IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

CR-ATTACKER: Exploiting Crash-reporting Systems using Timing Gap and Unrestricted File-based Workflow

SEONG-JOONG KIM[1,2], Myoungsung You[2] and SEUNGWON SHIN[2]
[1]The Affiliated Institute of ETRI, Daejeon, South Korea
[2]KAIST, Daejeon, South Korea

Corresponding author: Seungwon Shin (e-mail: claude@kaist.ac.kr).

This paragraph of the first footnote will contain support information, including sponsor and financial support acknowledgment. For example, "This work was supported in part by the U.S. Department of Commerce under Grant BS123456."

ABSTRACT Software vendors widely adopt crash-reporting systems to automate the collection of crash reports, enabling efficient diagnosis and management of software failures. However, these reports often contain detailed memory snapshots of crashed processes, which may include sensitive user data (e.g., credentials and cryptographic keys). Ensuring the security of crash-reporting systems is, therefore, critical to prevent information leakage and potential exploitation. This paper analyzes the security of common crash-reporting system architectures and identifies two novel attack vectors: (i) a timing gap vulnerability during partial privilege de-escalation and (ii) a file-based workflow exploitation between crash-reporting system components. By leveraging these attack vectors, we demonstrate that unprivileged attackers can extract arbitrary memory contents from other processes or manipulate the behavior of crash-reporting systems, leading to information leakage and system compromise. To mitigate these threats, we propose practical defense mechanisms that effectively neutralize both attack vectors, thereby enhancing the overall security of crash-reporting systems. We validate our findings through real-world evaluations on widely used open-source crash-reporting systems, which resulted in the discovery of four new CVEs related to ASLR bypass, arbitrary code execution, and denial-of-service (DoS) attacks. These findings highlight the urgent need for strengthened security measures in modern crash-reporting systems.

INDEX TERMS Software crash, crash reporter, exploitation, denial-of-service, bypass ASLR.

I. INTRODUCTION

SOFTWARE crashes, characterized by the unexpected termination of processes, are prevalent across various applications. Debugging and resolving these crashes is inherently challenging, as they often occur under specific conditions and lead to immediate process termination, making post-mortem analysis and reproduction difficult for developers. To address this issue, numerous crash-reporting systems have been developed and integrated by software vendors [2], [4], [5], [7]–[10], [19], [22], [23]. These systems automatically collect comprehensive diagnostic information from a crashed process, including memory snapshots, CPU registers, access permissions, and execution environment details, which are then stored in a local crash report file. Subsequently, the crash report is transmitted to a centralized error reporting

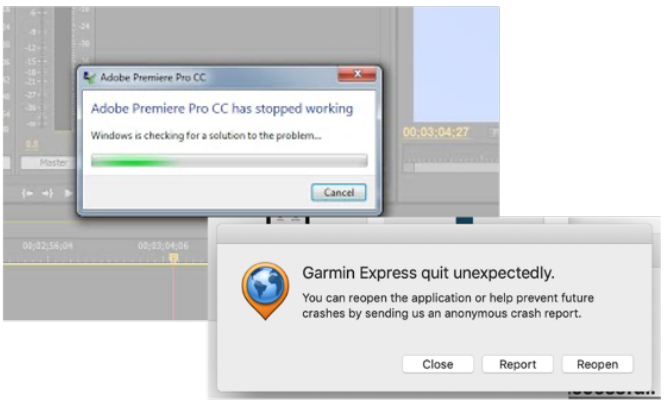


FIGURE 1: Examples of crash-reporting systems.

server, where developers can analyze the issue, reproduce the crash, and deploy patches efficiently. Given their utility, crash-reporting systems have become a standard component in modern software deployment environments. For instance, Windows Error Reporting (WER) [7] has been a built-in service since Windows Vista, while Automatic Bug Reporting Tool (ABRT) [8] and Ubuntu Error Tracker [23] serve similar roles in CentOS and Ubuntu Linux.

Despite their benefits, crash-reporting systems inherently pose security risks due to the sensitive nature of crash reports, which often contain confidential user data, such as login credentials and memory-mapped addresses. If these reports are exposed to attackers, they can be exploited for severe security breaches, including privacy leakage and bypassing critical security mechanisms. For example, crash reports for web browser applications allow attackers to gather browsing history (e.g., visited URLs) and authentication tokens. In addition, attackers with access to a crash report can extract the memory layout of processes, potentially defeating Address Space Layout Randomization (ASLR)—a fundamental protection against memory corruption exploits.

To mitigate these risks, prior research has primarily focused on removing sensitive data from crash reports before transmission to external servers. CREPE [14], for instance, is a privacy-preserving crash-reporting tool designed for web browsers. It automatically detects and removes sensitive information (e.g., login credentials and visited URLs) from crash reports based on predefined policies, preventing exposure to remote attackers who may intercept the data during transmission. However, existing studies fail to address threats posed by local attackers, as crash reports are still stored and managed within the local file system, making them vulnerable to local privilege escalation attacks.

Our Approach. In this paper, we conduct a comprehensive security analysis of widely used crash-reporting systems, focusing on their underlying architectural weaknesses. Our analysis reveals that existing crash-reporting systems store and process crash reports within the local file system, introducing multiple security vulnerabilities that can be exploited by unprivileged attackers. Building on this insight, we identify two novel attack vectors that allow local, unprivileged attackers to exploit crash-reporting systems. *Timing Gap Exploitation*: This attack vector targets the privilege de-escalation step performed during crash report creation. By leveraging a timing gap, an attacker can manipulate process ownership checks, thereby allowing access to sensitive crash reports belonging to other users. *Crash Report Manipulation Attack*: This attack vector exploits the file-based workflow of crash-reporting systems, enabling attackers to inject malformed crash reports. This manipulation can be used to discover vulnerabilities within the crash-reporting system itself and further escalate privileges or execute arbitrary code. To the best of our knowledge, this work presents the first attack model that combines crash reporter workflows with timing attacks, demonstrating critical security flaws in widely deployed crash-reporting systems.

This paper makes the following key contributions:

- *Comprehensive Security Analysis*: We perform a white-box security analysis on an open-source crash-reporting system, investigating its internal architecture and identifying two new attack vectors: a timing gap attack and a crash report manipulation attack.
- *Attack Model and Real-world Evaluation*: We develop a two-phase attack model based on these two vectors and conduct extensive evaluations across popular open-source crash-reporting systems, leading to the discovery of four new CVEs. These vulnerabilities include information leakage, ASLR bypass, arbitrary code execution, and denial-of-service (DoS) attacks.
- *Mitigation Strategies*: We propose practical mitigation strategies to eliminate the identified security risks, providing concrete solutions for enhancing the security of crash-reporting systems.

The remainder of this paper is structured as follows. Section II reviews previous studies and compares them with our work. Section III outlines the common workflow of existing crash-reporting systems and introduces our threat model, followed by an in-depth analysis of two novel identified attack vectors within this workflow. Section IV presents a two-phase attack model based on these attack vectors. Section V evaluates the proposed attack model through real-world attack scenarios. Section VII discusses the limitations of our approach. Finally, Section VIII concludes this paper.

II. RELATED WORK

This section reviews prior research on crash-reporting systems, categorizing existing studies into three key areas: (i) production-grade crash-reporting systems, (ii) privacy-preserving crash-reporting mechanisms, and (iii) fine-grained crash analysis techniques.

A. CRASH-REPORTING SYSTEMS

Most modern operating systems and mainstream applications employ proprietary crash-reporting systems to diagnose software failures effectively. For instance, Windows utilizes the Windows Error Reporting (WER) [7] system to handle crashes in Windows software, including the operating system itself and the Microsoft Office suite. WER continuously monitors process failures and generates crash reports. Upon detecting a fault, it initiates the *WerFault.exe* process, which collects diagnostic data, such as memory snapshots, CPU register states, and stack traces, and prompts the user to send a crash report to Microsoft for analysis.

In addition to OS-level crash reporting, individual applications often implement their own crash reporting mechanisms to enable fine-grained analysis tailored to their specific requirements. For example, the Chrome browser employs Crashpad [5] as its crash-reporting system on Windows and macOS, whereas it uses Breakpad [4], [19] in Linux environments. The main difference between these two reporting systems lies in their crash report generation approach: Crashpad

operates out-of-process, meaning it utilizes a dedicated external process to handle crash reporting, ensuring that reports are reliably generated even if the application is in an unstable state. In contrast, Breakpad functions as an in-process crash reporter, which, while lightweight and efficient, may fail to generate reports in cases of severe process corruption or instability. Similar to WER, both systems collect diagnostic crash data and transmit it to an external server, where the crash scenario is reconstructed by correlating the received dump with pre-existing debug symbols.

B. PRIVACY-PRESERVING CRASH REPORTING

Since crash reports may contain sensitive information, including user data, prior research has explored privacy-preserving crash reporting mechanisms. The majority of these studies [1], [6], [21] focus on identifying and mitigating the exposure of sensitive data within crash reports. For instance, Satvat et al. propose CREPE [21], a privacy-preserving crash reporting tool for web browsers. CREPE automatically detects sensitive data (e.g., user passwords) within crash reports and removes it before transmission to remote servers, thereby enhancing user privacy. KLASSIFI [1] extends this approach by leveraging machine learning (ML) techniques to detect and sanitize privacy-sensitive data across various system diagnostic sources, particularly memory dumps, while preserving essential debugging metadata. KLASSIFI incorporates a built-in knowledge base containing multiple classifiers, including ML-based classifiers, to identify a broad range of sensitive data types. Additionally, it allows users to augment the knowledge base with domain-specific and user-defined identifiers. Another approach [3], [15] focuses on replacing sensitive data in crash reports with obfuscated or synthetic data, thereby eliminating privacy concerns. For example, Castro et al. [3] propose a crash reporting technique that generates alternative inputs satisfying the same execution path conditions as the original crash-triggering input. These generated inputs, unrelated to the original data, enable vendors to reproduce crashes without directly accessing user-sensitive information.

While these studies effectively mitigate privacy risks associated with crash reporting tools, they still rely on storing crash reports in local file systems before forwarding them to remote servers. Our evaluation reveals that these locally stored reports introduce additional attack vectors, allowing adversaries to exploit them for application vulnerabilities and privacy leakage attacks (Section V).

C. FINE-GRAINED CRASH REPORT ANALYSIS

Crash reports serve as a critical resource for diagnosing application failures, identifying vulnerabilities, and facilitating efficient debugging. Numerous studies have explored automated techniques for analyzing these reports, leveraging methods such as execution path reconstruction [7], [12], [24]–[26] and memory state analysis [11], [16], [18], [20]. In the context of execution path reconstruction, POMP [12] employs reverse execution and taint analysis to track data flow and analyze crash artifacts, thereby reconstructing program semantics for root cause analysis and vulnerability detection. Similarly, REPT [24] enhances execution history reconstruction by integrating lightweight hardware tracing of control flow with offline binary analysis, enabling the recovery of data flow from crash reports with high fidelity. Another line of research focuses on memory snapshot analysis. CrashLocator [18], for instance, performs stack trace analysis to localize faults using core dumps. Additionally, RETracer [25], developed by Microsoft, utilizes Intel PT traces and core memory dumps to analyze crashes, filtering out noncritical vulnerabilities and reconstructing program semantics for efficient vulnerability detection.

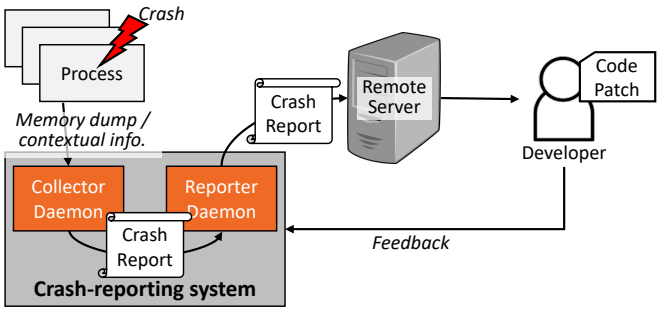


FIGURE 2: The common workflow of crash-reporting systems adopted by most production-grade systems.

While prior studies primarily aim to extract valuable debugging insights and identify security vulnerabilities from crash reports, this paper shifts the focus towards examining potential attack vectors within existing crash-reporting systems, highlighting their security implications.

III. BACKGROUND AND MOTIVATION

In this section, we first outline the common workflow of crash reporting adopted by current crash reporting systems [4], [5], [7]. We then introduce a threat model targeting crash reporting systems and examine the two security implications inherent in their workflow.

A. CRASH REPORTING WORKFLOW

A crash reporting system comprises two main components: the collector daemon and the reporter daemon. Figure 2 shows the common workflow of these daemons, which consists of four distinct steps. (1) The collector daemon continuously monitors the kernel to detect crashes in target processes. Upon detecting a crash, it retrieves contextual information from the kernel, including CPU register states and execution environment details. (2) Based on this information, it produces a crash report in a designated directory. (3) Subsequently, the system prompts the user for consent to transmit the crash report to a remote server, typically via a graphical alert message. If the user grants permission, the reporter daemon is triggered. (4) The reporter daemon processes the crash report by encoding it into a specialized format and transmits it to the remote server for further analysis. The remainder of this section provides details of each step.

Collecting Crash Information. When a process crashes, the kernel's core pattern handler captures a memory snapshot of the crashed process and generates a core dump. The collector daemon retrieves this core dump along with the process ID (PID) and additional contextual information (e.g., stack frames) from the kernel handler through an inter-process communication (IPC) mechanism, such as pipes, shared memory, or sockets. While most collector daemons employ pipe-based IPC, when a crash occurs within a container [17], an OS-level virtualization mechanism for running multiple isolated execution environments, the collected data is transmitted to the collector via socket-based IPC.

Generating Crash Reports. Based on the collected data, the collector daemon determines whether to generate a crash report. It first examines the configuration file stored in a predefined directory (e.g., the crash directory). If a blacklist rule matching the collected data exists, the daemon skips report generation for the detected crash. Otherwise, it creates a crash report file and stores it in the crash directory (e.g., */var/crash*). The file name is constructed using the absolute path of the crashed program and the user ID (UID), followed by the *.crash* file extension. A created crash report consists of two main components: contextual information and an encoded core dump. Contextual information includes environment variables, execution environment details, and data from the proc filesystem, such as the process memory map, thread information, and kernel structure metadata. The core dump contains the memory state of the crashed process, CPU registers, and stack variables. This information enables the reproduction of the detected crash on a remote server, facilitating effective debugging for the crashed application. Here, the generated crash report file inherits the permissions of the crashed process, meaning the crash directory may contain multiple reports with varying access permissions, depending on the process owner. To manage these files effectively, the crash directory is typically configured with the sticky bit, allowing multiple users to write files within the directory while preventing unauthorized access across users.

Requesting User Consent. The next step involves obtaining user consent for crash report transmission. The collector daemon then displays a GUI prompt to notify the user of the newly created report, as shown in Figure 1. If the user grants permission, the collector daemon creates a trigger file with the *.upload* file extension in the crash directory. This marks the final operation of the collector daemon.

Transmitting Crash Reports. The reporter daemon is responsible for securely transmitting crash reports to a remote server. It continuously monitors the crash directory for newly created crash report files generated by the collector daemon. Upon detecting a new crash report, the reporter daemon first verifies the Internet connection to ensure a reliable transmission. If the connection is unavailable, the report is added to a queue for deferred processing. Once a connection is established, the reporter daemon parses and encodes the crash report. The report contents are structured as key-value pairs, which are first converted into a hash table as an intermediate data structure. The daemon then serializes this table into a suitable format for transmission, such as binary-JSON (BSON), and sends it using various protocols (e.g., HTTP).

Crash reports contain memory snapshots, which can be large in size. To optimize transmission, the reporter daemon first generates a stack signature from the memory snapshot using debug symbols and a retrace tool. Instead of immediately transmitting the full memory snapshot, the daemon initially sends only the contextual information and stack signature to the server. If the server does not have a matching stack signature in its database, it requests the complete memory snapshot. The reporter daemon then encodes, compresses, and transmits the full snapshot. If a matching stack signature already exists, the server returns the UUID associated with the existing signature, avoiding redundant data transmission. Upon receiving a new crash report, the server stores the data in a database and triggers the retrace tool if no corresponding stack-trace results exist. Once the stack trace signature is computed, it is recorded in the database for future reference.

B. THREAT MODEL

Before introducing attack vectors within the current crash reporting systems, we define our threat model. Specifically, we target the standard crash reporting system architecture consisting of collector and reporter daemons (Section III-A). While our model can be extended to various operating systems, including Windows, macOS, and Linux, we focus on Linux distributions due to their widespread adoption.

We consider non-privileged local attackers (i.e., a malicious user) who possess the same access permission as normal user accounts but do not have root privileges. These attackers are assumed to have access to the report directory (e.g., */var/crash/*), where crash reports and configuration files are stored. However, we assume that attackers can only access crash reports and configuration files associated with their own processes, as the crash directory is typically configured with the sticky bit, which restricts cross-user access. We consider this assumption to be realistic and applicable to most Linux distributions. For instance, in Ubuntu, one of the most widely used Linux distributions, normal users without root privileges have write access to */var/crash/*, allowing them to store crash reports for their own processes, while access to other users' reports remains restricted.

To exploit the first attack vector, we further assume that attackers can trigger crashes in their target processes, which may include privileged system processes or normal user processes. This can be achieved by leveraging known denial-of-service (DoS) vulnerabilities in these processes. While DoS vulnerabilities are generally considered lower in severity compared to other types of security flaws (e.g., information leakage or arbitrary code execution), they are often more prevalent and easier to exploit. Consequently, even low-severity DoS vulnerabilities in system-level daemon processes can serve as an entry point for our first attack vector, ultimately leading to more severe security breaches.

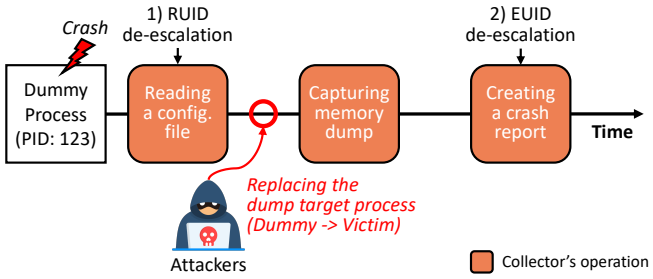


FIGURE 3: Attack vector 1: exploiting the timing gap during the privilege de-escalation steps. By abusing this gap, attackers can obtain access to the victim's crash report.

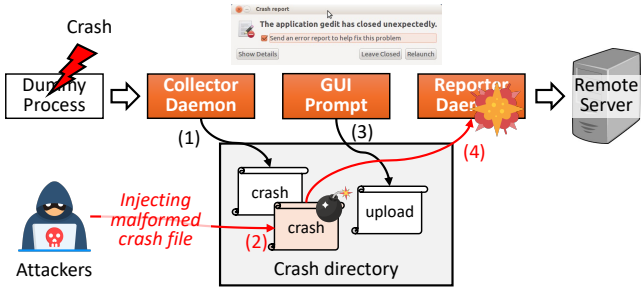


FIGURE 4: Attack vector 2: exploiting the file-based workflow between the collector and reporter daemons.

C. MOTIVATION

In this section, we introduce two attack vectors present in the current crash-reporting system architecture. Our analysis focuses on the critical observation that the crash reporting system has access to the memory contents of arbitrary processes, which may include sensitive information, and subsequently, stores crash reports containing these memory contents in a shared local crash directory accessible by multiple users. Based on this observation, we conduct a systematic analysis of the collector and reporter daemons' workflows and identify the following two attack vectors.

Timing Gap within Privilege De-escalation. Crash reports contain sensitive information (e.g., memory states) about a crashed process. Although access control mechanisms (e.g., a sticky bit) within the crash directory prevent attackers from reading reports belonging to other users, a timing gap exists between the time when the collector daemon retrieves crash data of a process and when it saves the generated report. To ensure that users can access crash reports for their own processes, the collector, which initially runs with root privileges using *setuid()*, should de-escalate its privileges to match those of the crashed process owner before saving the report. This process involves three steps, as shown in Figure 3. (1) The collector first modifies its real user ID (RUID) to match the crashed process owner, allowing it to read the necessary configuration file. (2) It then dumps the memory contents and other contextual information of the crashed process. (3) Finally, the collector drops its effective user ID (EUID) to that of the crashed process owner, ensuring that the report is stored under the appropriate user permissions.

Since reading and parsing a configuration file requires processing time, a timing gap exists between the moment a process crashes and the moment its memory contents are dumped. Moreover, the collector daemon references the PID of the crashed process to determine the target process for memory dumping and ownership assignment. Attackers can exploit this timing gap by triggering a crash in a dummy process and subsequently restarting a victim process with the same PID as the previously crashed dummy process. If the attacker successfully manipulates the PID reassignment, the collector daemon mistakenly associates the victim process's memory with the attackers' dummy process, ultimately saving the victim's crash report under the attacker's permissions.

This attack vector results in information leakage vulnerabilities. For instance, if an attacker extracts the Address Space Layout Randomization (ASLR) offset of a privileged process, they could bypass ASLR protections and combine this information with other memory corruption vulnerabilities to escalate privileges or gain arbitrary code execution.

File-based Workflow. The second attack vector exploits the file-based workflow of crash reporting systems. As shown in Figure 4, these systems rely on various files (e.g., .crash and .upload files) stored in a shared crash directory accessible by multiple users. For example, the reporter monitors the crash directory for newly created .upload files—generated through GUI prompts—and subsequently attempts to read the corresponding .crash file created by the collector daemon. Since the crash directory is shared among multiple users, a file-based locking mechanism is implemented to synchronize interactions between the collector and reporter daemons. Specifically, a .lock file is used to temporarily prevent the creation of new crash reports, forcing pending crashes to remain in the collector's queue. This file-based workflow facilitates flexible and efficient inter-component communication by enabling loose coupling between system components. However, it also introduces a critical security vulnerability. The primary concern is that the reporter daemon operates with root privileges, yet it processes files that normal users can create and modify. As shown in Figure 4, attackers within our threat model can inject malformed crash report files into the crash directory, causing the reporter daemon to process these malicious files instead of legitimate crash reports. These malformed files may contain malicious payloads designed to exploit known vulnerabilities within the reporter daemon, leading to severe security consequences.

This attack vector enables multiple exploitation scenarios, including information leakage, arbitrary code execution, and denial-of-service (DoS) attacks. For instance, if an attacker successfully exploits a vulnerability within the reporter daemon, they could execute arbitrary commands with root privileges, significantly compromising system security.

IV. ATTACK MODEL DESIGN

In this section, we propose a two-phased attack model designed to identify and exploit the two attack vectors in crash-

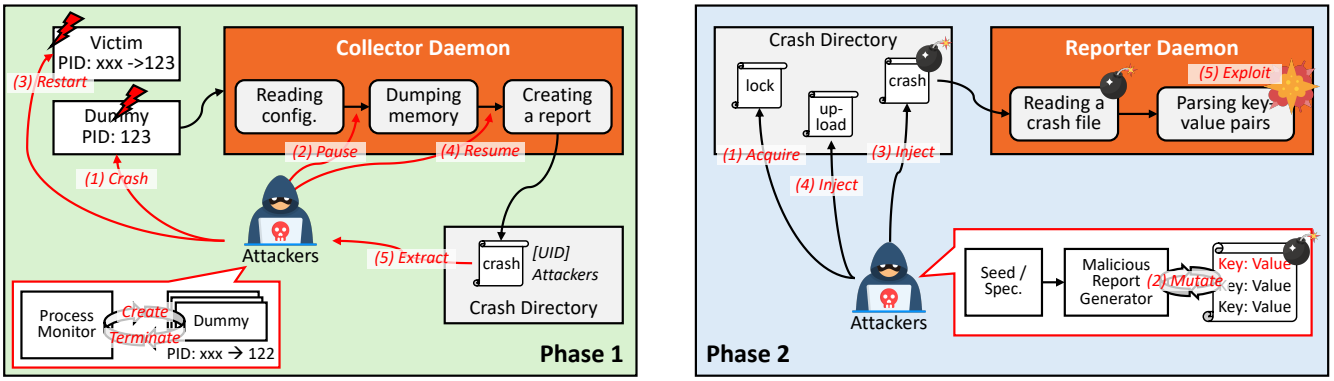


FIGURE 5: The overall workflow of our two-phased attack model. The first phase (the green box) is the dump target swapping attack, and the second phase (the blue box) is the malicious crash report injection attack.

reporting systems, thereby performing system-level exploitation, including ASLR bypass, arbitrary code execution, and DoS attacks. The overall architecture of the proposed attack model is shown in Figure 5. The first phase, referred to as *a crash report hijacking attack*, exploits the timing gap attack vector to manipulate the access permission associated with a newly generated crash report. This allows attackers to gain unauthorized access to crash reports belonging to other processes, extracting sensitive memory contents contained within them. The second phase involves a *malicious file injection attack* targeting the reporter daemon. This enables attackers to compromise the reporter daemon, leading to private data leakage and arbitrary code execution.

The remainder of this section provides a detailed breakdown of each phase, including the technical foundation and exploitation methodology.

A. PHASE 1: DUMP TARGET SWAPPING ATTACK

The first phase exploits a timing gap in the gradual privilege de-escalation process of the collector daemon. The attack is based on the key observation that the collector references only the PID when capturing memory contents and saving a crash report, without performing additional verification. Leveraging this observation, we design a dump target swapping attack, which allows attackers to swap the intended memory dump target, causing the collector to inadvertently dump the memory of a victim process. Here, the selected victim is a privileged daemon process known to have exploitable DoS vulnerabilities. As shown in Figure 5 (green box), the attack consists of five steps.

(1) The attackers force a crash in a dummy process, activating the collector. Upon activation, the collector drops its RUID from the root to the attackers' UID to read the corresponding configuration file. (2) They send a SIGSTOP signal to pause the collector, prolonging the timing gap. During this pause, they manipulate the victim process's PID to match that of the crashed dummy process. Since PIDs are allocated in a round-robin manner, the attackers use a process manager to continuously create and terminate dummy processes, ensuring that the next allocated PID matches the crashed process's PID. This process can be optimized by maintaining a pool of dummy processes. Once the PID is correctly positioned, (3) they forcefully terminate the victim process using a known DoS vulnerability. As the victim is a daemon process, the OS automatically restarts it, assigning the manipulated PID. (4) The attackers send a SIGCONT signal to resume the collector's operation. (5) The collector dumps the victim's memory and saves the crash report under the attackers' UID, as its RUID is already modified after step (1). As a result, the attackers can obtain unauthorized access to the crash report, which contains sensitive memory snapshots and execution state data of the victim process.

B. PHASE 2: MALICIOUS REPORT INJECTION ATTACK

As discussed in the previous section, the reporter operates within a file-based workflow, executing specific actions based on the presence of various files (e.g., .upload and .crash files) in the crash directory. The core security flaw in this design lies in the shared nature of the crash directory, which allows non-root attackers to inject malicious files into the reporter daemon's workflow. Since the reporter runs with root privileges, this attack can trigger vulnerabilities within the reporter, leading to privilege escalation and arbitrary code execution. To exploit this file-based workflow, we design a malicious crash report injection attack, as shown by the blue box in Figure 5. The attack proceeds as follows:

(1) The attackers create a .lock file, suspending the creation of new benign crash reports by other processes. (2) The attackers then create a malicious .crash file designed to exploit vulnerabilities within the reporter daemon. A standard crash report is structured as a set of key-value pairs following a predefined format. If a crash report deviates from this expected format, the reporter daemon rejects it at early stages, avoiding further processing. To ensure that the malicious crash report is both valid and capable of triggering vulnerabilities, the attackers employ a structured file generator. This generator constructs a syntactically valid crash report while mutating key-value pairs, increasing the likelihood of triggering exploitable conditions within the reporter daemon. (3) Once the malicious .crash file is created and injected into the crash

directory, (4) the generator produces a .upload file, signaling the reporter daemon to process the newly injected crash report. Attackers then monitor the reporter daemon's output files, (5) identifying unexpected behaviors or crashes caused by the malformed crash report. These unintended executions could lead to security breaches, including arbitrary code execution, unauthorized data access, or privilege escalation.

V. EVALUATION

To assess the effectiveness of our proposed attack model, we develop a prototype tool using Python. This tool automates the exploitation process for both attack phases. For Phase 1, we utilize a simple sleep process as a dummy process, allowing controlled manipulation of PIDs. For phase 2, monitoring of the crash directory and the reporter daemon's /proc file system is implemented using the *inotify* library [13]. The prototype provides real-time summaries of input files each time a new crash report is generated. We apply our prototype tool to two widely used open-source crash reporting systems: Apport [22] and Whoopsie [22].

We reveal three distinct types of security vulnerabilities:

- *Bypassing ASLR*: attackers can obtain memory base addresses and offsets of loaded libraries from a privileged victim process, bypassing ASLR protections [§ V-A].
- *Arbitrary code execution*: by injecting a malformed crash report, attackers can overwrite heap memory regions within the reporter daemon, ultimately achieving root-level code execution [§ V-B].
- *Denial of service*: attackers can disrupt a crash-reporting system by manipulating crash and upload files, preventing benign crash reports from being processed [§ V-C].

As a result of our findings, we discover four new CVEs related to these vulnerabilities, leading to security patches, as summarized in Table 1.

A. BYPASSING ASLR

The first identified vulnerability involves bypassing ASLR in system daemon processes. By exploiting known DoS vulnerabilities targeting a privileged daemon, attackers can defeat ASLR protections, significantly increasing the feasibility of memory corruption exploits. Specifically, the first phase of our attack model (dump target swapping attack) enables attackers to capture a detailed memory snapshot of the victim process. To maximize the probability of PID manipulation, attackers employ a pool of sleep dummy processes combined with a file-based lock mechanism (*.lock* file). The attack proceeds as follows: (1) Attackers force a crash in a dummy process, activating the collector daemon. (2) A .lock file is created in the crash directory to prevent the creation of additional crash reports, ensuring that the attack remains undisturbed. (3) The attackers then spawn multiple dummy processes in a process pool, increasing the likelihood of matching PIDs between a dummy process and the victim process. (4) Once enough dummy processes are running, the .lock file is removed, allowing normal crash reporting to resume. (5) The collector's crash report contains critical contextual information extracted from the victim's */proc* filesystem. By analyzing the extracted memory snapshot, attackers can: (i) Identify base addresses of stack and heap regions. (ii) Extract offsets of dynamically loaded libraries (e.g., libc). (iii) Locate function addresses used by the victim process. These leaked addresses allow attackers to bypass ASLR and construct precise exploit payloads, facilitating memory-based attacks such as Return-Oriented Programming (ROP).

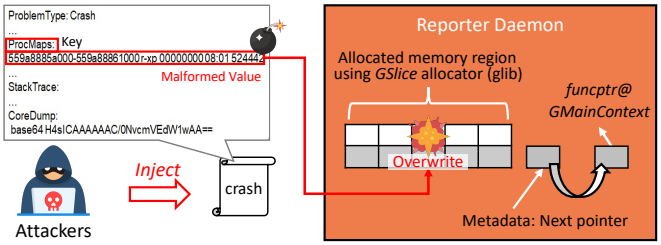


FIGURE 6: Attackers can insert a malformed crash report that causes a heap overflow within the reporter daemon.

B. ARBITRARY CODE EXECUTION

We identify an integer overflow vulnerability in the reporter daemon (Whoopsie), enabling attackers to execute arbitrary code with root privileges. This vulnerability can be exploited through the second phase of our attack model. As described in Section IV, attackers can inject a malformed crash report containing malicious key-value pairs into the crash directory. When the reporter daemon processes this file, it loads key-value pairs into heap memory without enforcing length restrictions on the key size. If the key length exceeds INT32_MAX, an integer overflow occurs.

Figure 6 shows the exploitation process. (1) When the reporter daemon reads the attacker-injected .crash file, it loads key-value pairs into heap memory using BSON encoding. The lack of key length validation results in an integer overflow when allocating memory for the key. (2) Due to this miscalculation, the allocated buffer is smaller than expected, leading to a heap-based buffer overflow when the excessively long key is written into memory. (3) The heap overflow allows the attackers to corrupt adjacent memory regions, including function pointers stored within the *GMainContext* structure. Here, *GMainContext* is responsible for handling main event loops in GLib-based applications, including Whoopsie. One of the key elements in this structure is a callback function pointer (dispatch) used for processing events. (4) By overwriting the dispatch function pointer in *GMainContext*, the attackers redirect execution to a controlled memory region containing shell-code or an ROP chain. When the event loop executes, it inadvertently calls the attackers' payload, leading to arbitrary code execution with root privileges. As a result, the attackers can spawn a root shell, modify system files, or escalate privileges further by injecting backdoors or altering security configurations.

TABLE 1: A list of real-world vulnerabilities discovered using the proposed method.

Package	CVE	Description	Severity	Status
whoopsie 0.2.69	CVE-2020-12135	A buffer overflow vulnerability in `bson_ensure_space()` in `bson.c` of whoopsie 0.2.69 allows an attacker to execute arbitrary code within the whoopsie process via a specially crafted BSON input.	High	Fixed Release
whoopsie 0.2.69	CVE-2020-15570	The `parse_report()` function in `whoopsie.c` mishandles memory allocation failures, allowing an attacker to cause a denial of service via a malformed crash file.	-	Fixed Release
whoopsie 0.2.69	CVE-2020-11937	In whoopsie, `parse_report()` in `whoopsie.c` allows a local attacker to cause resource exhaustion through a crafted file.	Medium	Fixed Release
apport 2.20.11	CVE-2020-15701	An unhandled exception in `check_ignored()` in `apport/report.py` can be exploited by a local attacker to cause a denial of service. If the `mtime` attribute is a string value in `apport-ignore.xml`, it triggers an unhandled exception, resulting in a crash.	Critical	Fixed Release

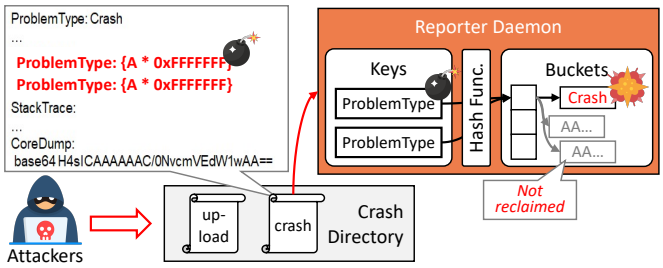


FIGURE 7: By injecting malformed crash and upload files, attackers can trigger the kernel's OOM-killer, thereby conducting a system resource exhaustion attack.

C. DENIAL OF SERVICE

By utilizing our attack model, attackers can employ multiple methods to conduct Denial-of-Service (DoS) attacks against crash-reporting systems:

Exploiting the World-Writable .lock File. Crash-reporting systems utilize a .lock file to coordinate operations among multiple collector instances within the crash directory. If attackers gain control over this .lock file and refuse to release it, the crash-reporting system becomes unable to function properly, leading to a denial of service.

Sending a SIGSTOP Signal. Typically, users cannot send a SIGSTOP signal to privileged processes. However, during the privilege de-escalation process, the collector temporarily reduces its real user ID (UID) to that of the user to check for file existence (the first phase of our attack model). At this moment, attackers can send a SIGSTOP signal to the collector, disrupting its operation and effectively halting the crash-reporting process.

Forcing an Unhandled Exception. Before generating a crash report, the collector checks a blacklist containing pairs of file types and paths to exclude. If attackers modify this list to omit the file-type element, it triggers an unhandled exception, causing the collector to crash. This manipulation can lead to a denial of service by exploiting the collector's error-handling mechanism.

Creating Malformed Crash Reports. The reporter daemon processes crash reports formatted as key-value pairs and stores them in a hash table. As shown in Figure 7, attackers can submit a malformed crash report designed to cause hash collisions, leading to abnormal behavior. Specifically, when a hash collision occurs, existing data may be replaced or improperly chained with new data, resulting in potential memory resource exhaustion. This unchecked memory growth can trigger the Out-Of-Memory (OOM) killer, potentially terminating arbitrary processes within the system.

These DoS vulnerabilities within crash-reporting systems can be exploited to disrupt normal operations, leading to system instability and potential service outages.

VI. MITIGATION STRATEGIES

In this section, we present mitigation strategies designed to prevent the exploitation of the two attack vectors identified in crash-reporting systems: (i) the timing gap in the privilege de-escalation step and (ii) the file-based workflow. Following the disclosure of the four CVEs identified by our attack model, vendors have adopted these mitigation strategies to enhance the security of their crash-reporting systems.

(i) Mitigation for Timing Gap Exploitation. The first attack vector enables attackers to manipulate the collector daemon's memory dump operation by exploiting a timing gap during privilege de-escalation, allowing them to swap the dump target from a dummy process to a victim process. The root cause of this vulnerability is that the collector daemon fails to validate the original crash target (i.e., the process that initially crashed). Instead, it identifies the dump target solely based on the stored PID, without verifying whether the process associated with the PID has changed. This PID-based process identification, combined with a timing gap between privilege de-escalations, allows attackers to replace the intended dump target before the collector captures its memory contents.

To eliminate this vulnerability, the collector daemon should securely track the original dump target by creating a file descriptor linked to the /proc filesystem path of the crashed process at the time of the crash. Before performing a memory dump, the collector should verify that the file descriptor still corresponds to the original process to ensure the target has not been swapped. This mitigation strategy has been integrated into the latest version of the crash reporter.

(ii) Mitigation for File-based Workflow. The second attack vector leverages the file-based nature of crash reporting workflows, allowing attackers to inject malicious files (e.g.,

.crash, .lock, and .upload files) into the shared crash directory to manipulate the behavior of the collector and reporter daemons. The problem here is that the crash directory is shared among multiple users, allowing attackers to modify or inject files that influence the workflow of the crash-reporting system. The collector and reporter daemons process these files without sufficient validation, making them susceptible to malicious file injection attacks.

To prevent unauthorized manipulation, we propose the following security enhancements. First, a crash-reporting system should enhance file access control mechanisms beyond the sticky bit by allowing only authorized processes to modify critical files, preventing unauthorized users from reading or injecting crash-related files. In addition, files that trigger critical actions in the collector and reporter daemons should be stored in a separate directory with strict privilege restrictions. This prevents attackers from inserting or modifying control files that dictate crash-reporting system behavior. For example, Polkit, the Linux authorization manager, previously restricted file access to root-only permissions. To further enhance security, we propose an additional rule allowing only the owner of a crashed process to access its crash report, ensuring privacy and restricting unauthorized data access. While this additional file access policy is still under discussion and has not yet been implemented in upstream projects, patches addressing each identified vulnerability have been successfully integrated, significantly enhancing the security of crash-reporting systems against these attack vectors.

VII. DISCUSSION

In this section, we examine the limitations of our two attack vectors targeting crash reporting systems and discuss their real-world security implications.

A. POTENTIAL ATTACK CONSTRAINTS.

Our proposed attacks are based on three key assumptions: (i) attackers have read/write access to the crash directory, (ii) the collector and reporter daemons synchronize their operations using a file-based workflow, and (iii) attackers can leverage known DoS vulnerabilities to crash and terminate target processes. While these assumptions remain reasonable in many real-world scenarios, they introduce certain constraints.

Most Linux-based crash reporting systems meet the first assumption. For example, popular crash reporting frameworks such as Apport and Launchpad, used in Ubuntu, RedHat, and CentOS, allow normal users to access the crash directory, which is protected by the sticky bit permission. However, certain crash reporting systems impose stricter access controls, restricting normal users from accessing crash reports that contain memory dumps and contextual information. For instance, Windows Error Reporting (WER) does not meet this assumption, as it stores crash reports in a system-protected directory with NTFS access control lists, ensuring that only administrative users or the Windows system can access these files. Similarly, ABRT [8], while functionally similar to Apport, stores crash reports in a privileged directory [23], further restricting access.

The second assumption is that the collector and reporter daemons interact through a file-based mechanism, allowing attackers to manipulate inputs fed into the reporter daemon. This assumption holds for many crash reporting systems but does not apply universally. For example, while Apport follows a file-based workflow, ABRT utilizes predefined IPC mechanisms and the D-Bus IPC channel for communication between its collector and reporter daemons. This presents a challenge, as it restricts attackers from directly injecting malicious inputs into the reporter daemon, making our second attack vector more difficult to exploit. Consequently, attackers must integrate an additional vulnerability into their exploit chain—such as an IPC hijacking attack—to intercept communication between the collector and reporter daemons.

The third assumption requires that attackers be able to force a crash in their target processes—which often have higher privileges—and subsequently terminate them to manipulate crash report handling. In most cases, normal users lack the necessary privileges to terminate system-level processes. However, numerous known DoS vulnerabilities exist that specifically target privileged system processes. By exploiting these low-severity DoS vulnerabilities, attackers can force crashes in system processes, enabling the exploitation of our proposed attack vectors. More critically, these attack vectors significantly amplify the impact of DoS vulnerabilities, allowing attackers to escalate their capabilities from mere service disruption to more severe consequences, such as arbitrary code execution or sensitive data extraction.

B. REAL-WORLD ATTACK SCENARIOS.

The two attack vectors presented in this paper demonstrate how unprivileged attackers can exploit crash reporting systems to compromise privileged processes or access sensitive information on the same host. This approach is particularly effective in real-world environments where direct exploitation of high-value targets is difficult due to robust security protections. By leveraging the crash reporting workflow, attackers can bypass conventional security barriers and extract sensitive data from well-protected applications.

For instance, the Chrome browser securely stores ID-password pairs for websites visited during web browsing, making it a high-value target for attackers. Due to Chrome's comprehensive security mechanisms, directly exploiting an information leakage vulnerability to steal user credentials is highly challenging. However, the attack vectors proposed in this paper provide an alternative exploitation strategy. Rather than extracting credentials from Chrome directly, attackers can exploit a known DoS vulnerability to force the crash of the keyring process, a component responsible for securely managing stored credentials. Unlike information leakage vulnerabilities, which are difficult to find and exploit, DoS vulnerabilities are more prevalent and often considered low-severity. Once the keyring process crashes, the crash reporting system generates a crash report containing sensitive

user credentials. The attacker can then leverage our first attack vector to extract these credentials from the created crash report. Thus, an otherwise low-severity DoS vulnerability can be escalated into a severe information leakage attack. This demonstrates that the attack vectors described in this paper are not limited to theoretical scenarios but can be practically applied to bypass existing security defenses and extract high-value information from real-world applications.

VIII. CONCLUSION

In this paper, we analyzed the security of real-world crash-reporting systems, identifying two critical attack vectors that allow unprivileged attackers to exploit these systems. We developed a systematic attack model that demonstrates how these vulnerabilities can be leveraged to leak user-sensitive information from privileged processes and achieve arbitrary code execution. Through our attack model, we identified four distinct vulnerabilities in two widely used crash-reporting systems, each of which was assigned a CVE. In response, we proposed effective and practical mitigation strategies, which led to actual security patches from vendors, significantly improving the security posture of these systems. We believe this work serves as a foundation for developing more resilient crash-reporting mechanisms.

REFERENCES

[1] H. Min A. Kaul, M. Kesarwani and Q. Zhang. "Knowledge & Learning-based Adaptable System for Sensitive Information Identification and Handling". In Proceedings of the 14th IEEE International Conference on Cloud Computing (CLOUD), 2021.

[2] Bugzilla. "Bug Tracker". https://www.bugzilla.org, 2024.

[3] M. Castroa, M. Costa, and J.-P. Martin. "Better Bug Reporting with Better Privacy". In Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2008.

[4] Chromium. "Breakpad: A set of client and server components which implement a crash-reporting system". https://chromium.googlesource.com/breakpad/breakpad, 2024.

[5] Chromium. "Crashpad: A crash-reporting system". https://chromium.googlesource.com/crashpad/crashpad, 2024.

[6] R. Ding, H. Hu, W. Xu, and T. Kim. "DESENSITIZATION: Privacy-Aware and Attack-Preserving Crash Report". In Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS), 2020.

[7] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. "Debugging in the (Very) Large: Ten Years of Implementation and Experience". In Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), 2009.

[8] Red Hat. "Automatic Bug Reporting Tool". https://fedoraproject.org/wiki/Automatic_Bug_Reporting_Tool, 2024.

[9] Adobe Systems Inc. "Adobe: Submitting Crash Reports". https://helpx.adobe.com/photoshop/kb/submit-crash-reports.html, 2024.

[10] Apple Inc. "Technical Note TN2123: CrashReporter". https://developer.apple.com/library/content/technotes/tn2004/tn2123.html, 2024.

[11] P. Chen X. Xing P. Wang J. Xu, D. Mu and P. Liu. "CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump". In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2016.

[12] X. Xing P. Liu P. Chen J. Xu, D. Mu and B. Mao. "Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts". In Proceedings of the 26th USENIX Security Symposium (Security), 2017.

[13] Linux Kernel. "inotify(7) - Linux manual page". https://man7.org/linux/man-pages/man7/inotify.7.html, 2024.

[14] K.Satvat, M. Shirvanian, M. Hosseini, , and N. Saxena. "CREPE: A Privacy-Enhanced Crash Reporting System". In Proceedings of the 20th ACM Conference on Data and Application Security and Privacy (CODASPY), 2020.

[15] J. Zou L. Zhou, L. Yu and H. Min. "Privacy-Preserving Redaction of Diagnosis Data through Source Code Analysis". In Proceedings of the 35th International Conference on Scientific and Statistical Database Management (SSDBM), 2023.

[16] B. Liblit M. Polishchuk and C. W. Schulze. "Dynamic heap type inference for program understanding and debugging". In Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2007.

[17] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: a state-of-the-art review. IEEE Transactions on Cloud Computing, 7(3):677–692, 2017.

[18] S.-C. Cheung R. Wu, H. Zhang and S. Kim. "Crashlocator: Locating crashing faults based on crash stacks". In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2014.

[19] Mozilla Crash Reporter. "Mozilla Crash Reporter"". https://support.mozilla.org/en-US/kb/mozillacrashreporter, 2022.

[20] R. Salkeld and G. Kiczales. "Interacting with dead objects". In Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA), 2013.

[21] K. Satvat and N. Saxena. "Crashing Privacy: An Autopsy of a Web Browser's Leaked Crash Reports". In arXiv preprint arXiv:1808.01718, 2018.

[22] Ubuntu. "Apport Crash Duplication Detection". https://blueprints.launchpad.net/ubuntu/+spec/apport-crash-duplicates, 2024.

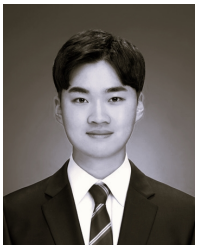
[23] Ubuntu. "Ubuntu: CrashReporting". https://wiki.ubuntu.com/CrashReporting, 2024.

[24] B. Kasikci B. Niu U. Sharma R. Wang W. Cui, X. Ge and I. Yun. "REPT: Reverse Debugging of Failures in Deployed Software". In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018.

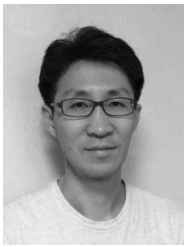
[25] S. K. Cha Y. Fratantonio W. Cui, M. Peinado and V. P. Kemerlis. "RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps". In Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016.

[26] R. Wu, M. Wen, S.C. Cheung, and H. Zhang. "ChangeLocator: Locate Crash-Inducing Changes Based on Crash Reports". In Proceedings of the 40th International Conference on Software Engineering (ICSE), 2018.

SEONG-JOONG KIM is a senior researcher at the Attached Institute of ETRI. Before that, he interned at Samsung Electronics in the capacity of a software engineer. He is a recipient of the Samsung Electronics Scholarship. His research interests are all aspects of computer security, with a current focus on making Linux distributions more secure.



MYOUNGSUNG YOU is a PhD candidate in the School of Electrical Engineering at KAIST. He received his M.S. degree in Computer Science from the School of Computing at KAIST and his B.S. degree in Computer Science from Chungbuk National University. His research interests include cloud networking, programmable network data planes, network security, and privacy communication.



SEUNGWON SHIN is an associate professor in the School of Electrical Engineering at KAIST. He received his Ph.D. degree in Computer Engineering from the Electrical and Computer Engineering Department, Texas A&M University, and his M.S. degree and B.S. degree from KAIST, both in Electrical and Computer Engineering. He is currently an executive vice president at Samsung Electronics, leading the security team in the IT & Mobile Communications Division. His research interests span the areas of Software-defined networking security, DarkWeb analysis, and Cyber threat intelligence.