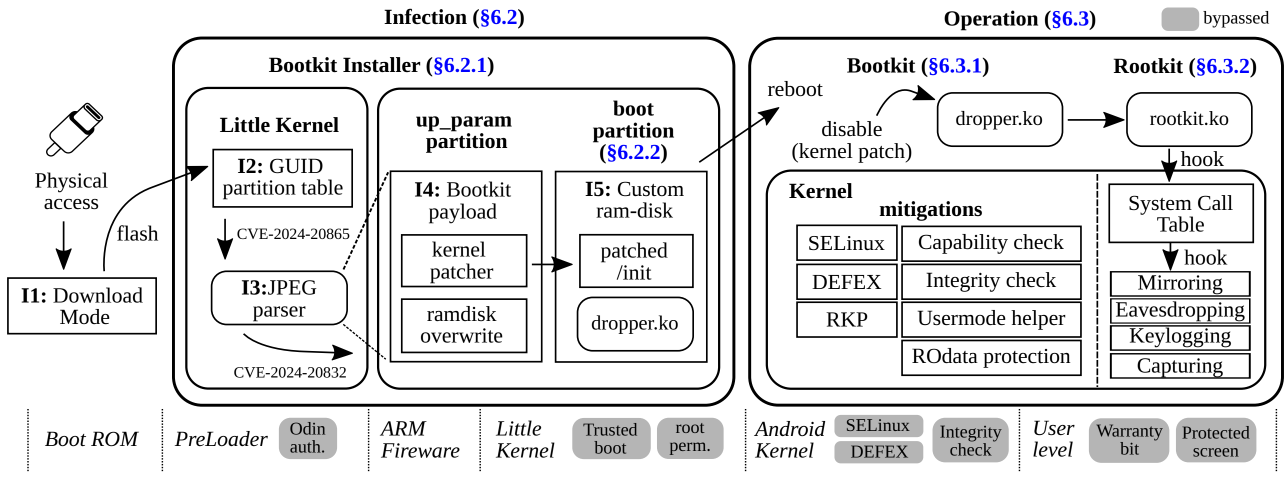

Figure 4: **Overview of BootKitty on Android.** The attack chain involves exploiting Odin mode and vulnerabilities in the LK bootloader to install a persistent payload. The bootkit modifies the boot partition and kernel, disabling security mechanisms such as SELinux, Defeat Exploit (DEFEX), and Real-time Kernel Protection (RKP). The rootkit then hooks into the system call table and enables malicious activities like mirroring, keylogging, and data capture while it bypasses integrity checks.

preloader, which is running on EL3. Then, it loads ARM Trusted Firmware to set up TEE, which runs the secure OS (*e.g.,* TEEGRIS [92]). At the same time, the preloader switches to the Normal World to initialize the LK at EL1. The LK then loads and starts the Android OS at EL0. This process ensures the Secure and Normal Worlds operate independently while securely transitioning between stages.

6.1.2 Multi-Layered Security Architecture in Android

Trusted Boot. Trusted Boot [90] is Samsung's implementation of Google's Verified Boot. It verifies bootloaders, kernels, and platform builds to prevent unauthorized or outdated versions from compromising the system. Besides, Trusted Boot captures snapshots of the system state during boot and securely stores them in TrustZone [81]'s TEE. A Trustlet [51] evaluates these snapshots and triggers a special bit (*e.g.,* Knox Warranty Bit) if it detects outdated or tampered components, such as unsigned kernels or disabled security features. This mechanism ensures device integrity beyond Secure Boot by validating signatures.

Security Enhancement. Security Enhancement (SE) for Android [23] enhances OS-level security through MAC enforcement. It applies MAC at two levels: kernel and middleware. At the kernel level, SELinux policies restrict access to sensitive resources by embedding access check hooks. At the Middleware level, Middleware MAC (MMAC) [20] extends these controls to inter-component communication, ensuring secure interactions between Android applications and system components. By integrating these layers, SE for Android achieves data isolation, limits root privileges, and protects applications from unauthorized access.

TrustZone-based Integrity Measurement Architecture. The TrustZone-based Integrity Measurement Architecture (TIMA) [30] safeguards kernel integrity by leveraging TrustZone's Secure World. It continuously monitors the Android kernel for unauthorized modifications using mechanisms such as Periodic Kernel Measurement (PKM) [30] and RKP [91]. PKM conducts periodic scans to detect the signs of kernel tampering, while RKP actively enforces kernel integrity by intercepting and blocking unauthorized modifications to code, data, and control flows in real time. By integrating these protections, TIMA strengthens system resilience against sophisticated kernel-level attacks.

DEFEX. DEFEX [87] surveils the abnormal behaviors of an application and mitigates potential threats. It can automatically terminate the application that attempts unauthorized actions, such as LPE attacks.

6.2 Bootkit Infection on Android

Exploitation Overview. Figure 4 illustrates the overview of BootKitty on Android. We develop the bootkit installer and bootkit based on the publicly available PoC [16]. The bootkit installation begins by enabling download mode (also known as the Odin mode) on the target Android device. This allows the flash of the GUID Partition Table (GPT) [105] without authentication (CVE-2024-20865 [75]) and modifies the Partition Information Table (PIT) [40] to insert a malicious payload. A heap overflow vulnerability (CVE-2024-20832 [74]) exists in LK's custom JPEG parser. The exploitation flashes a crafted JPEG file to the `up_param` partition, which leads to arbitrary code execution and persistence. Finally, the bootkit installer deploys a kernel patcher and a custom ramdisk. It disables security mitigations and enables stealthy kernel module loading, which demonstrates the risks of weak validation in critical boot components.

I1 **Activating Download Mode.** The bootkit installation requires physical access to the target Android phone to activate the Download Mode. This specialized mode is triggered by a hardware button combination while the device is powered off. If the phone is on or locked, we can force a shutdown to enable Download Mode, allowing USB communication for low-level operations like firmware flashing. Physical access is critical, as it enables direct manipulation of the boot sequence for subsequent attack phases. Once Download Mode is enabled, we flash the malicious bootkit installer to the device. The installer exploits vulnerabilities in GPT during the flashing process, targeting the LK bootloader.

I2 **Flashing Payload.** Samsung's Odin tool, used for flashing firmware in Download Mode, typically verifies image signatures before installation. However, a vulnerability (CVE-2024-20865 [75]) allows an attacker to flash the GPT without authentication. By exploiting this, BootKitty can modify GPT to update PIT, enabling the flashing of arbitrary data, including the partition (*e.g.,* `up_param`) containing a malformed JPEG (*e.g.,* `secure_error.jpg`) for further exploitation.

I3 **JPEG Parser Heap Overflow.** The custom JPEG parser has a heap overflow vulnerability (CVE-2024-20832 [74]) due to missing file size validation. After flashing a malicious JPEG to the `up_param` partition, we trigger the overflow, enabling arbitrary code execution and persistence, as `up_param` is not verified during boot. This allows the bootkit installer to write its payload into `up_param`, which stores boot-related configurations and assets. We detail the vulnerability in (Appendix B).

I4 **Kernel Patcher Deployment.** BootKitty writes the bootkit payload into the `up_param` partition. The payload consists of a kernel patcher that disables security mitigations and a ramdisk payload that replaces the device's existing ramdisk to allow persistent modifications.

I5 **Loading Kernel Modules via Custom Ramdisk.** Finally, BootKitty injects a specially crafted payload into the boot partition. The modified ramdisk contains ① a patched `init` process to ensure execution of the attacker's payload at boot, and ② a kernel module (`dropper.ko`) that loads additional malicious code. Our method utilizes a custom ramdisk to covertly load malicious kernel modules. By preparing the custom ramdisk, BootKitty establishes persistence, enabling the attacker to execute arbitrary code upon every reboot.

Building a Custom Ramdisk. A custom ramdisk is crucial for bypassing Android's security mechanisms and stealthily loading a malicious kernel module. While Android enforces kernel integrity, its support for dynamic module loading presents an exploitable opportunity. To leverage this, we construct a custom ramdisk by modifying the boot image (`boot.img`), which contains the primary bootloader and initial filesystem. Inside this partition, the ramdisk provides essential scripts and binaries, including `init`, the system's initialization process. By extracting the ramdisk, we identify `init` as the core entry point for system initialization. We modify `init` to execute custom shellcode, using `openat()`, `finit_module()`, and `close()` to load a kernel module. The modified `init` and the malicious module are then repacked into a custom ramdisk, allowing us to inject our payload before security mechanisms activate. To further evade detection, we embed the modified ramdisk into a disguised file (`secure_error.jpg`). A patched LK routine extracts this data and overwrites the existing ramdisk in memory, modifying the Device Tree to reflect updated hardware configurations. These modifications enable arbitrary kernel module loading, effectively bypassing Android's kernel mitigations and making low-level exploitation possible via boot-time manipulation.

Bypassed Mitigation	Patch Region	Location Name
① SELinux	Code	`sel_write_enforce()`
② DEFEX	Code	`task_defex_enforcing()`
③ RKP	Code	`rkp_init()`
④ Capability check	Code	`security_capable()`
⑤ Integrity check	Code	`security_integrity()`
⑥ Unsermode helper	Data	USERMODEHELPER_PATH
⑦ ROData protection	Code	`mark_rodata_ro()`

Table 3: **Summary of Bypassing Android Mitigations.**

6.3 BootKitty Operations on Android

6.3.1 Bootkit Operation

Achieving arbitrary code execution in the kernel is essential for loading a malicious driver for a rootkit. To achieve this, BootKitty disables or bypasses key mitigations in the Android kernel, as shown in Figure 4 (right). We summarize the techniques and patch locations (*i.e.,* code or data) to bypass these mitigations in Table 3.

Bypassing Android Mitigations. ① *(Disabling SELinux Enforcement)* BootKitty disables SELinux by modifying its core enforcement mechanisms. Most Android devices run SELinux in enforcing mode, which logs and blocks unauthorized actions. By analyzing the Android kernel, we identify and patch key functions (*e.g.,* `sel_write_enforce()`) responsible for enforcing SELinux policies. This effectively bypasses security checks and allows for unrestricted execution of unauthorized actions. ② *(Disabling DEFEX in Samsung Knox)* DEFEX is supposed to prevent privilege escalation by restricting specific process execution, which is enforced via `task_defex_enforce()`. By patching this function, BootKitty disables Samsung Knox's DEFEX security feature. ③ *(Circumventing RKP)* Next, BootKitty circumvents Android's RKP by modifying memory protection parameters. RKP is initialized via `rkp_init()`, which enforces protection on data (*e.g.,* `rodata`) and code regions to prevent kernel modifications. One critical parameter, `endrodata`, marks the boundary of the `rodata` section. By modifying endrodata, we extend the writable range of rodata, allowing modifications to otherwise protected memory. ④ *(Bypassing Kernel Capability Restrictions)* The Linux kernel enforces fine-grained privilege controls through capabilities (*e.g.,* binding low ports, changing file ownership). BOOTKITTY modifies security_capable() to force these security checks to always succeed, effectively bypassing permission restrictions and granting unauthorized access to privileged operations. ⑤ *(Evading Android Integrity Checks)* To bypass Android's integrity verification mechanisms, BOOTKITTY modifies security_integrity(), which checks whether system files and configurations remain unaltered. As a result, unauthorized modifications go undetected. ⑥ *(Hijacking Kernel-Invoked User-Space Processes)* BOOTKITTY modifies USERMODEHELPER_PATH, which stores the execution path for user-space helper programs invoked by the kernel. By altering this path, BOOTKITTY tricks the system into executing malicious user-space programs. ⑦ *(Preventing Kernel Memory from Read-Only)* Finally, BOOTKITTY prevents the kernel from enforcing write protection on read-only sections (*e.g.,* rodata) by patching mark_rodata_ro(). The kernel invokes this function during initialization, and once enforced, any modification, such as hooking the syscall table, causes a kernel crash. However, the above patch allows continuous tampering with protected kernel memory, facilitating long-term persistence of malicious components.

Bypassing Warranty Bit Validation. The Android device validates the warranty bit during boot to detect unauthorized modifications to the system. To preserve the warranty bit [88], we carefully determine the optimal timing for kernel patching. We apply the patch immediately after this validation to ensure kernel modifications did not notice the modified warranty bit. As a result, Knox-enabled features such as Samsung Pay [89] remain functional, allowing an adversary to conceal the presence of a rootkit.

6.3.2 Rootkit Operation

Unlike traditional rootkits on Windows or Linux, the implementation of the BOOTKITTY's rootkit on Android demands substantial effort to address challenges arising from undocumented system internals and stringent security mechanisms. This section presents the technical details (and challenges) in the design and implementation of BOOTKITTY.

Syscall Table Hooking. The rootkit's core functionality involves hooking the syscall table to intercept and manipulate user interactions on the mobile device. The syscall table resides in the rodata section, marked as read-only by the set_mark_ro_data() function and protected by the rkp_init() mechanism. These hypervisor-enforced protections ensure syscall table integrity. Hooking the syscall table is challenging due to its dynamic population during runtime and lack of symbol exposure, making it inaccessible through tools like kallsyms [55]. To locate it, we first build the manufacturer-provided kernel. Using a debugger, we identify the syscall table by its well-known signature and surrounding addresses. Once located, we calculate the necessary symbol offsets. To exploit the syscall table, a bootkit patches rkp_init() and mark_rodata_ro(), disabling write protection for rodata, making the syscall table writable. With write permissions enabled, we hook the syscall table, modifying entries to execute arbitrary code.

Keylogging. On touch-based devices like smartphones, user input is processed through touch events managed by the kernel's input subsystem. The input_register_handler() function can be used to iterate through input handlers and identify the touch input device (*e.g.,* touch screen). During touch input processing, raw coordinates from the touchscreen are adjusted to match the device's display resolution. This ensures accurate mapping of touch events to the screen. Once adjusted, the coordinates are used for further processing, including virtual keyboard input. The activation status of the virtual keyboard UI is determined by reading the cmd_result file, which is associated with the Touch Screen Processor (TSP) [7] driver. On the rootkit side, a kernel thread periodically monitors this file and processes touch data only when the keyboard UI is active. To be specific, keyboard input is captured by predefining an array of coordinate values corresponding to each key. When touch coordinates fall within a key's bounds, the associated key information is logged.

Eavesdropping. PCM data interception enables real-time eavesdropping on microphone or speaker activity in Android audio processing. Android relies on Pulse Code Modulation (PCM) [28], a standard format for digital audio data. PCM devices handle audio input and output, with raw data processed at the kernel level through callback functions defined in the snd_pcm_ops structure. A critical function in this structure is copy_user(), which transfers PCM data between user space and kernel space. By hooking this function, we can intercept PCM data being processed by the system. The intercepted data can then be decoded and reconstructed into audio signals.

Bypassing Android Screen Capture Restrictions. Android restricts screen capture and recording on sensitive screens using the *eHidden* and *eSecure* flags, managed by the SurfaceFlinger graphics system. These flags prevent screen capture and mirroring on protected screens. To bypass the restrictions, we hook the CreateSurface() function. Hooking requires injecting a malicious library into the target process, which then modifies CreateSurface() to disable the *eHidden* and *eSecure* flags. As a result, screen capture and mirroring become possible on restricted screens.

Screen Mirroring. To enable real-time screen mirroring, we install a ffmpeg [33] binary on the device. Upon receiving a command from the C&C server, we execute the built-in screenrecord program to capture video data from the target application. The captured data is then piped to ffmpeg, which applies the appropriate encoding options to convert it into the

MPEG-TS format. Finally, the converted data is streamed in real time to the C&C server.

7 Implementation

We implement BootKitty as a multi-platform bootkit-rootkit targeting Windows, Linux, and Android, designed to establish early and persistent control with minimal attacker interaction. Our goal is to build a cohesive framework by selectively applying known techniques only when they meet strict criteria: availability, effectiveness, stealth, compatibility with the latest OS versions, and demonstrated reliability. While the Windows component builds on well-documented prior works [1,97], the Linux and Android implementations required substantial engineering effort due to scarcity of existing references and the need for custom chaining and debugging mechanisms.

Initial Stage Exploitation. Our approach automates the infection process and minimizes manual steps for the attacker. For Windows, we use a BadUSB input generator [27], which mimics keyboard input via a crafted USB dongle. The injected keystrokes download the BootKitty payload from an external server and escalate privileges by simulating a "Y" (*i.e.,* yes) input to approve the User Account Control (UAC) [69] prompt. The attacker only needs to plug in the BadUSB device for less than 10 seconds. Once done, the BootKitty installer infects the system and activates on the next reboot. For Linux, the crafted keyboard input downloads and executes the payload on the target system. Since Linux lacks UAC restrictions, the attack requires running a LPE exploit to gain root access. For Android, the attacker physically connects the target device via USB. The device is forcibly booted into Odin mode [31] through a power cycle. Exploiting CVE-2024-20865 [75] bypasses Odin authentication, enabling BootKitty installation.

Cross-Platform Implementation. We develop BootKitty that integrates bootkit with rootkit capabilities in multi-platforms. Table 4 summarizes BootKitty implementation. BootKitty targets Windows, Linux, and Android, employing a unified strategy: compromising the early boot process to establish persistent control over the system. In each case, the attack begins by bypassing integrity verification mechanisms, such as Secure Boot or Trusted Boot, followed by disabling kernel-level security protections to ensure long-term persistence. While the core logic remains consistent across platforms (*i.e.,* bypassing integrity checks, disabling kernel defenses, and maintaining control), the implementation varies significantly due to differences in OS architecture and security models. This approach allows BootKitty to adapt to each platform while preserving its overall goal of stealthy, persistent rootkit deployment.

- **Windows:** We develop modules compatible with Windows 24H2, operating irrespective of hardware restrictions. The Windows module consists of 2,366 Lines of Code (LoC) for the bootkit and 5,540 LoC for the rootkit, exploiting CVE-2022-21894 [72], CVE-2023-24932 [71] to bypass Secure Boot. During the development of the Windows bootkit module, we heavily relied on the BlackLotus bootkit [1]. While our implementation is primarily inspired by the BlackLotus bootkit, we selectively adopted techniques from prior work such as DSE and PatchGuard bypasses from EfiGuard [97], and hiding mechanisms from the VectorKernel [109] rootkit. We also incorporated SSDT hooking and keylogging methods from other rootkits, based on their proven effectiveness against modern defenses. These choices were made not solely due to their public availability, but because of their demonstrated practicality and impact. Each component was selected and integrated based on its reliability, stealth, and compatibility with Windows 24H2, the latest version available at the time of writing. Rather than merely combining isolated techniques, we developed a cohesive framework that enhances flexibility and persistence. This design supports the realistic simulation of modern rootkits and aids the development of effective defenses.
- **Linux:** The Linux module targets Ubuntu 24.04.01 with the 6.8.0-31 kernel on the Lenovo IdeaPad Slim 3. We prioritize the newest available OS versions and hardware when known vulnerabilities exist, which led to the selection of this OS and model. The module consists of 1,074 LoC for the bootkit and 585 LoC for the rootkit, leveraging CVE-2024-35235 [76], CVE-2024-47177 [77], CVE-2024-5290 [78], and CVE-2023-40238 [73] for privilege escalation, along with the LogoFAIL vulnerability. As of writing, PoCs for each of the above 1-day vulnerabilities were available, but no full-chain exploit combining them had been released. To achieve LPE, we chained three 1-day vulnerabilities, developing the LogoFAIL PoC from scratch.
- **Android:** The Android implementation is the most challenging due to debugging obstacles and the undocumented internals of Android OS. The Android module was tested on the Samsung Galaxy A325N (Android 4.14.186) as it requires a high-impact vulnerability, such as bypassing Odin authentication [75] to evade the secure boot chain early without complex user interactions. The module comprises 315 and 2,866 LoC for the bootkit and rootkit, respectively, exploiting CVE-2024-20865 [75], and CVE-2024-20832 [74] to manipulate boot parameters and disable security mechanisms.

Debugging UEFI Firmware. Debugging UEFI firmware is challenging due to hardware dependency, lack of persistent debugging tools, and limited crash recovery. Among the available debugging methods, we adopted the QEMU emulator to write exploits. Using Chipsec [58], a firmware analysis tool by Intel, we dumped the SPI flash region to extract the UEFI firmware. The vulnerable driver, `BmpDecoderDxe`, was then extracted from the firmware using UEFITool [57] and

OS	Version	Hardware Spec.	Line of Code		1-day Vulnerability
			Bootkit	Rootkit	
Windows	24H2 (Oct 2024)	Any	2,366 (C)	5,540 (C, C++, ASM)	CVE-2022-21894 [72], CVE-2023-24932 [71]
Linux	Ubuntu 24.04.01 (kernel: 6.8.0-31)	Lenovo IdeaPad Slim 3 15IAH8	1,074 (C, ASM)	585 (C)	CVE-2024-35235 [76], CVE-2024-47177 [77], CVE-2024-5290 [78], CVE-2023-40238 [73]
Android	Android 4.14.186	Galaxy A325N (Before Mar 2024)	315 (C, ASM)	2,866 (C, C++)	CVE-2024-20865 [75], CVE-2024-20832 [74]

Table 4: **Overview of BootKitty implementation.**

loaded into the virtual environment. Within the UEFI shell, we retrieved information about the loaded driver, including its image base address. This address enabled us to rebase the driver in a local debugger, allowing for precise dynamic analysis (e.g., consistent memory space information between the real UEFI environment and the local debugger).

Debugging Android Firmware. We utilize MTKClient [2] to debug MediaTek SoC-based devices (*i.e.,* Galaxy A325N). MTKClient exploits chipset vulnerabilities to modify partitions and provides features like rooting, partition dumping, and firmware flashing. To prepare for debugging, we switch the device to Boot ROM Mode (BROM Mode) by shorting specific mainboard pins. Once in BROM Mode and connected to MTKClient, we examine the firmware's internal state. Applying the publicly released 1-day vulnerability requires knowing the internal heap address of our payload and a controllable buffer address. To achieve this, we modified the LK bootloader to record memory values in an unused partition during boot, leaking the necessary information.

Hooking Point Analysis on Linux. Linux bootkits in UEFI remain relatively underexplored, necessitating a thorough analysis of the boot process to identify key functions for installing hooks. We focus on GRUB2 execution, kernel loading, kernel initialization, and kernel decompression. We identify patching the `start_image()` function in the bootloader as the first step to influence subsequent hooks. The hooked function then patches `decompress_kernel()`, allowing arbitrary kernel modification immediately after the kernel is decompressed from the `vmlinuz` image. Finally, we successfully hook the `vmlinux` image, the uncompressed, raw Linux kernel.

8 Discussion and Limitations

Limitations. The current version of BootKitty requires a physical access to the victim's machine or device for initial infection. For Windows and Linux, if an adversary could execute arbitrary commands with SYSTEM privileges on Windows or root privileges on Linux, BootKitty could remotely deploy the bootkit and rootkit modules. However, the Windows module of BootKitty requires an additional LPE exploit, as the current version bypasses UAC by simulating keyboard input instead of using an LPE exploit. For Android, BootKitty cannot deploy its code remotely due to the absence of a powerful low-level vulnerability that would allow bypassing bootloader authentication for kernel module manipulation. As a final note, although BootKitty has not been tested in a universal environment due to variations across systems and devices, we provide valuable insights.

BootKitty Detection. BootKitty compromises the early boot process (*i.e.,* running before the OS loads), rendering it persistent and undetectable. As the infection occurs so early, the OS cannot reliably attest to its own integrity. Hence, hardware-based attestation can help defend against bootkits. First, UEFI vendors can implement detection modules to verify boot components before execution, preventing bootkits from running at startup. Second, monitoring the MOK list ensures only authorized boot components from the same manufacturer are loaded, which prevents unauthorized modifications and maintain boot integrity. Third, continuous monitoring of boot components after signature verification enables detection of bootkits even when Secure Boot is bypassed. This approach addresses a key weakness in traditional trust-chain models. We leave monitoring module deployment at the UEFI level as our future work.

BootKitty Mitigation. We discover that memory permissions can significantly ease infiltration for adversaries. During the UEFI's DXE phase, drivers' code sections are configured with RWX (Read, Write, Execute) permissions by default, posing a security risk. Similarly, Android's LK faces the same permission issue. Introducing fine-grained access controls would increase the difficulty for adversaries to execute shellcode from an initial vulnerability. Additionally, multiple layers of security mechanisms must be implemented beyond Secure Boot to achieve stronger protection in the UEFI environment. UEFI currently relies heavily on Secure Boot but lacks complementary security enhancements. If Secure Boot is bypassed or disabled, the system becomes highly vulnerable to a range of attacks. A robust firmware security stack requires defense-in-depth strategies that do not depend solely on Secure Boot. Incorporating features like Address Space Layout Randomization (ASLR) could further impede adversaries by making code execution locations less predictable.

9 Related Work

Evolution of Bootkits. Early studies, such as BootRoot [96], Vbootkit [49], and Stoned Bootkit [48], focus on a BIOS-based bootkit that manipulates the MBR, Volume Boot Record (VBR) [39], or NT loader (NTLDR) [35, 85]. With the transition from BIOS to UEFI, new attack surfaces have emerged. Researchers have identified vulnerabilities in the UEFI firmware and boot components, highlighting the expanding threat landscape [15,114]. These findings underscore the evolution of bootkit attacks, necessitating robust mitigation strategies.

Evolution of Rootkits. Early malware targeting user-space programs replaces system binaries with modified versions to conceal malicious activities, mirroring the primary objective of traditional rootkits. Notable examples include T0rn [14] and the SunOS Trojan [59], which manipulate commands like `ls` and `ps` to hide files, processes, and network connections while covertly harvesting sensitive data. As OS enforce a separation between user and kernel spaces to enhance security, rootkits adapte by shifting their focus to core OS functions within the kernel space [19]. This evolution leads to the emergence of true rootkits, such as FU Rootkit [34], Fivesys [82], Mingloa [86], and Demodex [29] in Windows environments, alongside Knark [26], Adore [26], Raptile [6], and Syslogk [80] in Linux environments.

UEFI Attacks and Defenses. The transition from BIOS to UEFI enhances security (*e.g.,* Secure Boot), boot speed, and flexibility, however, it inevitably expands the attack surface accordingly. In UEFI firmware, System Management Interrupts (SMI) [60] handlers are responsible for executing System Management Mode (SMM) code in response to specific system events. SPENDER [112] discovers the vulnerability in such SMIs. Similarly, malicious SMM drivers facilitate rootkit deployment, highlighting critical security gaps [99]. Meanwhile, boot script flaws during the S3 [41] resume process introduce a new attack vector [45, 108]. Besides, studies on the Intel Management Engine (ME) and UEFI firmware uncovers additional attack vectors [44, 79]. In response, UEFI implements enhanced protections such as SMM isolation and SMRAM, along with detection mechanisms [50] (*e.g.,* Secure Boot, Boot Guard [102], Chipsec [58], and UEFITool [57]).

10 Conclusion

Bootkits and rootkits remain one of the most challenging security threats due to their ability to evade detection, bypass modern defenses, and persist at the lowest levels of a system. In this paper, we present BootKitty, a hybrid bootkit-rootkit capable of bypassing security mechanisms across Windows, Linux, and Android. We demonstrate the feasibility of stealthy, low-level attacks despite existing mitigations. By implementing and analyzing these attacks, we uncover undocumented system behaviors and low-level vulnerabilities that adversaries could exploit. This research highlights critical security gaps and identifies areas where stronger defenses are needed. Additionally, we hope that our work lays the groundwork for enhancing OS security, ultimately strengthening boot-time and kernel-level protections.

Ethical Considerations

Our research explores the implementation of a bootkit-rootkit hybrid with the potential for adversarial weaponization. To mitigate risks and prevent misuse, we have not publicly released the full source code or binaries. However, we are willing to provide sanitized binaries or source code, excluding malicious payloads such as data exfiltration, to verified research institutions upon request. This approach strikes a balance between transparency and responsible disclosure, enabling reproducibility while minimizing the risk of unethical use.

Artifacts

We provide pre-configured VM and QEMU images that reproduce the bootkit's functionality for testing and analysis:

- **VM environment:** The pre-built VMware VM images are available for download at `https://zenodo.org/records/15501870`
- **QEMU virtual disks:** The QCOW2 images, ready to run on QEMU, are available for download at `https://zenodo.org/records/15582744`

Acknowledgements

We thank the anonymous reviewers and our shepherd, Fabio Pagani, for their feedback and guidance. We are also grateful to Sumin Hwang and Youngjin Sim for their collaboration on this project. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00337414; Binary Micro-Security Patch Technology Applicable with Limited Reverse Engineering under SW Supply Chain Environments and No. RS-2024-00437306; Development of Integrated Platform for Expanding and Safety Applying Memory-safe Languages), and the KITRI Best of the Best program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] Blacklotus uefi windows bootkit. https://github.com/ldpreload/BlackLotus.

[2] Mtkclient. https://github.com/bkerler/mtkclient.

[3] Vector kernel. https://github.com/daem0nc0re/VectorKernel, 2023.

[4] Akbal, E., Yakut, Ö. F., Dogan, S., Tuncer, T., and Ertam, F. A digital forensics approach for lost secondary partition analysis using master boot record structured hard disk drives. *Sakarya University Journal of Computer and Information Sciences* (2021).

[5] Alendal, G., Dyrkolbotn, G. O., and Axelsson, S. Forensics acquisition—analysis and circumvention of samsung secure boot enforced common criteria mode. *Digital Investigation* (2018).

[6] Alton, L. *Root Kit Discovery with Behavior-based Anomaly Detection through eBPF*. PhD thesis, Technische Universität Wien, 2024.

[7] Android. Touch devices. https://source.android.com/docs/core/interaction/input/touch-devices?hl=en, 2025.

[8] Android Open Source Project. Trusty tee. https://source.android.com/docs/security/features/trusty?hl=en.

[9] Android Open Source Project. Android virtualization framework (avf) overview. https://source.android.com/docs/core/virtualization, 2023.

[10] Android Open Source Project. Verified boot. https://source.android.com/docs/security/features/verifiedboot?hl=en, 2024.

[11] Android Open Source Project. Avf architecture. https://source.android.com/docs/core/virtualization/architecture?hl=en#pkm-vendor-modules, 2025.

[12] AppArmor Project. Apparmor: Linux kernel security module. https://apparmor.net/, 2024.

[13] ARM. Learn the architecture - aarch64 exception model. https://developer.arm.com/documentation/102412/0103/Privilege-and-Exception-levels/Exception-levels?lang=en#md214-exception-levels__fig_exception_levels.

[14] Baliga, A., Chen, X., and Iftode, L. Paladin: Automated detection and containment of rootkit attacks. *Department of Computer Science, Rutgers University* (2006).

[15] Bashun, V., Sergeev, A., Minchenkov, V., and Yakovlev, A. Too young to be secure: Analysis of uefi threats and vulnerabilities. In *Proceedings of the 13th Conference of Open Innovations Association FRUCT (FRUCT 2013)* (2013).

[16] Bellom, M. R., Neveu, R., Melotti, D., and Viala, G. Attacking samsung galaxy: A boot chain and beyond. *Blackhat USA* (2024).

[17] Binarly. Brly-logofail-2023-003. https://github.com/binarly-io/Vulnerability-REsearch/blob/main/LogoFAIL/BRLY-LOGOFAIL-2023-003.md, 2023.

[18] binarly. Logofail exploited to deploy bootkitty, the first uefi bootkit for linux. https://www.binarly.io/blog/logofail-exploited-to-deploy-bootkitty-the-first-uefi-bootkit-for-linux, 2024.

[19] Bravo, P., and García, D. F. Rootkits survey. *architecture* (2011).

[20] Bugiel, S., Heuser, S., Sadeghi, A.-R., and Darmstadt, T. Towards a framework for android security modules: Extending se android type enforcement to android middleware. *Intel Collaborative Research Institute for Secure Computing* (2012).

[21] Busch, M., Westphal, J., and Mueller, T. Unearthing the trustedcore: A critical review on huawei's trusted execution environment. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 2020)* (2020).

[22] Cannoles, B., and Ghafarian, A. Hacking experiment by using usb rubber ducky scripting. *Journal of Systemics* (2017).

[23] Chen, H., Li, N., Enck, W., Aafer, Y., and Zhang, X. Analysis of seandroid policies: Combining mac and dac in android. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)* (2017).

[24] Chiang, K., and Lloyd, L. A case study of the rustock rootkit and spam bot. *HotBots* (2007).

[25] Cooper, D., Polk, W., Regenscheid, A., Souppaya, M., et al. Bios protection guidelines. *NIST Special Publication* (2011).

[26] Dai Zovi, D. Kernel rootkits. *SANS Institute, InfoSec Reading Room* (2001).

[27] Dan Goodin. This thumbdrive hacks computers. "badusb" exploit makes devices turn "evil". https://arstechnica.com/information-technology/2014/07/this-thumbdrive-hacks-computers-badusb-exploit-makes-devices-turn-evil/, 2023.

[28] Documentation, A. L. Pcm (digital audio) interface. https://vovkos.github.io/doxyrest/samples/alsa/page_pcm.html, 2017.

[29] Dor Nizar, Malware Researcher. The return of ghost emperor's demodex. https://www.sygnia.co/blog/ghost-emperor-demodex-rootkit/, 2024.

[30] Dorjmyagmar, M., Kim, M., and Kim, H. Security analysis of samsung knox. In *Proceedings of the 19th International Conference on Advanced Communication Technology (ICACT 2017)* (2017).

[31] Drake, J. J., Lanier, Z., Mulliner, C., Fora, P. O., Ridley, S. A., and Wicherski, G. *Android hacker's handbook*. John Wiley & Sons, 2014.

[32] ESET. Bootkitty: Analyzing the first uefi bootkit for linux. https://www.welivesecurity.com/en/eset-research/bootkitty-analyzing-first-uefi-bootkit-linux/, 2024.

[33] FFmpeg Developers. Ffmpeg. https://www.ffmpeg.org/, 2023.

[34] Florio, E. When malware meets rootkits. *Virus Bulletin* (2005).

[35] Gao, H., Li, Q., Zhu, Y., Wang, W., and Zhou, L. Research on the working mechanism of bootkit. In *Proceedings of the 8th International Conference on Information Science and Digital Content Technology (ICIDT 2012)* (2012).

[36] Garcia, L., Brasser, F., Cintuglu, M. H., Sadeghi, A.-R., Mohammed, O. A., and Zonouz, S. A. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS 2017)* (2017).

[37] GNU. Gnu grub 2 manual. https://www.gnu.org/software/grub/manual/grub/grub.html, 2023.

[38] GNU Project. Gnu grub manual 2.12: shim_lock. https://www.gnu.org/software/grub/manual/grub/html_node/shim_005flock.html.

[39] Grill, B., Bacs, A., Platzer, C., and Bos, H. "nice boots!"– a large-scale analysis of bootkits and new ways to stop them. In *Proceedings of the 12th DIMVA Conference (DIMVA 2015)* (2015).

[40] Haizar, N., Kee, D. M. H., Chong, L. M., and Chong, J. H. The impact of innovation strategy on organizational success: A study of samsung. *Asia Pacific Journal of Management and Education* (2020).

[41] Han, S., Shin, W., Park, J.-H., and Kim, H. A bad dream: Subverting trusted platform module while you are sleeping. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)* (2018).

[42] IBM. Mandatory access control. https://www.ibm.com/docs/en/aix/7.2?topic=security-mandatory-access-control, 2025.

[43] INSYDE. Insyde official website. https://www.insyde.com/, 2025.

[44] INTEL. What is intel® management engine? https://www.intel.com/content/www/us/en/support/articles/000008927/software/chipset-software.html, 2023.

[45] JIAO, W., LI, Q., CHEN, Z., AND CAO, F. Uefi security threats introduced by s3 and mitigation measure. In *Proceedings of the 7th International Conference on Signal and Image Processing (ICSIP 2022)* (2022).

[46] KASPERSKY. Mosaicregressor: Lurking in the shadows of uefi. https://securelist.com/mosaicregressor/98849/.

[47] KIM, S., PARK, J., LEE, K., YOU, I., AND YIM, K. A brief survey on rootkit techniques in malicious codes. *J. Internet Serv. Inf. Secur.* (2012).

[48] KLEISSNER, P. Stoned bootkit. *Black Hat USA* (2009).

[49] KUMAR, N., AND KUMAR, V. Vbootkit 2.0-attacking windows 7 via boot sectors. In *Proceedings of the 7th Hack in the Box Security Conference (HITBSecConf 2009)* (2009).

[50] KUZMINYKH, I., AND YEVDOKYMENKO, M. Analysis of security of rootkit detection methods. In *Proceedings of the 1st IEEE International Conference on Advanced Trends in Information Theory (ATIT 2019)* (2019).

[51] LAPID, B., AND WOOL, A. Navigating the samsung trustzone and cache attacks on the keymaster trustlet. In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS 2018)* (2018).

[52] LI, X., WEN, Y., HUANG, M. H., AND LIU, Q. An overview of bootkit attacking approaches. In *Proceedings of the 7th International Conference on Mobile Ad-hoc and Sensor Networks (MSN 2011)* (2011).

[53] LINUX KERNEL DOCUMENTATION. Kernel lockdown. https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html, 2024.

[54] LINUX KERNEL DOCUMENTATION. Kernel mode signing. https://docs.kernel.org/admin-guide/module-signing.html, n.d.

[55] LINUX MAN PAGES. kallsyms - linux manual page. https://man.cx/kallsyms(8), 2023.

[56] Little kernel. https://github.com/littlekernel/lk.

[57] LONGSOFT. Uefitool - github repository. https://github.com/LongSoft/UEFITool, 2023.

[58] LOUCAIDES, J., AND BULYGIN, Y. Platform security assessment with chipsec. In *Proceedings of the 17th CanSecWest Conference (CanSecWest 2014)* (2014).

[59] MANAP, S. Rootkit: Attacker undercover tools. *Personal Communication* (2001).

[60] MANNTHEY, K. System management interrupt free hardware. In *Proceedings of the 2nd Linux Plumbers Conference (LPC 2009)* (2009).

[61] MATROSOV, A., RODIONOV, E., AND BRATUS, S. *Rootkits and bootkits: reversing modern malware and next generation threats*. No Starch Press, 2019.

[62] MICROSOFT. Kernel patch protection. https://learn.microsoft.com/en-us/previous-versions/windows/hardware/design/dn613955(v=vs.85)?redirectedfrom=MSDN, 2017.

[63] MICROSOFT. Kernel mode signing. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-requirements--windows-vista-and-later-, 2022.

[64] MICROSOFT. Virtualization-based security (vbs). https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs, 2023.

[65] MICROSOFT. Bitlocker overview. https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/, 2024.

[66] MICROSOFT. Credential guard. https://docs.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard, 2024.

[67] MICROSOFT. Driver signing policy. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-, 2024.

[68] MICROSOFT. Hypervisor-protected code integrity (hvci). https://learn.microsoft.com/en-us/windows-hardware/drivers/bringup/device-guard-and-credential-guard, 2024.

[69] MICROSOFT. User account control overview. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/, 2024.

[70] MICROSOFT LEARN. Overview of boot options in windows. https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/boot-options-in-windows, 2024.

[71] MICROSOFT SUPPORT. How to manage the windows boot manager revocations for secure boot changes associated with cve-2023-24932. https://support.microsoft.com/en-us/topic/how-to-manage-the-windows-boot-manager-revocations-for-secure-boot-changes-associated-with-cve-2023-24932-41a975df-beb2-40c1-99a3-b3ff139f832d, 2023.

[72] NIST. Cve-2022-21894: Secure boot security feature bypass vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2022-21894, 2022.

[73] NIST. Cve-2023-40238: Out-of-bounds write in insyde firmware. https://nvd.nist.gov/vuln/detail/CVE-2023-40238, 2023.

[74] NIST. Cve-2024-20832. https://nvd.nist.gov/vuln/detail/CVE-2024-20832, 2024.

[75] NIST. Cve-2024-20865. https://nvd.nist.gov/vuln/detail/CVE-2024-20865, 2024.

[76] NIST. Cve-2024-35235. https://nvd.nist.gov/vuln/detail/cve-2024-35235, 2024.

[77] NIST. Cve-2024-47177. https://nvd.nist.gov/vuln/detail/cve-2024-47177, 2024.

[78] NIST. Cve-2024-5290. https://nvd.nist.gov/vuln/detail/cve-2024-5290, 2024.

[79] OGOLYUK, A., SHEGLOV, A., AND SHEGLOV, K. Uefi bios and intel management engine attack vectors and vulnerabilities. In *Proceding of the 20th Conference of Open Innovations Association FRUCT (FRUCT 2017)* (2017).

[80] PÉREZ, D. Á. Syslogk rootkit. executing bots via" magic packets". *The Journal on Cybercrime and Digital Investigations* (2023).

[81] PINTO, S., AND SANTOS, N. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)* (2019).

[82] POGONIN, D., AND KORKIN, I. Microsoft defender will be defended: Memoryranger prevents blinding windows av. *arXiv preprint arXiv:2210.02821* (2022).

[83] REDINI, N., MACHIRY, A., DAS, D., FRATANTONIO, Y., BIANCHI, A., GUSTAFSON, E., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Bootstomp: On the security of bootloaders in mobile devices. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)* (2017).

[84] RESEARCHERS, E. Lojax: First uefi rootkit found in the wild, courtesy of the sednit group. Tech. rep., Tech. rep. ESET, 2018.

[85] RODIONOV, D., MATROSOV, A., AND HARLEY, D. Bootkits: Past, present and future. In *Proceedings of the 24th Virus Bulletin International Conference (VB 2014)* (2014).

[86] SALINAS, R. Fantastic rootkits and where to find them (part 2). https://www.cyberark.com/resources/threat-research-blog/fantastic-rootkits-and-where-to-find-them-part-2, 2023.

[87] SAMSUNG. Defeat exploit. https://docs.samsungknox.com/admin/fundamentals/whitepaper/samsung-knox-mobile-security/system-security/defeat-exploit/, 2025.

[88] SAMSUNG ELECTRONICS CO., L. Samsung knox platform for enterprise - white paper. https://image-us.samsung.com/SamsungUS/samsungbusiness/solutions/topics/iot/071421/Knox-Whitepaper-v1.5-20210709.pdf, 2021.

[89] SAMSUNG ELECTRONICS CO., L. Samsung pay: A secure and convenient mobile payment solution. https://www.samsung.com/samsung-pay, 2024.

[90] SAMSUNG KNOX. Trusted boot. https://docs.samsungknox.com/admin/fundamentals/whitepaper/samsung-knox-mobile-security/system-security/trusted-boot/.

[91] SAMSUNG KNOX. Real-time kernel protection. https://docs.samsungknox.com/admin/fundamentals/whitepaper/samsung-knox-mobile-security/system-security/real-time-kernel-protection/, 2025.

[92] SAMSUNGDEVELOPER. Samsung teegris. https://developer.samsung.com/teegris/overview.html.

[93] SELINUX PROJECT. Selinux: Security-enhanced linux. https://selinuxproject.org/page/Main_Page, 2017.

[94] SINGH, A., AND BHARDWAJ, A. Android internals and telephony. *Int. J. Emerg. Technol. Adv. Eng* (2014).

[95] SMOLÁR, M. Blacklotus uefi windows bootkit. https://www.welivesecurity.com/2023/03/01/blacklotus-uefi-bootkit-myth-confirmed/, 2023.

[96] SOEDER, D., AND PERMEH, R. eeye bootroot. *BlackHat USA* (2005).

[97] SUICHE, M. Efiguard. https://github.com/tandasat/EfiGuard, 2018. GitHub repository.

[98] SWEET, M. *CUPS (Common Unix Printing System)*. Pearson Education, 2001.

[99] SZAKNIS, M., AND SZCZYPIORSKI, K. The design of the simple smm rootkit. In *Proceedings of the 9th International Conference on Wireless Communication and Sensor Networks (ICWCSN 2022)* (2022).

[100] TEAM, R. H. B. Shim: A first-stage uefi bootloader. https://github.com/rhboot/shim/blob/main/README.md, 2024.

[101] TIANOCORE. tianocore-edk2. https://github.com/tianocore/edk2.

[102] TIANOCORE. Intel® boot guard. https://tianocore-docs.github.io/Understanding_UEFI_Secure_Boot_Chain/draft/secure_boot_chain_in_uefi/intel_boot_guard.html, 2021.

[103] TIANOCORE. Machine owner key (mok). https://tianocore-docs.github.io/Understanding_UEFI_Secure_Boot_Chain/draft/additional_secure_boot_chain_implementations/machine_owner_key_mok.html, 2021.

[104] TIANOCORE. Uefi secure boot. https://tianocore-docs.github.io/Understanding_UEFI_Secure_Boot_Chain/draft/secure_boot_chain_in_uefi/uefi_secure_boot.html, 2021.

[105] UEFI. 5. guid partition table (gpt) disk layout. https://uefi.org/specs/UEFI/2.10/05_GUID_Partition_Table_Format.html, 2022.

[106] UEFI FORUM. About uefi forum. https://uefi.org/about.

[107] UEFI FORUM. Uefi specification: Secure boot and driver signing. https://uefi.org/specs/UEFI/2.10/32_Secure_Boot_and_Driver_Signing.html, 2022.

[108] UEFI FORUM. Uefi platform initialization: S3 resume. https://uefi.org/specs/PI/1.9/V5_S3_Resume.html, 2024.

[109] UNDERGROUND MALWARE COMMUNITY. Vectorkernel rootkit, 2021. Known Windows kernel-mode rootkit referenced in security research and malware reverse engineering discussions.

[110] WANG, Y., GU, D., LI, W., LI, J., AND WEN, M. Virus analysis on idt hooks of rootkits trojan. In *Proceedings of the International Symposium on Information Engineering and Electronic Commerce (ISEEC 2009)* (2009).

[111] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium (USENIX Security 2002)* (2002).

[112] YIN, J., LI, M., WU, W., SUN, D., ZHOU, J., HUO, W., AND XUE, J. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP 2022)* (2022).

[113] ZHANG, J., LIU, S., PENG, J., AND GUAN, A. Techniques of user-mode detecting system service descriptor table. In *Proceedings of the 13th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2009)* (2009).

[114] ZHOU, Y., PENG, G., LI, Z., AND LIU, S. A survey on the evolution of bootkits attack and defense techniques. *China Communications* (2024).

[115] ZIMMER, V., ROTHMAN, M., AND MARISETTY, S. *Beyond BIOS: developing with the unified extensible firmware interface*. Walter de Gruyter GmbH & Co KG, 2017.

Appendix

A LogoFAIL Vulnerability (CVE-2023-40238)

Manufacturers develop UEFI [106] firmware based on the EDK2 [101] open-source project, each implementing unique image parsers for boot logo rendering. LogoFAIL [17] is a critical vulnerability in image parsers that exists in the Insyde, AMI, and Phoenix firmware, occurring early in the boot process, which amplifying its severity. We selected this vulnerability, CVE-2023-40238 [73], for bootkit development due to its significant impact on the UEFI and boot chain. For this study, we used a device running the vulnerable firmware version LTCN30WW. This out-of-bounds write flaw in Insyde [43] firmware arises during BMP file processing.

Figure 5 illustrates the exploitation process of the LogoFAIL vulnerability. The attack follows these steps:

1. **EFI Partition (Malicious Image Overwrite)**: BOOTKITTY modifies the `bootkit.bmp` file in the EFI partition, embedding shellcode inside the image. Then, the legitimate boot logo is replaced with a malicious one.

2. **NVRAM Modification (Enable Custom Boot Logo)**: The system firmware stores boot configuration settings

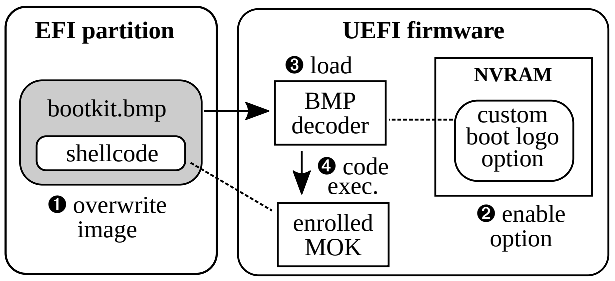

Figure 5: **LogoFAIL exploit.** The EFI partition is overwritten, a custom boot logo option is enabled, and the BMP decoder loads it, triggering shellcode execution to enroll a custom MOK.

in NVRAM. BootKitty enables the *custom boot logo option* in the UEFI [106] firmware settings, ensuring that the malicious `bootkit.bmp` is processed at boot.

3. **UEFI Firmware (Processing the Malicious Image)**: The firmware loads the BMP decoder to process the boot logo. Due to a vulnerability in the decoder, it incorrectly processes the malicious image.

4. **Exploiting UEFI Execution (Code Execution & Persistence)**: The embedded shellcode executes during the decoding process. The attacker then enrolls a MOK [103], allowing persistence across reboots and bypassing Secure Boot.

Arbitrary Write to Execute Shellcode. The essence of the vulnerability is described in Figure 6. As shown in the code, an attacker can control the input value of `height` and set it to zero (Line 6), causing the `BltEntry` address to point below `BltOutput` address. This enables arbitrary writes to a controllable memory location (Line 11-13), allowing code execution during the DXE [115] phase. UEFI [106] lacks $W \oplus X$ protection (Write XOR Execute) in its memory space, allowing arbitrary code execution by overwriting the code section. Specifically, we patch the code to redirect execution to a crafted image containing the shellcode.

B JPEG Parser Heap Overflow Vulnerability (CVE-2024-20832)

Samsung introduces a custom JPEG parser in LK to display logos and error messages at boot. In this parser, the JPEG file is stored in a fixed-size structure without proper size checks, resulting in a heap overflow. Owing to the LK's simplistic heap algorithm and the lack of mitigation features, an attacker can exploit this overflow to achieve arbitrary code execution within LK. Since memory in LK has RWX permissions (*i.e.*, Read, Write, and Execute), the shellcode embedded in the image file can be executed.

```
1  int64 fastcall DecodeRLE8(
2      EFI_GRAPHICS_OUTPUT_BLT_PIXEL *BltOutput,
3      int8 *a2, int64 a3, BMP_IMAGE *Image)
4  {
5      // if height is 0, index becomes negative value
6      BltEntry = &BltOutput[width * (height - i - 1)];
7      ...
8      do
9      {
10         // BltEntry points wrong memory -> Arbitrary write
11         BltEntry->Red = *(BYTE *)(a3 + 4 * v16 + 2);
12         BltEntry->Green = *(BYTE *)(a3 + 4 * v16 + 1);
13         BltEntry->Blue = *(BYTE *)(a3 + 4 * v16);
14         ++BltEntry;
15         --v17;
16     }
17     while (v17);
18 }
```

Figure 6: **Code snippet for** `CVE-2023-40238` [17].

C The BootKitty Incident: Community Response and Evaluation

During testing of the BootKitty bootkit on our internal server (1–16 November, 2024), we uploaded the payload to a test webserver, as our attack model relied on a BadUSB command to fetch the initial stage. During this period, an external crawler accessed the server and retrieved the bootkit binary (5 November, 2024). The crawler flagged the sample due to the term "Bootkit" in the file name and publicly disclosed its existence. The sample quickly gained attention as the first publicly identified Linux bootkit targeting UEFI systems. This milestone sparked interest from both the research community and the public, with BootKitty being widely referred to as the "first UEFI bootkit for Linux" [18, 32] (27 November, 2024). The incident highlighted the importance of Secure Boot and reignited awareness of firmware-level threats [17], challenging the prevailing assumption that UEFI bootkits target only Windows systems. It also ignited broad discussions on UEFI security within both industry and academia. After recognizing the unintended leak and public analysis, our team promptly contacted the analysts to clarify that the binary was part of BootKitty, a research prototype (28 November, 2024). The analysis report was later updated (2 December, 2024) to reflect that BootKitty was developed by a student research team and was not part of an active threat campaign.