2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS) | 979-8-3503-6563-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/QRS62785.2024.00064

The PIT-Cerberus Framework: Preventing Device Tampering During Transit

Rakesh Podder[1], Jack Sovereign[1], Indrajit Ray[1], Madhan B. Santharam[2], and Stefano Righi[2]
[1]Colorado State University, Fort Collins, Colorado, USA
[2]AMI US Holdings Inc., Duluth, GA, USA
[1]{rakesh.podder, jack.sovereign, indrajit.ray}@colostate.edu,
[2]{madhans, stefanor}@ami.com

Abstract—When a computing device, such as a server, workstation, laptop, tablet, etc. is shipped from one site to another (for example, from a vendor to a customer or from one branch location of an organization to another) it can potentially be subjected to unauthorized firmware modifications. The industry has sought to partially address this issue by focusing on securing the boot process. Secure boot provides attestation methods by a hardware root-of-trust to confirm the integrity of the device's BIOS/UEFI firmware. However, once a device boots up, it is relatively easy for a malicious adversary to tamper with the firmware. In this paper, we address this problem by preventing a secure boot unless done by an authorized user. We extend a hardware root of trust (HRoT) processor's ability to perform secure attestation by implementing a new functionality to securely lock and unlock the BIOS/UEFI or the BMC (Baseboard Management Controller) and implementing an authentication mechanism in the HRoT for determining authorized users. This ensures that the secure boot process won't commence unless authorized appropriately and provides a robust mechanism for securing the device's firmware during transit. The proposed PIT-Cerberus framework (PIT = Protection In Transit) leverages strong cryptographic techniques and has been implemented within a trusted microcontroller. We have contributed the PIT-Cerberus framework's libraries to Project Cerberus, an open-source project that offers a security platform for server hardware.

Keywords– Firmware Security; BIOS/BMC; Hardware Root of Trust; Secure Boot

1. Introduction

Firmware is the main software that allows a device's hardware to communicate with the operating system [1]. When a computer is turned on, its BIOS (Basic Input/Output System) firmware takes on the task of initializing the hardware by loading the System Management Interrupts, starting the Advanced Configuration and Power Interface and initiating the loading of the operating system. This mode of operation of the device is a high-privilege CPU mode, which stands apart from standard operating system execution modes like protected mode or long mode. The Unified Extensible Firmware Interface (UEFI) boot process, increasingly replacing BIOS, mirrors the conventional BIOS boot procedure's flow. Most platforms based on UEFI start their boot process with a minimal core block of code. This stage is known as the Security (SEC) phase [2]. It lays the groundwork for a secure boot, acting as a gatekeeper to verify and authenticate the integrity of all subsequent firmware and software that will be loaded onto the system.

Modern computing systems are also often equipped with baseboard management controllers (BMC) which are specialized processors that monitor the physical state of the machine using sensors and communicate that to system administrators via independent communication channels. BMC firmware is highly privileged and allows for remote management and control, even when the system is shut down. A boot to BIOS, UEFI, a hypervisor or OS is not necessary as the BMC functions even if the server is shutdown.

The BIOS/UEFI or the BMC controller are the most critical component of a modern day computing device. Any compromise of the BIOS/UEFI or the BMC firmware code can give an attacker complete control over a system, allowing them to circumvent nearly all higher-level security safeguards. Moreover, once a device has booted up, an attacker can compromise other device functionalities that are present. Unfortunately, firmware attacks can be very difficult to identify and repair [3]. Thus, ensuring firmware security via prevention is critical but exceedingly challenging.

In this work, we address the problem of preventing firmware tampering before the secure booting, when a device is physically shipped from a manufacturer to a user (following a sale) or from a legitimate user to another user (for example, following a transfer, relocation or resale). In other words, we are concerned with protecting a device during transit from unauthorized modifications to firmware (whence the phrase "*Protection in Transit – PIT*"). To modify the firmware there needs to be either a way to write to the storage location of the firmware code or to physically replace the firmware code with a bad one. Here, our concern is specifically with the former method. We observe that such modification requires that the device is booted up (at least to a minimum state where firmware memory I/O is enabled). Thus, we recast this problem as one of ensuring that the first boot of the BIOS/UEFI or BMC following shipping is exclusive to the authorized downstream user. We assume that the manufacturer or legitimate user would not intentionally tamper with the device. We also assume that there is no physical tampering of the device (for example, opening the device and installing a malicious chip).

We realize this objective by having the device manufacturer implement a BIOS or BMC lock post-production and introduce a mechanism for user authentication by the device before self-unlock to ensure its use is only by the rightful authorized

2693-9177/24/$31.00 ©2024 IEEE
DOI 10.1109/QRS62785.2024.00064

user. Manufacturers of modern computing devices frequently use a tamper-proof micro-controller [4] as a hardware root of trust for a verified and reliable machine boot-up following software attestation principles. We employ such a hardware root of trust (HRoT) in the BIOS/UEFI or BMC boot process to perform user authentication, locking, and unlocking of the device. The locking of the device is executed at the BIOS or BMC level and the HRoT triggers the unlock after successful authentication.

Building upon the foundations laid by Project Cerberus [5], an open-source initiative aimed at establishing a hardware root of trust (HRoT) for server platforms, we enhance Cerberus capabilities to carve out a more resilient security protocol. While Cerberus effectively orchestrates secure attestations for firmware on devices, it does not inherently possess lock/unlock or user authentication functionalities. Our implementation of the PIT-Cerberus framework incorporates this lock/unlock mechanisms tied to an authentication protocol, allowing for more stringent control over the BIOS or BMC even before the boot-up phase. This is pivotal in preventing unauthorized access during a device's transit and ensures that the integrity and security of the firmware remain uncompromised from the point of departure to the point of receipt. AMI International, an industry leader in BIOS/BMC firmware, is currently beta testing the framework and plans to incorporate it in their production lines. We are in the process of open-sourcing our extended PIT-Cerberus library, confident in its ability to bolster the security landscape.

2. Related Works

Several potential attacks against conventional BIOS and EFI/UEFI firmware have been demonstrated by security researchers under laboratory conditions. Sacco and Ortega discuss the topic of BIOS infections and describe a proof-of-concept demonstration of a persistent BIOS malware infection [3]. According to the authors, detecting and eradicating BIOS infections is challenging because they reside in a section of the computer's memory that is inaccessible to standard antivirus or anti-malware software. Wojtczuk and Rutkowska have presented research on the security of Intel BIOS and described several vulnerabilities and attack vectors [6] that can be utilized to attack the BIOS such as exploiting firmware vulnerabilities, accessing the BIOS via hardware debugging tools, and employing software-based assaults such as buffer overflow attacks. A new type of malware known as System Management Mode (SMM) rootkits [7] has been introduced by Duflot and Pornin. In this paper, the authors explain the nature of SMM, its role in computer hardware, and the vulnerabilities that can be used to get access to SMM.

Firmware level malware has been somewhat more common in embedded systems. Using the HP-RFU vulnerability in LaserJet printers as a case study, A. Cui, M. Costello, and S. Stolfo demonstrate the development of a proof-of-concept printer malware capable of network reconnaissance, data exfiltration, and propagation to other devices [8]. They highlight the widespread nature of vulnerable embedded devices and the challenges in patching them, with only a small percentage of the vulnerable population being patched. The paper also emphasizes the limitations of firmware update signing and identifies vulnerabilities in third-party libraries found in firmware images. Overall, the findings underscore the need for effective host-based defense mechanisms to protect vulnerable embedded systems.

PsycoB0t [9], a notable router botnet, infiltrated the firmware of about 85,000 DD-WRT home routers, turning them into instruments for conducting severe network-disrupting Distributed Denial of Service (DDoS) attacks. Barnaby Jack demonstrated the unauthorized extraction of cash from ATMs through the modification of their firmware, a technique infamously known as "jackpotting" [10]. Charlie Miller uncovered serious risks within the firmware of certain Apple laptop batteries that could potentially lead to malfunctions or overheating [11]. Employing PostScript [12], a ubiquitous language in electronic and desktop publishing, Costin showcases the susceptibility of certain Lexmark printers to memory inspection and arbitrary modifications, which can lead to exposure of sensitive data or disruption of device functionality. Kevin Fu's groundbreaking work in medical device security highlights the perilous reality of exploiting embedded devices [13], with his real-world attacks on an implantable cardioverter defibrillator and an automated external defibrillator illuminating these life-threatening vulnerabilities. These instances underscore the crucial need for robust firmware security across diverse devices, as firmware forms the base code controlling hardware, making successful infiltration by an attacker profoundly dangerous.

The integrity of the System BIOS is critical for ensuring the security and functionality of computer systems, particularly during their transit through the supply chain. Unauthorized modifications, such as malicious firmware updates and firmware modifications, pose significant threats [14]. Malicious actors exploit vulnerabilities or weak security measures to alter the BIOS, introducing malware or Trojans that can lead to data breaches, device malfunctions, or system takeovers [15]. These threats are exacerbated by man-in-the-middle attacks [16] and supply chain attacks [17], which intercept or tamper with firmware updates or implant malware, respectively. Such attacks exploit the low-level operation of firmware, making detection and mitigation difficult, and represent a persistent security challenge [18][19][20].

Hardware-based trusted computing utilizes Trusted Execution Environments (TEE) and secure elements like the Trusted Platform Module (TPM) [21] to enhance security. TPMs, which have been included in computers for over a decade, establish a root of trust in a secure cryptographic core, protecting keys and credentials even if the OS is compromised. The latest standard, TPM 2.0, is present in most modern computers and supports various elliptic curve signature schemes, essential for certain core security services [22]. However, vulnerabilities like CVE-2020-10713 in the GRUB2 bootloader and others in bootloaders from Eurosoft, New Horizon Datasys, and CryptWare have shown that attackers can bypass Secure Boot to execute malicious code. These vulnerabilities can be somewhat

mitigated by blacklisting the affected bootloaders in the UEFI Secure Boot Forbidden Signature Database (DBX), which can be updated via UEFI firmware updates or Windows Update. Beyond bootloaders, attackers can target deeper UEFI components, exploiting specific vulnerabilities for targeted attacks. Recent research has uncovered numerous high-impact vulnerabilities (CVE-2022-28858, CVE-2022-36372, CVE-2022-32579, CVE-2022-27493 and CVE-2022-33209, CVE-2022-23930, CVE-2022-31644, CVE-2022-31645, CVE-2022-31646, CVE-2022-31640 and CVE-2022-31641) in UEFI firmware components, indicating an ongoing risk despite advancements in UEFI security technologies [23].

The latest TPM chips support Windows Secure Boot, ensuring the OS only boots if its hashes match those in the chip, preventing rootkit interference. However, TPM doesn't ensure complete security against keyloggers or phishing attacks [24]. Microsoft's installation criteria for Windows 11 include a mandatory TPM 2.0 module to enhance security features. Despite this, new Registry modifications [25] have been identified, enabling users to circumvent not only the TPM 2.0 stipulation but also the minimum memory and secure boot prerequisites for the operating system.

Ian Haken shows how attacks bypasses BitLocker by exploiting systems without pre-boot authentication and using a mock domain controller [26]. It requires the target machine to be domain-joined and have had a domain user log in previously. The attacker sets up a fake domain, tricks the system into accepting a new password due to an "expired" password scenario, and corrupts the local credentials cache. This allows offline login with the new password, giving the attacker access to all user data and the ability to install malware. The method is simple, requires physical access, and quickly circumvents BitLocker without complex tools, representing a serious security risk.

3. Threat Model

In this section we describe the threat model. We begin by identifying the key entities within the PIT-Cerberus framework. The Hardware Root of Trust (HRoT) is a critical component that leverages the Project Cerberus embedded framework. Devices such as laptops, workstations, and commercial servers, which integrate BIOS/BMC functionalities, constitute the DEVICE. The HRoT together with the DEVICE comprise the PRODUCT that is shipped to the USER. The USER is defined as an authorized individual with access rights to the DEVICE. Additionally, the COMPANY Server (or simply the SERVER), a part of the COMPANY's PIT-Cerberus framework infrastructure, is responsible for overseeing the DEVICE's locking and unlocking mechanisms.

We assume that the HRoT processor is tamper-proof and trusted. The COMPANY is trusted and there is no malicious insider threat at the COMPANY. The COMPANY programs the HRoT processor with the PIT-Cerberus and related libraries and data in a secure manner. (Note that any modern computing device that employs a root of trust for software attestation, also makes a similar assumption about the root of trust.) We also assume that the SERVER is honest and not curious. Any confidential information that is stored on the SERVER is protected against confidentiality and integrity breaches. We also assume that the HRoT can operate independent of the DEVICE and can setup and sustain communication with SERVER.

The USER will need to use a second computing device (for example, a smartphone or a laptop) to send and receive information to/from SERVER and send information to the PRODUCT during initial boot up. We assume that the USER trusts this device.

We assume that an adversary is able to sniff on the communication channel between the HRoT and the SERVER; it can replay messages on this communication channel, tamper with the messages and can also insert messages on this communication channel. The threat model is particularly concerned with the attacker's capabilities, especially during the period when the PRODUCT is in transit from the COMPANY to the boot-up of the DEVICE by the USER. This transit phase is the un-trusted zone. Within this scenario, attackers may attempt to intercept and analyze packets exchanged during the DEVICE unlocking process, potentially gaining access to sensitive information. Replay attacks represent another threat, where attackers could delay or resend packets to mislead the HRoT, USER, or SERVER. Additionally, there is a risk of attackers injecting false information, aiming to disrupt ongoing services. PIT-Cerberus is not designed to protect against such denial-of-service attacks. A significant threat is posed by man-in-the-middle attacks, where an attacker impersonating the SERVER or a legitimate USER could try to compromise the firmware integrity or gain unauthorized access by bypassing security measures.

The PIT-Cerberus framework is not designed to protect the PRODUCT from physical tampering during transit. Such an attack can for example, replace the HRoT with a microcontoller of the attacker's choosing or embed a hardware Trojan. Such protection is difficult to implement. If needed, we can assume that when a device is shipped from the COMPANY to the USER, it is protected in a package using tamper-proof seals [27][28]. This type of attacks would not only cause damage to the PRODUCT but also alert the USER, which is contrary to the attacker's objectives.

The Federal Information Processing Standard (FIPS 140-2) [29] outlines four security levels for cryptographic modules, ranging from minimal physical protection at Level 1 to comprehensive environmental safeguards and tamper detection at Level 4. An example of the highest security standards is IBM's 4758 PCI cryptographic adapter [30], which adheres to FIPS 140-1 Level 4, equipped with internal tamper detection and sensors for environmental attacks.

Additionally, we consider the COMPANY Server a trusted zone because the COMPANY employs its own Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), alongside security policies and personnel, to guard against any form of privilege escalation, information disclosure, or DoS-type attacks [31]. Furthermore, the protocol design and

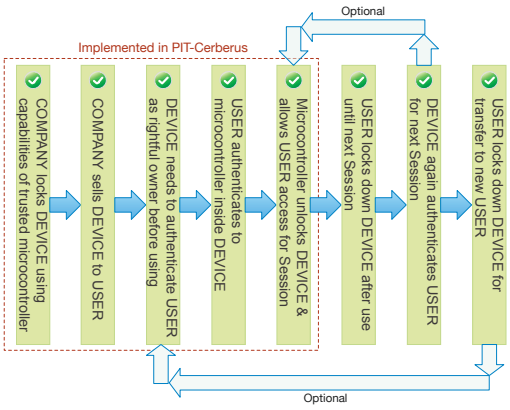

Figure 1. High-Level Workflow of PIT-Cerberus Protocol

the implementation of various protocols, such as the key size of Advanced Encryption Standard (AES), prime modulus of Elliptic-curve cryptography (ECC), curve selection for ECC, and the Digital Signature Algorithm (DSA), are meticulously chosen and programmed within a Sensitive Compartmented Information Facility (SCIF) at the COMPANY. So, we assume that any form of physical tampering and the analysis of side-channel information, including power, timing, and electromagnetic radiation, are impractical for attackers.

4. Proposed Approach

Our approach to solving the problem of protection in transit is to implement a BIOS-level lock/unlock facility before the secure boot implemented by a hardware root of trust – HRoT – microcontroller. The same protocol can be implemented at the BMC level and we use the term BIOS lock to also include locking of the BMC firmware. An overall workflow of PIT-Cerberus Protocol is shown in Figure 1.

This protocol is implemented in the HRoT that is embedded with the DEVICE. Prior to starting the boot-up of the BIOS, the HRoT establishes a secure state, which verifies the user and product information. This process occurs before the full BIOS boot-up and OS loading. We assume that the HRoT processor is able to establish a rudimentary communication with the outside world without relying on the networking capabilities offered by the host machine.

The PIT-Cerberus protocol terminates with success if HRoT has the confidence that the correct USER has initiated an unlock request. The HRoT relies on the SERVER to validate if a correct USER has initiated the unlocking process. The PIT-Cerberus protocol implements a novel challenge-response authentication scheme involving the HRoT, USER and SERVER for this validation. One important feature of our protocol is that this authentication does not require the HRoT to be programmed with the legitimate user's identity or shared secret. Thus, if the device is resold, the HRoT can perform the same interaction with the next user (shown as the lower "Optional" flow in Figure 1. If the next user is established as the legitimate owner of the device, the user is granted access to boot-up the BIOS. Once verification is complete, the device turns on, and the BIOS continues with the hardware initialization process. We assume that secure attestation of firmware is performed as usual after the unlock. In the following, we discuss the PIT-Cerberus protocol in more details.

4.1 The PIT-Cerberus Protocol

The protocol in shown in Figure 2. There are 4 major entities in the protocol: (i) HRoT: It is the tamper-proof microcontroller that acts as the hardware root of trust. It is the main component that will lock or unlock the BIOS of the DEVICE. PIT-Cerberus protocol is implemented in the HRoT processor, (ii) DEVICE: It is the entity (with BIOS/BMC) being protected in transit via PIT-Cerberus, (iii) USER: USER is the individual authorized to operate the DEVICE. USER has established a trust relationship with the COMPANY in Figure 1, and (iv) SERVER (COMPANY Server): It works on behalf of the COMPANY to mediate the HRoT - USER authentication. PRODUCT is the asset that COMPANY shipping to an USER which is a DEVICE with HRoT. The HROT and USER communicate via the internet with the SERVER to lock or unlock the DEVICE. We assume that this channel ensures the authenticity of endpoints and integrity of messages.

The PIT-Cerberus protocol is divided into two main parts: the Locked State and the Unlocked State. Under the Locked State, the HRoT is operational but BIOS has not been loaded and the remaining part of the DEVICE is non-functional in the PRODUCT. The Locked State is further divided into various sub-states.

1) Locked State: The "Locked State" is the initial state of the PRODUCT when it leaves the manufacturer. To put the DEVICE of the PRODUCT to the Locked State, the SERVER creates a unique PRODUCT registration ID, REG.ID, programs it into the HRoT, and also stores it locally. The SERVER also allocates this unique REG.ID to a particular authorised USER. The DEVICE is locked by BIOS or BMC and the DEVICE becomes unresponsive to typical USER inputs. This Locked State has several sub-states that outline the conditions and steps needed to transition the DEVICE into the Unlocked State. Each sub-state serves as a checkpoint in verifying the authenticity of the USER trying to access the DEVICE. When a DEVICE is powered on, by default it is in the Locked State but the HRoT inside the PRODUCT transitions from one sub-state to another.
   a) In the Locked State, the HRoT first enters the Key Generation sub-state in which it first generates an ECC (Elliptic Curve Cryptography) private key (d_A) - public key (q_A) pair, where, $q_A = d_A \times G$. G is the base point of the chosen elliptic curve. The Key Generation sub-state together with the Key Exchange sub-state (described next) forms a Elliptic-curve Diffie-Hellman (ECDH) key agreement protocol [32].
   b) Once the key pair is successfully generated, the HRoT enters the Key Exchange sub-state. In this state, the

HRoT sends the public key (Q_A) to the SERVER. The SERVER generates a private key d_B, computes a public key $q_B = d_B \times G$ and an AES secret key $S = d_B \times q_A$. The SERVER transmits its public key to the HRoT, which then computes the same AES secret key S as $S = d_A \times q_B = d_A \times \{d_B \times G\} = d_B \times \{d_A \times G\} = d_B \times q_A$.

c) At the end of the ECDH protocol, the HRoT enters the Idle sub-state waiting for a USER to initiate an unlock request.

As illustrated in Figure 2, all steps of the locking procedure are performed within a Sensitive Compartmented Information Facility (SCIF) at the COMPANY.

2) Unlocked State: The "Unlocked State" represents the phase where the DEVICE, initially in a locked position, is unlocked and allowed to boot fully, enabling the USER to use it as intended. The PRODUCT is shipped to a legit USER and, a secure connection has been established between USER–SERVER and SERVER–HRoT. To raise an unlock request, USER has to log into SERVER with an multi-factored authentication schema.

a) Once a USER makes an unlock request to SERVER and the DEVICE is powered on, the SERVER first

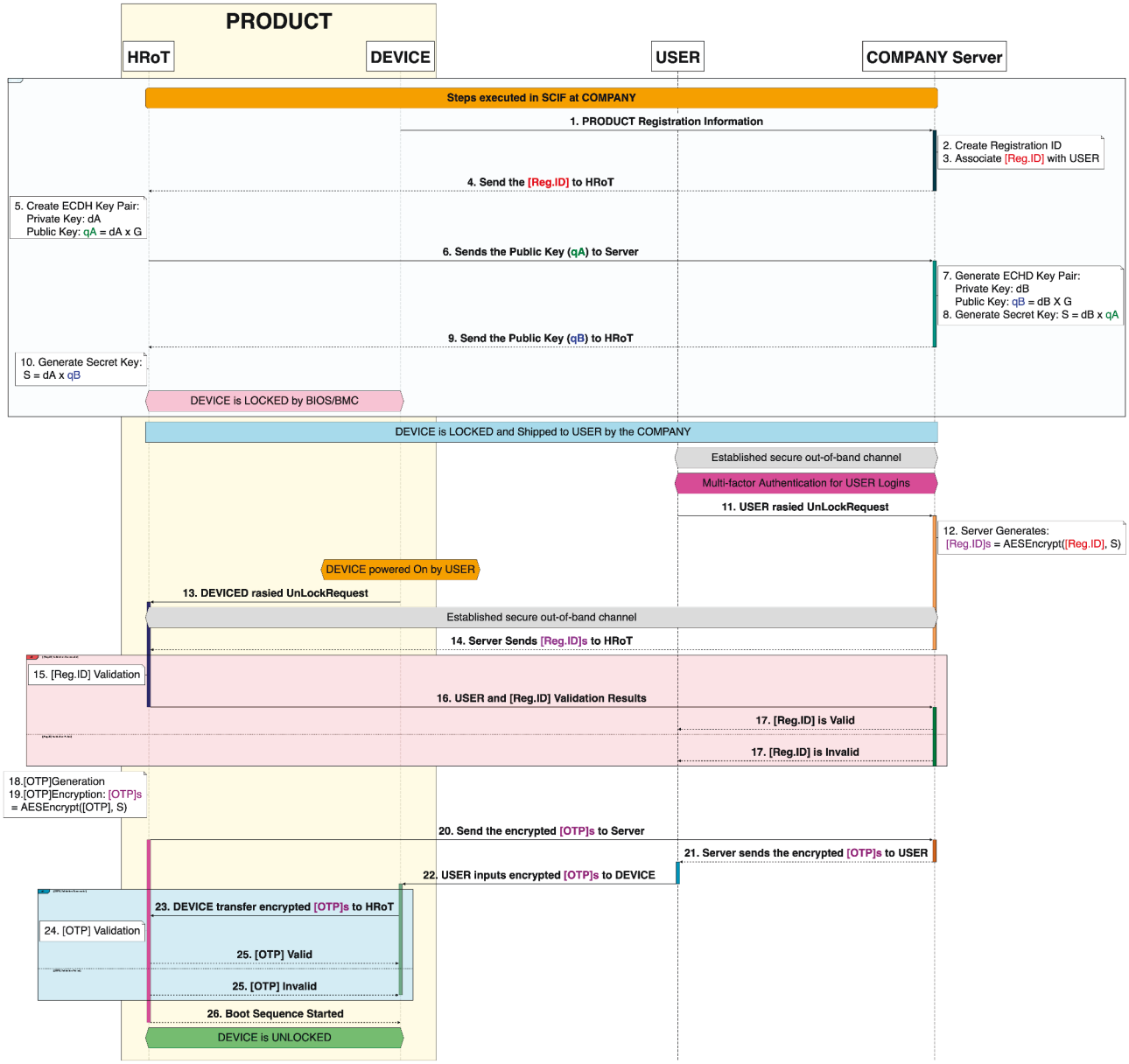

Figure 2. PIT-Cerberus Protocol for Locking and Unlocking DEVICE

sends REG.ID$_S$ to HRoT. The HRoT then initiated Reg.ID$_S$ Validation state where it decrypts the REG.ID$_S$ and validated against the stored REG.ID. This state validates that the USER is correctly identified by the HRoT. Then the HRoT enters the OTP Generation sub-state. The HRoT creates a one-time password (OTP) for the USER, and encrypts OTP using AES-GCM [33] with the secret key S to generate OTP$_S$. The HRoT sends these encrypted value to the SERVER and enters the OTP Exchange sub-state.

b) The HRoT continues to wait in the OTP Exchange sub-state until it receives another response from the USER.

c) The SERVER, upon receipt of OTP$_S$, looks up the correct USER to send the the encrypted OTP$_S$ via an out of band channel (secure email or mobile phone). We assume only the authorized USER has access to this communication facility between the SERVER and the USER.

d) On receiving OTP$_S$, the USER enters it into the HRoT, which then enters the OTP Validation sub-state. Upon successful validation of the OTP by the HRoT, it triggers the DEVICE Unlocked State which is nothing but initializing the booting process in the DEVICE.

To manage the boot process of a BIOS/BMC device using a microcontroller such as Microchip CEC1702 as a Hardware Root of Trust (HRoT), the process involves initializing the HRoT upon device power-up to conduct self-tests and initiate secure boot procedures. The HRoT verifies the integrity and authenticity of the USER's legitimacy. If the firmware passes these checks, the HRoT allows the boot process to proceed, handing over control to the BIOS/BMC for hardware initialization and operating system launch. These actions leverage the HRoT's secure boot capabilities to either permit or prevent the device from reaching an operational state, ensuring that only authenticated and integrity-checked firmware can execute, thus providing a robust security measure against unauthorized access.

5. PIT-CERBERUS IMPLEMENTATION

The PIT-Cerberus protocol has been implemented through an open-source embedded framework, known as Project Cerberus [5]. Originally conceived by Microsoft, Project Cerberus is a framework for hardware root-of-trust that is compliant with the NIST 800-193 standards, with an unclonable identity. This platform enforces a secure boot process for DEVICE firmware and provides a secure mechanism to attest to the state of the DEVICE firmware.

PIT-Cerberus is distributed as set of APIs. These APIs encapsulate the various functionalities extended by the HRoT processor to lock and unlock the BIOS/BMC of the DEVICE. The design of these APIs is guided by the principle of enabling Cerberus to lock and unlock the BIOS in a secure and efficient manner. The API consists of the following functions:

(i) `lock()`: This function uses the internal `pit_keygenstate()`, `keyexchangestate()`, `pit_secretkey()` to perform all the operations needed to lock the BIOS and loads the secret key in a 32-byte empty array (*secret*).

(ii) `pit_keygenstate()`: This function is responsible for generating the ECC key pair for HRoT. It loads the length of the key in *key_length*, initializes private key in *privkey*, public key in *pubkey*, and the state of the HRoT in *state* parameter.

(iii) `keyexchangestate()`: Exchange the public key of HRoT and SERVER. On success, `keyexchangestate` will initialize *pubkey_cli* with the HRoT's public key and load the *pubkey_serv* variable with a public key received from the SERVER.

(iv) `pit_secretkey()`: It takes ECC private key from HRoT and SERVER, computes the secret key and loads in *secret* parameter.

(v) `unlock()`: Do all the unlocking operations. These operations generate OTP, encrypt it, send to SERVER and validate the OTP.

(vi) `receive_product_info()`: This function receives the product information from the SERVER and assigns it to some parameters, such as encrypted registration ID (*EncryptedProductID*), tag (*EncryptedProductIDTag*), registration ID size (*ProductIDSize*), an initialization vector (IV) used for encryption (*aes_iv*), and size (in bytes) of the vector in (*aes_iv_size*).

(vii) `pit_connect()`: It initiates a connection to designated SERVER. It takes the port address of the SERVER as a input and returns an integer pointing to the file descriptor (socket) which can be used to send/receive from the SERVER.

(viii) `pit_OTPgen()`: This function generates a random string representing *OTP*. Additionally, encrypts that *OTP* using the secret key (*secret*) as key for the AES-GCM and loads in *encOTP* parameter.

(ix) `pit_encryption()`: It encrypts a message using a secret key. This function takes secret key, message and use AES-GCM method to encrypt the message and loads into *ciphertext* parameter.

(x) `send_unlock_info()`: This function sends the encrypted OTP (*encOTP*) to the SERVER which will be later sent to USER.

(xi) `pit_OTPValidation()`: This function validates the encrypted OTP (*encOTP*). It does this by taking encrypted *OTP* (*OTP$_S$*) as input, decrypting it, and comparing it against the original OTP (*OTP*). If valid, the function returns 1 and the parameter *result* will hold *true* otherwise *false*.

(xii) `pit_decryption()`: This function takes encrypted message (*ciphertext*), secret key (*secret*) as input, decrypts it and loads the message to the provided *plaintext* buffer.

Figures 3 and 4 describe the sequence of the function calls to implement the PIT-Cerberus protocol.

In the SERVER implementation, we leverage the cryptography

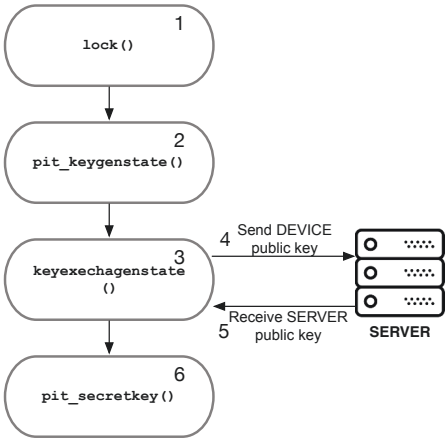

Figure 3. Function call sequence in Locked State of "*pit*"

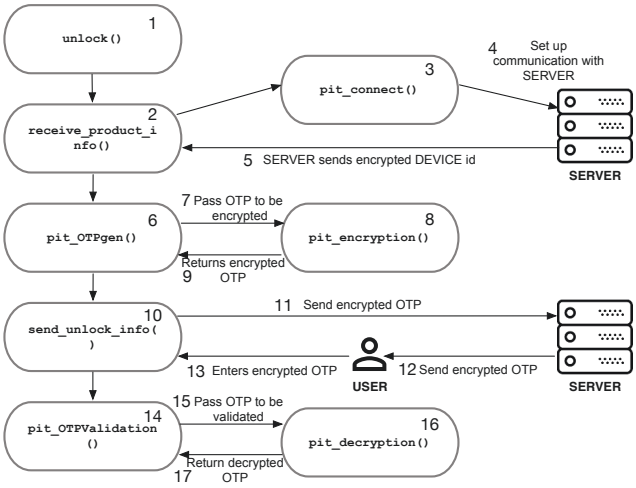

Figure 4. Function call sequence in Unlocked State "*pit*"

library for Python, employing three key modules — Elliptic Curve Cryptography (ECC), serialization, and Advanced Encryption Standard - Galois/Counter Mode (AES-GCM). The ECC module facilitates the generation of a public-private key pair for the SERVER and furnishes a method to perform the Elliptic-curve Diffie–Hellman (ECDH) protocol, enabling the generation of a shared secret using the public key of another party. This module is indispensable to our implementation, although any library that supports the generation of an elliptic curve key pair and the ECDH protocol — compliant with NIST standards [35] — would suffice.

The serialization module is tasked with key distribution and reception. Prior to the SERVER's public key transmission to HRoT, it is encoded into DER format via the serialization module. Similarly, HRoT's public key undergoes DER serialization before being sent to the SERVER. Upon receipt, the SERVER employs the serialization module to convert the DER-encoded public key from HRoT into a format compatible with the cryptography library, specifically an EC key. Libraries providing key serialization that adhere to NIST standards are suitable for this process.

The AES-GCM module is responsible for data encryption and decryption. AES-GCM is the chosen form of AES owing to its native support in Cerberus. Coupled with the shared secret derived from ECDH, the AES-GCM module serves as the key for encryption and decryption, ensuring secure and coherent communication between the SERVER and HRoT. For DEVICES requiring Cerberus functionality, the core set of source code delivers a suite of foundational features that can be integrated and ported. The provided code, largely device-agnostic, defines the required abstraction layers. Therefore, we tailored Cerberus by introducing a new library and APIs encompassing the lock and unlock mechanisms for BIOS/BMC. These modifications were instituted within the core module, designated as *PIT-Cerberus*.

A detailed documentation about the APIs and "*pit*" library is already publicly available in GitHub [36].

6. Evaluation & Discussion

Our primary focus with this evaluation was to evaluate the PIT-Cerberus framework's performance with the library. We developed various experimental test scenarios to evaluate our framework with the library's performance. We ran our experiments on 2 virtual Linux servers. The client side (assumed as HRoT) has a 5000 MHz 12th Gen Intel(R) Core(TM) i7-12700K processor, x86_64 architecture, 20 cpus and Server in Inter(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz, x86_64 architecture, 12 CPU(s).

In order to evaluate that the PIT-Cerberus framework is working as it is supposed to, we used server level Project Cerberus with "*pit*" library with a C socket as a client and a SERVER implemented in Python. As designed and expected, our protocol delivered the anticipated results, with successful operations observed across both HRoT that uses PIT-Cerberus framework, and the SERVER. Figures 5 and 6 provide visual confirmation of these outputs.

As depicted in Figure 5, the DEVICE (BIOS) initiates in a locked state as the SERVER, equipped with its own distinct set of ECC key pair as shown in Figure 6, establishes a connection with HRoT. Following the successful exchange of ECC key pairs between HRoT and the SERVER, computation of the secret key is initiated.

We designed a simple two-step verification protocol for login. Once the USER logs in with their credentials, they can raise an unlock request on the SERVER. Upon receiving an unlock request from the USER on SERVER, SERVER commences communication with the HRoT. The unlocking process initiates when the SERVER transmits the encrypted PRODUCT registration ID (REG.ID_S) to HRoT. Subsequently, HRoT decrypts and validates this PRODUCT registration ID, the output of which is displayed in the HRoT terminal. Upon successful Product ID validation, HRoT generates an one-time password OTP. This OTP is encrypted using AES-GCM with the secret key (S), and dispatched to the SERVER, which

then forwards this encrypted OTP (OTP_S) to the USER's email address (as illustrated in Figure 7). The USER, upon receiving the OTP_S, inputs it into the HRoT terminal. HRoT then decrypts this OTP_S utilizing the secret key (S), and performs a validation check. If the validation is successful, HRoT proceeds to unlock the DEVICE, as demonstrated in Figure 5.

In addition to the evaluation of the PIT-Cerberus library with the core of Project Cerberus, we also carry out extensive testing ("Compatibility Testing") to validate their compatibility with different SERVER libraries that use for the generation of ECC key pairs and shared secret key. Different SERVER environments employ varied libraries for the ECC key pair generation and shared secret key computation, each having its own intricacies and idiosyncrasies. It was crucial to confirm that the PIT-Cerberus library interacts seamlessly with these different SERVER libraries without any interoperability issues. To achieve this, we conduct a series of interoperability tests in multiple SERVER environments, each employing different libraries for key pair generation and shared secret key computation. We have tested with commonly used libraries, such as OpenSSL [37], libsodium [38], and Cryptlib [39], which are widely accepted for their robustness and compliance with industry standards. In each case, we confirm that the PIT-Cerberus library that has been used in HRoT, is able to correctly and efficiently compute and exchange ECC key pairs and shared secret keys without any compatibility issues. We also ensure that the computed keys adhered to the relevant NIST standards, and the elliptic curve cryptography functionality is up to the mark.

Furthermore, we have validated that the PIT library could correctly interpret the serialized keys received from the SERVER and could successfully execute the Elliptic-curve Diffie–Hellman (ECDH) operation to generate a shared secret key. We have also checked the successful encryption and decryption of data using the derived shared secret key. Through this extensive testing, we have validated that the PIT-Cerberus library is compatible with a wide range of libraries for generating ECC key pairs and shared secret key, thus enhancing the universal applicability and adaptability of our protocol.

We show a few results of our validation process in a proof of concept incorporating an OpenSSL-based SERVER. The objective is to demonstrate the universal capabilities of our designed library in handling operations regardless of the library types. In the execution of our model, HRoT transmits the public key to the SERVER, encoded in Distinguished

```
pit_crypto: test_OTPgen
pit_crypto: test_OTPvalidation
pit_crypto: test_decryption
pit: test_pit_lock
Device is Locked.
pit: test_pit_unlock
User initiated Unlock Request......
PRODUCT ID Validation Successful. pid_status is: 1
OTP Generation and Encryption Successful.
Please Enter your OTP:
b'\x03hs\x7f ?\x86\xfc\xa9\xfa\xb3\x12\xec\x1cGa1\x07^\xf3\x04\xca!\xa9]\x11\xea0wpb?1\x13\x16?\xf8\xc1\xac\x11\xe6\xda\x8c7\x07\xce\xedU
\x1d\xc9\tOaH\xa3\xdf\x98i\xb5AD\x1c?=V{\xef_\x0bx\x80\xcex\xb3\xed\x0e]9\x9f\xda\xe1W\x8e\xa2\r\xdb\xd54\xce\xdd\xe3\xe8q\x15\x00/b[0\xf
0\xc8\xc7\xee,\xce\x9f\x1b\xe6\xbd,\xcdY&\x1aj^c3\xbd\xef\x80\xc1\xd7~g\xc0\xcf\x1e'
Encrypted OTP sent to Server.
OTP Validation Successful. pid_status is: 1
```

Figure 5. Output from HRoT Terminal

```
tallahassee:~/Rakesh$ python3 pid-server.py
Generated: Server ECC Key Pair.

Server X value : 768465705340819100253629315231074600515073783807449255146945688342708940824
Server y value : 109252678345389310469317484089853095411478063296788366629630286751158294466709
listening
accepted
Connected by ('127.0.0.1', 40564)
Recived: Cerberus ECC Key Pair.

Client X value : 864774840645440664077226990236261786998423991788212449554964793995626422512 39
Client y value : 457020251591109096967518400286968162193238137950203812681101274813747646578 07

Generating shared secret..... Successful.
listening
listening
accepted
Connected by ('127.0.0.1', 34996)
[DEMO]: Encrypted OTP : b'\x03hs\x7f ?\x86\xfc\xa9\xfa\xb3\x12\xec\x1cGa1\x07^\xf3\x04\xca!\xa9]\x11\xea0wpb?1\x13\x1
6?\xf8\xc1\xac\x11\xe6\xda\x8c7\x07\xce\xedU\x1d\xc9\tOaH\xa3\xdf\x98i\xb5AD\x1c?=V{\xef_\x0bx\x80\xcex\xb3\xed\x0e]9
\x9f\xda\xe1W\x8e\xa2\r\xdb\xd54\xce\xdd\xe3\xe8q\x15\x00/b[0\xf0\xc8\xc7\xee,\xce\x9f\x1b\xe6\xbd,\xcdY&\x1aj^c3\xbd
\xef\x80\xc1\xd7~g\xc0\xcf\x1e'

Enter your email: rakeshpodder3@gmail.com
```

Figure 6. Output from SERVER Terminal

Encoding Rules (DER) format. As is depicted in Figure 8 (top-right), the OpenSSL SERVER effectively interprets the public key delivered by HRoT. The DER format public key is subsequently outputted in the SERVER terminal, further validating the successful exchange of cryptographic keys between HRoT and the SERVER.

As further demonstrated in Figure 8 (top-left), it can be observed that OpenSSL is proficient in interpreting both public and private keys generated by the SERVER. This empirical evidence substantiates that cryptographic library, as implemented in our framework, is compatible with OpenSSL and by extension, compliant with the National Institute of Standards and Technology (NIST) standards for cryptographic algorithms. Therefore, our methodology exhibits a broad spectrum of interoperability, ensuring compatibility with any cryptographic systems that adhere to NIST standards. The evidence provided in Figure 8 (bottom) offers definitive proof that, by employing OpenSSL in conjunction with our Python SERVER, identical secret keys can be produced. This not only demonstrates the robustness and reliability of our key generation process, but also unequivocally confirms the compatibility of our implementation with systems conforming to NIST standards. As such, we can assure the broader applicability of our system, facilitating secure and effective communication between diverse cryptographic frameworks that adhere to these industry-accepted norms.

Apart from validations and compatibility tests, we tested the protocol's exceptions and failures handling capability thoroughly. For this we designed various test scenarios. Table 1 shows some scenarios of our testing. Its all during the unlocking phases. We also tested out when the DEVICE is locked by USER and shipped to another USER, the process will be started from starting phase of key_gen_state to the unlocking_state.

We have successfully modified the Project Cerberus embedded framework, giving birth to PIT-Cerberus for HRoT, a system equipped to secure PRODUCT during transit by locking the DEVICE (with BIOS/BMC). PIT-Cerberus boasts the necessary cryptographic capabilities that a microcontroller or microprocessor requires for BIOS locking. Our work, including the "*pit*" library, is made publicly available in our GitHub repository [36]. Additionally, a comprehensive set of instructions for operating PIT-Cerberus has been provided, further promoting the accessibility and usability of our solution.

Figure 7. Encrypted OTP Sent to Email

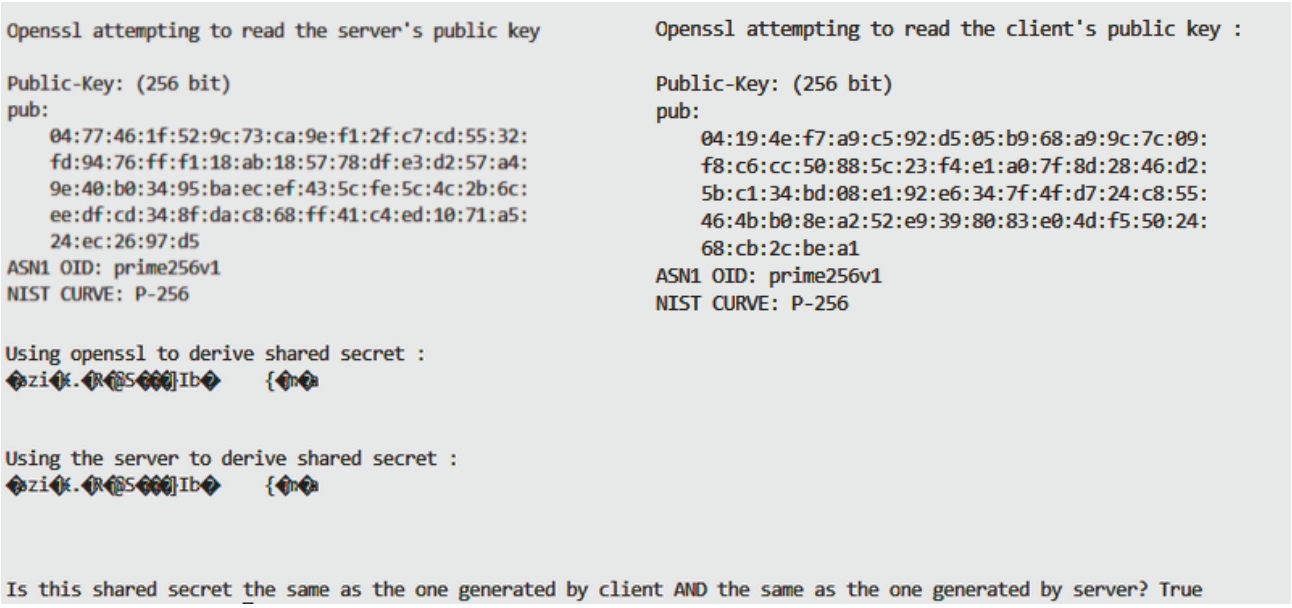

Figure 8. SERVER (top-left) & HRoT's (top-right) public key in DER format, (bottom) secret key of OpenSSL and Python SERVER

Table 1. Protocol's exceptions and failures handling capability in various test scenarios

Test Scenarios	Description	Handling	State
Server unavailability	The Server loses connections during locking or unlocking	The program will wait until the timer specified time. If the connection is not back, it will revert back to the previous state and throws a timeout exception.	Lock state
REG.ID validation fails	The encrypted REG.ID$_S$ fails to validate due to wrong REG.ID or empty.	If the encrypted REG.ID$_S$ cannot be validated, the program throws a REG.ID validation failure error and reverts back to the lock state	Lock state
OTP validation fails	The encrypted OTP$_S$ fails to validate due to wrong OTP or empty generated by HRoT.	If the encrypted OTP$_S$ cannot be validated, the program throws a OTP validation failure error and reverts back to OTP_gen state.	OTP_gen state
Unable to send OTP$_S$ to USER email	The encrypted OTP$_S$ fails to deliver to the USER due to wrong connectivity issues.	If the encrypted OTP$_S$ is not available to the USER, the HRoT will wait until the specified timer expires, then throw a timeout exception and revert back to the OTP_gen state.	OTP_gen state

7. SECURITY ANALYSIS

In the event of attacks that could lead to SERVER or service unavailability, we have implemented several exceptions and error handling mechanisms within the program to prevent unwanted interruptions or crashes during the unlocking procedure. We will not consider such attacks in the security analysis.

During the unlocking process, attackers can monitor and sniff on the communication channel established between the HRoT–SERVER and SERVER–USER. By doing so, they can capture encrypted PRODUCT registration ID (REG.ID$_S$) sent from the SERVER to HRoT upon an unlock request initiated by the USER. Additionally, attackers can intercept and steal login credentials when the USER attempts to log into the SERVER, potentially gaining unauthorized privileges. Furthermore, when HRoT sends an encrypted OTP to the SERVER, and the SERVER forwards this encrypted OTP (OTP$_S$) to the USER, attackers monitoring the communication channel could obtain the encrypted OTP (OTP$_S$) and misuse it for improper authorization access to the PRODUCT.

Another potential threat is replay attacks, where the attacker does not need to decrypt the message but merely reuses it. For instance, an attacker could capture a packet during USER logging into a system and replay it to gain unauthorized access without needing the USER's password. This allows the attacker to impersonate a valid USER, tricking the SERVER into sending the OTP$_S$ to a malicious USER. This constitutes a man-in-the-middle (MitM) attack, where the attacker impersonates USER to perform privilege escalation through sniffing and replaying data. An attacker could also capture the REG.ID$_S$ and replay them back to the HRoT to impersonate the SERVER and launch an MitM attack.

The protocol we've designed and implemented is robust against such attacks. For instance, when the SERVER transmits the REG.ID$_S$, they are encrypted using the Advanced Encryption Standard - Galois/Counter Mode with a 256-bit key (AES-GCM 256 Key) encryption scheme. AES-GCM is resistant to several types of attacks including *Known Plaintext Attack (KPA)*, *Chosen Plaintext Attack (CPA)*, *Chosen Ciphertext Attack (CCA)*, and *Ciphertext-Only Attack (COA)*. The design of this algorithm ensures that knowing a *plaintext-ciphertext* pair does not significantly aid in deducing the encryption key, as the relationship between *plaintext* and *ciphertext* yields minimal useful information. Additionally, with a 256-bit key size, AES-GCM-256 is currently infeasible to break through brute-force attacks, considering the 2^{250} possible key combinations.

However, to address advanced threats like the *Adaptive Chosen-Plaintext Attack (ACPA)* and *Adaptive Chosen-Ciphertext Attack (ACCA)*, where the attacker adapts their choices based on feedback from previous encryptions or decryptions, we employ Elliptic Curve Diffie-Hellman (ECDH) with a 256-bit or higher prime modulus for the elliptic curve. This is used to generate a secret key for AES-GCM encryption. In every session or iteration, this secret key is randomly generated, ensuring a new secret key for subsequent encryptions. Utilizing ECDH-256 for key exchange allows for the generation of a new AES-GCM-256 key for each session or even each message, greatly bolstering security. This method prevents key reuse and significantly complicates any attempt by attackers to compromise the communication. The synergy of AES-GCM-256 and ECDH-256 offers extensive security measures, where AES-GCM-256 secures the confidentiality and integrity of communications, and ECDH-256 ensures secure key negotiation, safeguarding against eavesdropping and MitM attacks.

To mitigate replay attacks, where an attacker could reuse

captured REG.ID$_S$ to launch MitM attacks or disrupt workflow, we've incorporated Lamport timestamps to ensure the freshness and integrity of messages transmitted over the communication channel. Thus, when the SERVER sends the REG.ID to the HRoT, it's encrypted using our protocol's encryption scheme, which maintains encryption randomness and is coupled with a Lamport timestamp to preserve message freshness. Furthermore, as the HRoT establishes a connection with the SERVER, each connection utilizes a unique session key. Every message sent to or from the SERVER or HRoT undergoes validation checks for session integrity, timestamps, and encryption, effectively preventing replay attacks on the HRoT or SERVER.

Additionally, to thwart adversaries attempting to impersonate the SERVER and launch a MitM attack, we employ the Digital Signature Algorithm (DSA), which leverages keys derived from Elliptic Curve Cryptography (ECC). This approach effectively safeguards against MitM attacks, ensuring the authenticity of communications and protecting against impersonation and data integrity threats.

We implement multi-factor authentication (MFA) when the USER tries to log into the SERVER using an otp (generated and sent by the SERVER). This means that even if an attacker manages to sniff the login credentials, they would not be able to log into the SERVER, as the login otp is sent to the USER's registered email ID. Additionally, a session is established between the USER and the SERVER at the start of communications to ensure the freshness of messages and prevent replay attacks, significantly reducing the possibility of successfully impersonating a valid USER in a MitM attack.

While there is a possibility that an attacker might sniff the OTP$_S$ as the SERVER sends them to the USER, obtaining these OTP$_S$ would first require the attacker to log into the SERVER and initiate the unlock request. This process validates the legitimacy of the USER by checking the REG.ID with the HRoT, after which the HRoT generates the OTP and sends it to the SERVER. However, as previously mentioned, attacks on this process are considered infeasible. This indicates that the PRODUCT (via HRoT) is connected and in possession of a legitimate user. Therefore, intercepting the OTP$_S$ would not be beneficial for the attacker, as the OTP needs to be manually inputted into the PRODUCT to initiate DEVICE booting process. This layered security approach ensures that the authentication mechanism is robust against unauthorized access attempts.

8. CONCLUSION

The exponential growth in hardware integrated circuits (ICs) inevitably raises parallel concerns regarding device security. Through our research, we have addressed a critical aspect of this concern - the protection of products during transit through third-party mediums. The protocol we have devised successfully safeguards devices from the potential introduction of trojans - a principal reason for the increasing focus on Hardware Root of Trust (HRoT) in contemporary industrial discourse.

Our modified version of the Cerberus embedded framework, PIT-Cerberus, extends security capabilities to any microcontroller or microprocessor, enhancing device security significantly. The robustness of our solution lies in its ability to protect the device at its most vulnerable state - during transit - by ensuring that only a verified user can unlock and boot the device's BIOS, thereby thwarting any attempts at introducing trojans.

Looking ahead, we intend to further enhance the utility and security features of PIT-Cerberus. A notable direction of our future work involves porting the hardware-agnostic PIT-Cerberus to a microchip-specific I2C protocol, to establish serial communication for performing the Key Exchange Scheme. We remain committed to continuously refining our protocol to address an ever-evolving landscape of hardware security challenges, thus ensuring the safeguarding of devices in an increasingly interconnected world.

ACKNOWLEDGMENT

This work was partially supported by the U.S. National Science Foundation under Grant No. 1822118 and 2226232, the member partners of the NSF IUCRC Center for Cyber Security Analytics and Automation – AMI, NewPush, Cyber Risk Research, NIST and ARL – the State of Colorado (grant #SB 18-086) and the authors' institutions. 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 National Science Foundation, or other organizations and agencies.

REFERENCES

[1] J. Ganssle, *The firmware handbook*, 1st ed. Burlington, CO, USA: Elsevier, 2004.

[2] M. Krichanov and V. Cheptsov, "UEFI virtual machine firmware hardening through snapshots and attack surface reduction," in *2021 Ivannikov Ispras Open Conference (ISPRAS)*. IEEE, 2021, pp. 30–36.

[3] A. L. Sacco and A. A. Ortega, "Persistent BIOS infection," in *CanSecWest Applied Security Conference*, 2009.

[4] Y. Gui, A. S. Siddiqui, and F. Saqib, "Hardware based root of trust for electronic control units," in *SoutheastCon 2018*. IEEE, 2018, pp. 1–7.

[5] Microsoft. (2018) Project Cerberus. [Online]. Available: https://github.com/Azure/Project-Cerberus

[6] R. Wojtczuk and A. Tereshkin, "Attacking intel bios," *BlackHat, Las Vegas, USA*, 2009.

[7] S. Embleton, S. Sparks, and C. Zou, "SMM rootkits: a new breed of os independent malware," in *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, 2008, pp. 1–12.

[8] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *20th Annual Network & Distributed System Security Symposium*, 2013. [Online]. Available: https://doi.org/10.7916/D8P55NKB

[9] A. Maassen, "Network bluepill-stealth router-based botnet has been ddosing dronebl for the last couple of weeks," 2009. [Online]. Available: https://www.dronebl.org/blog/8
[10] B. Jack, "Jackpotting automated teller machines redux," *Black Hat USA*, 2010.
[11] C. Miller, "Battery firmware hacking," *Black Hat USA*, pp. 3–4, 2011. [Online]. Available: http://pirate-network.com/pocorgtfo/pocorgtfo11/batteryfirmware.pdf
[12] A. Costin, "Hacking mfps," in *The 28th Chaos Communication Congress*, 2011.
[13] S. Hanna, R. Rolles, A. Molina-Markham, P. Poosankam, J. Blocki, K. Fu, and D. Song, "Take two software updates and see me in the morning: The case for software security evaluations of medical devices." in *HealthSec*, 2011.
[14] S. Schmidt, M. Tausig, M. Hudler, and G. Simhandl, "Secure firmware update over the air in the internet of things focusing on flexibility and feasibility," in *Internet of Things Software Update Workshop (IoTSU). Proceeding*, 2016.
[15] Z. Basnight, J. Butts, J. Lopez Jr, and T. Dube, "Firmware modification attacks on programmable logic controllers," *International Journal of Critical Infrastructure Protection*, vol. 6, no. 2, pp. 76–84, 2013. [Online]. Available: https://doi.org/10.1016/j.ijcip.2013.04.004
[16] M. Conti, N. Dragoni, and V. Lesyk, "A survey of man in the middle attacks," *IEEE communications surveys & tutorials*, vol. 18, no. 3, pp. 2027–2051, 2016.
[17] J. F. Miller, "Supply chain attack framework and attack patterns," *The MITRE Corporation, MacLean, VA*, 2013.
[18] D. Cooper, W. Polk, A. Regenscheid, M. Souppaya *et al.*, "Bios protection guidelines," *NIST Special Publication*, vol. 800, p. 147, 2011.
[19] A. Fuchs, C. Krauß, and J. Repp, "Advanced remote firmware upgrades using tpm 2.0," in *ICT Systems Security and Privacy Protection: 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30-June 1, 2016, Proceedings 31*. Springer, 2016, pp. 276–289.
[20] A. Vasselle, P. Maurine, and M. Cozzi, "Breaking mobile firmware encryption through near-field side-channel analysis," in *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, 2019, pp. 23–32.
[21] C. Mitchell, *Trusted computing*. Iet, 2005, vol. 6.
[22] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger, "{TPM-FAIL}:{TPM} meets timing and lattice attacks," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2057–2073.
[23] L. Constantin. (August 12, 2022) New exploits can bypass secure boot and modern uefi security protections. CSO. [Online]. Available: https://www.csoonline.com/article/573395/new-exploits-can-bypass-secure-boot-and-modern-uefi-security-protections.html
[24] A. A. Zharkova, "Application encryption with trusted platform module to implement standards in windows 11 environment," in *2023 Seminar on Information Computing and Processing (ICP)*. IEEE, 2023, pp. 239–241.
[25] L. Abrams. (July 2, 2021) How to bypass the windows 11 tpm 2.0 requirement. BleepingComputer. [Online]. Available: https://www.bleepingcomputer.com/news/microsoft/how-to-bypass-the-windows-11-tpm-20-requirement/
[26] I. Haken, "Bypassing local windows authentication to defeat full disk encryption," *Black Hat Europe*, 2015.
[27] M. C. Lu, Q. X. Huang, M. Y. Chiu, Y. C. Tsai, and H. M. Sun, "Psps: A step toward tamper resistance against physical computer intrusion," *Sensors*, vol. 22, no. 5, p. 1882, 2022.
[28] T. Moran and M. Naor, "Basing cryptographic protocols on tamper-evident seals," *Theoretical Computer Science*, vol. 411, no. 10, pp. 1283–1310, 2010.
[29] F. Pub, "Security requirements for cryptographic modules," *FIPS PUB*, vol. 140, pp. 140–2, 1994.
[30] T. W. Arnold, C. Buscaglia, F. Chan, V. Condorelli, J. Dayka, W. Santiago-Fernandez, N. Hadzic, M. D. Hocker, M. Jordan, T. E. Morris *et al.*, "IBM 4765 cryptographic coprocessor," *IBM Journal of Research and Development*, vol. 56, no. 1.2, pp. 10–1, 2012.
[31] A. Bakshi and Y. B. Dujodwala, "Securing cloud from ddos attacks using intrusion detection system in virtual machine," in *2010 second international conference on communication software and networks*. IEEE, 2010, pp. 260–264.
[32] R. Haakegaard and J. Lang, "The elliptic curve diffie-hellman (ecdh)," 2015.
[33] M. J. Dworkin, "Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac," 2007.
[34] R. Podder and R. K. Barai, "Hybrid encryption algorithm for the data security of esp32 based iot-enabled robots," in *2021 Innovations in Energy Management and Renewable Resources (52042)*. IEEE, 2021, pp. 1–5.
[35] E. Barker, "Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms," *NIST special publication*, pp. 800–175B, 2016.
[36] R. Podder and J. Sovereign. Project cerberus with pit protocol. [Online]. Available: https://github.com/cryptoknight13/Project-Cerberus
[37] J. Viega, M. Messier, and P. Chandra, *Network security with openSSL: cryptography for secure communications*. O'Reilly Media, Inc., 2002.
[38] I. S. Consortium. (2013) Libsodium documentation: Introduction. [Online]. Available: https://libsodium.gitbook.io/doc/
[39] P. Gutmann, "Cryptlib encryption toolkit," 2008. [Online]. Available: https://cryptlib.com/downloads/manual.pdf