PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks

Quan Chen*
North Carolina State University
qchen10@ncsu.edu

Ahmed M. Azab
Samsung Research America
a.azab@samsung.com

Guruprasad Ganesh
Samsung Research America
g.ganesh@samsung.com

Peng Ning
Samsung Research America
peng.ning@samsung.com

Abstract

Commodity operating systems kernels are typically implemented using low-level unsafe languages, which leads to the inevitability of memory corruption vulnerabilities. Multiple defense techniques are widely adopted to mitigate the impact of memory corruption on executable code and control data. Nevertheless, there has not been much attention to defend against corruption of non-control data despite the fact that previous incidents of kernel exploitation showed that corrupting non-control data is a real threat.

We present PrivWatcher, a framework for monitoring and protecting the integrity of process credentials and their usage contexts from memory corruption attacks. PrivWatcher solves multiple challenges to achieve this objective. It introduces techniques to isolate and protect the data that define process credentials and guarantee the locality of this data within the protected memory. Then, by adopting a dual reference monitor model, it guarantees the Time of Check To Time of Use (TOCTTOU) consistency between verification and usage contexts for process credentials. Moreover, it provides a secure mechanism that allows the presumably protected kernel code to verify the protected data without relying on unprotected data fields.

PrivWatcher provides non-bypassable integrity assurances for process credentials and can be adapted to enforce a variety of integrity policies. In this paper, we demonstrate an application of PrivWatcher that enforces the original semantics of the OS kernel's access control policy: a change in process privileges is legitimate only if an uncompromised kernel would have allowed it. We implemented a PrivWatcher prototype to protect Ubuntu Linux running on x86-64. Evaluation of our prototype showed that PrivWatcher is effective and efficient.

*The author developed some of the subject matter of this paper while employed at Samsung Research America.

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates

ISBN 978-1-4503-4944-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3052973.3053029

1. INTRODUCTION

Memory corruption vulnerabilities are a common and inevitable security weakness of modern operating system (OS) kernels, due to the fact that they are routinely written in low-level memory-unsafe languages such as C and assembly. These vulnerabilities pose a great risk to the security of the entire system. For instance, in recent years the Linux kernel has seen several such vulnerabilities [2, 3, 4, 5, 6] that led to privilege escalation exploits.

Exploits based on memory corruption vulnerabilities can be categorized according to the type of memory content they target: 1) executable code, 2) control-flow data, and/or 3) non-control-flow data.

Multiple defense techniques are widely adopted to mitigate the impact of the first two types of attacks. Commodity OS kernels implement data execution prevention and read-only mapping of executable code to mitigate the threat of code injection attacks. They also implement stack canaries and address space randomization to mitigate the threat of control flow hijacking through corrupting control data. In addition, there has been a plethora of existing research to enforce and enhance these protection techniques. Multiple proposed systems (e.g., [18, 19, 20, 42, 50, 48, 28, 54]) monitor the kernel from an isolated domain to guarantee that its code base cannot be modified, relocated or illegally amended. Previous defense efforts also aim to make the protection of kernel control data more robust. There are existing proposals (e.g., [26, 45, 40]) to prevent or otherwise mitigate the threat of control flow hijacking attacks against the kernel. Researchers have also recently demonstrated the feasibility of enforcing control-flow integrity (CFI) [15] protection on commodity OS kernels [27, 32].

This paper focuses on the third type of attacks, which is memory corruption of kernel non-control data. Specifically, PrivWatcher is designed to monitor and protect process credentials data in Linux. Process credentials are data fields that determine the subject identity (i.e., privilege) of a process. Hence, it is a crucial part of the kernel's access control subsystem. Previous research indicated that memory corruption of non-control data is an equally significant security threat as corruption of control data [25, 35]. This observation also applies to the kernel. Linux kernel's monolithic design allows process credentials to be subverted by single memory error anywhere in the kernel's code base (e.g., third-party drivers). By simply overwriting process

credentials, attackers can successfully escalate privileges for arbitrary processes without injecting code into the kernel or violating kernel control flow. In fact, virtually all of the existing memory corruption exploits (e.g., [56]) against the Linux kernel achieved privilege escalation by manipulating the credentials of attacker-controlled processes.

Requirements: Memory corruption vulnerabilities in the kernel are frequently exploited to overwrite non-control data fields in order to give elevated privileges to attacker processes. Protecting against such exploits requires a defense technique to prevent exploitable kernel-level code from directly modifying process credentials. In addition, the intended usage contexts of process credentials (i.e., the process they are supposed to describe) must also be secure from arbitrary modifications. There are multiple challenges that need to be addressed to achieve this objective.

The first challenge is *isolating process credentials*. Since non-control kernel data fields, including process credentials, are usually allocated from the kernel heap, they can be scattered throughout the kernel address space. More importantly, focusing integrity protection only on data fields that are relevant to describing process privileges minimizes the protection overhead.

The second challenge is *validating runtime updates to process credentials*. This is not straightforward because determining the legitimacy of an update is dependent on both the semantics of the new values and the context in which they will be used. Thus, the validation routine must be able to both verify the new values and ensure the authenticity of the verification context.

The third challenge is *enforcing the time of check to time of use (TOCTTOU) consistency*. Enforcing TOCTTOU consistency requires: 1) protecting credentials so they are not modified after verification, and 2) ensuring that credentials are verified and used in the same context. An example of the later requirement is ensuring that verified process credentials for a privileged process cannot be reused to describe the privileges of an attacker-controlled process.

Introducing PrivWatcher: We propose PrivWatcher, a framework that allows both integrity verification and protection of process credentials against memory corruption attacks. PrivWatcher introduces a set of novel techniques that together enable it to solve the above challenges and prevent memory corruption vulnerabilities from being exploited to give unauthorized privileges to processes. In the following, we provide a summary of these techniques:

1) *Separate Allocation of Process Credentials:* PrivWatcher proposes a set of kernel modifications that prevents the credentials data to be mixed with other data fields. This technique guarantees that PrivWatcher runs with minimum possible performance overhead by avoiding to mediate access to unrelated data fields.

2) *Dual Reference Monitor:* PrivWatcher has two components. The first resides in an execution domain that is isolated from the kernel. PrivWatcher assumes that this domain possesses control over the Memory Management Unit (MMU). It leverages this control to relocate process credentials into a safe region that is non-writable to the kernel. PrivWatcher prevents the kernel from arbitrarily changing process privileges. As a result, memory corruption exploits will lose their ability to corrupt these non-control data fields, which are crucial to secure kernel operations.

The second component is a set of kernel extensions that guarantees the TOCTTOU consistency of process credentials. In particular, this ensures that firstly, runtime memory for process credentials can only be allocated from the safe region (therefore vetted by PrivWatcher); and secondly, the context of use is the same as the context of verification. If either of these checks fail, the offending process will be permanently denied access to any sensitive resources.

3) *Data Integrity Verification:* PrivWatcher introduces multiple heuristics to identify potentially malicious updates to process credentials, e.g., updates that violate the kernel's Discretionary Access Control (DAC) policy.

We implemented a PrivWatcher prototype for Ubuntu Linux and conducted performance evaluation based on it. The performance evaluation results show that PrivWatcher is effective and incurs minimal performance overhead.

Besides the prototype, some of the techniques presented in this paper are adopted by Samsung Knox Real-time Kernel Protection [49], which is deployed on recent models of Samsung Galaxy smartphones.

Contribution: The major contributions of this paper are:

- A complete and practical solution to prevent memory corruption vulnerabilities in the kernel from being exploited to elevate the privilege of arbitrary processes.
- A novel technique to ensure TOCTTOU consistency of protected data fields, which allows the kernel to be extended to verify data integrity without relying on unprotected data.
- Introducing techniques to verifying that changes of process credentials do not violate the kernel's original access control policies.
- Full prototype implementation and evaluation of the presented techniques.

In the remainder of the paper: We describe assumptions and threat model in Section 2. We present the design and implementation in Section 3 and Section 4 respectively. Section 5 evaluates PrivWatcher performance. Section 6 summarizes related work. Finally, we conclude in Section 7.

2. THREAT MODEL AND ASSUMPTIONS

Assumptions: PrivWatcher assumes that it can run within an execution domain that is isolated from the kernel. It also assumes that it can control the kernel's MMU operations and that the kernel's page tables are protected from malicious modifications. PrivWatcher further assumes that kernel is protected against code modification, code injection and return-to-user attacks. It is worth noting that many previous proposals achieved all these properties, for instance by using virtualization (e.g., [50, 46, 18]), or hardware isolated execution domains (e.g., [19, 33]). PrivWatcher also assumes the kernel adopts a protection technique to mitigate the impact of code-reuse attacks (e.g., [27, 26, 45, 40, 15, 44, 31]).

Threat Model: PrivWatcher aims at protecting process credentials and their usage contexts from attackers seeking to corrupt them using memory exploits in order to gain unauthorized privileges for their processes. Given this threat model, attackers can corrupt process credentials through four possible attack vectors. For convenience, we refer to

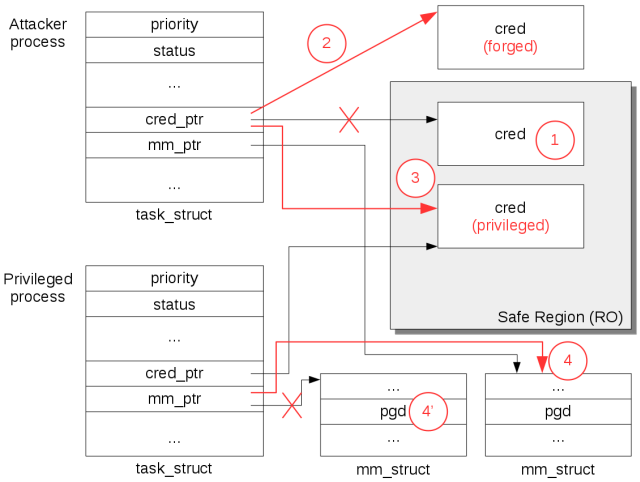

Figure 1: Attack vectors. (1) Overwrite attack: process credentials data are directly corrupted; (2) Forging attack: data pointers are corrupted to refer to forged credentials outside of protected memory; (3) Reuse attack: data pointers are corrupted to refer to existing, validated privileged credentials; (4) and (4') Address space injection attack: malicious binaries are injected into the address space of a privileged process.

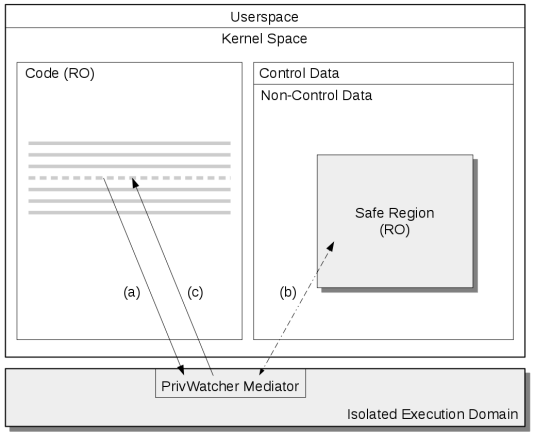

Figure 2: Overview of system architecture. Process credentials are stored in the safe region (non-writable to kernel). (a) Kernel requests to create or update process credentials; (b) PrivWatcher mediates changes to process credentials, allocating a new copy from the safe region if required; (c) Updated process credentials returned to the kernel for use.

them as: 1) *overwrite attack*, 2) *forging attack*, 3) *reuse attack*, and 4) *address space injection attack*. Figure 1 illustrates how each of these attacks can take place. The safe region here represents memory areas protected by PrivWatcher.

Figure 1 illustrates how `task_struct` (i.e., Process Control Block) is organized in Linux. It also shows how process credentials are grouped in a separate structure and referred to by a pointer field in `task_struct`. In an *overwrite attack*, process credentials are directly corrupted to give elevated privileges to attacker processes. For example, the `uid/gid` identifiers of an attacker process can be overwritten with zero values so they are granted root privileges.

The pointer field in `task_struct` can also be corrupted so that it refers to either a credentials structure forged by the attackers outside of the protected memory areas (and therefore not verified by PrivWatcher), or to an existing, privileged credentials structure that has been validated by PrivWatcher in the context of another process. We refer to the former as *forging attack*, and the latter as *reuse attack*.

In an *address space injection attack*, attackers aim to inject malicious binaries in the context of a privileged process. This can be done by corrupting the Page Global Directory (`pgd`) data field to point to a completely forged set of page tables. As shown in Figure 1, the `pgd` field is linked to the `task_struct` through a separate structure (`mm_struct`) that defines the memory map of the process. Either one of the two pointers that link the `task_struct` to the `pgd` can be corrupted to achieve that objective. Note that attackers can also corrupt the page table contents directly, however such attacks are out of scope of this paper since PrivWatcher assumes orthogonal techniques to protect the page tables.

PrivWatcher defends against all of the above attack vectors, sealing off any potential avenues of bypassing the protection. It should be noted that forging attack, reuse attack, and address space injection attack violate TOCTTOU consistency, since the context in which privileged credentials are used (i.e., the attacker process) is different from the context where verification (or the lack thereof, in the case of forging attack) previously took place (i.e., the privileged process).

PrivWatcher can prevent attacks that use memory corruption to directly manipulate process credentials as well as those that do so through simple code reuse techniques, for example by invoking existing kernel functions to allocate privileged credential and assign it to attacker processes. Nevertheless, as mentioned PrivWatcher assumes that the kernel would adopt orthogonal mechanisms to prevent complex (e.g., Turing complete) code reuse attacks. This threat model is realistic because the majority of existing code reuse mitigation techniques focus on coarse grained control flow integrity, which can be effective in preventing sophisticated code reuse attacks but can be subverted by simpler forms of code reuse (e.g., forming malicious arguments and jumping to a kernel function entry point).

3. SYSTEM DESIGN

PrivWatcher prevents privilege escalation attacks that manipulate process credentials via kernel memory corruption. Figure 2 shows the architectural design of PrivWatcher. At a high level, PrivWatcher isolates process credentials into a safe region. By leveraging control over the kernel's MMU operations, the safe region is non-writable to the kernel. PrivWatcher then interposes checks whenever the privilege of a process changes and its credentials need to be updated. These checks verify that the new process privilege does not violate the kernel's access control policies. For example, under POSIX-style Discretionary Access Control (DAC) policy [34] implemented on most Unix/Linux systems, a non-root process is never allowed to become root unless it is executing a root-setuid binary [24].

To effectively harden OS kernels and provide new opportunities for intrusion detection, PrivWatcher needs to achieve

the following objectives in order to address the challenges of protecting process credentials: 1) secure allocation that isolates protected data fields (i.e., process credentials), 2) non-bypassable protection that guarantees the TOCTTOU consistency of process credentials, and 3) effective integrity verification that enforces the original semantics of the kernel's access control policy. In the rest of this section, we describe in detail how PrivWatcher achieves all these objectives. Afterwards, we present an in-depth security analysis of the introduced techniques.

3.1 Isolating Process Credentials

To address the first challenge, PrivWatcher 1) identifies critical non-control data fields relevant to process credentials, and 2) reorganizes the kernel's memory layout so that these fields are aggregated in a well-defined memory region.

3.1.1 Process Credentials in Linux

The Linux kernel maintains its Process Control Blocks (PCB) as the `task_struct` structure. The kernel allocates a separate instance of this structure for each process that it creates. The structure encodes information necessary for process management, such as process identification, process state and memory map. Data fields relevant to process credentials are grouped in another data structure called the credentials structure (`cred`). The privileges of a process are described by an instance of the `cred` structure, which is referred to by a pointer field in `task_struct`.

```
1  struct cred {
2    atomic_t  usage;
3  #ifdef CONFIG_DEBUG_CREDENTIALS
4    atomic_t  subscribers;          /* number of processes subscribed */
5    void    *put_addr;
6    unsigned  magic;
7  #define CRED_MAGIC  0x43736564
8  #define CRED_MAGIC_DEAD  0x44656144
9  #endif
10   kuid_t    uid;                  /* real UID of the task */
11   kgid_t    gid;                  /* real GID of the task */
12   kuid_t    suid;                 /* saved UID of the task */
13   kgid_t    sgid;                 /* saved GID of the task */
14   kuid_t    euid;                 /* effective UID of the task */
15   kgid_t    egid;                 /* effective GID of the task */
16   kuid_t    fsuid;                /* UID for VFS ops */
17   kgid_t    fsgid;                /* GID for VFS ops */
18   unsigned  securebits;           /* SUID-less security management */
19   kernel_cap_t  cap_inheritable;  /* caps our children can inherit */
20   kernel_cap_t  cap_permitted;    /* caps we're permitted */
21   kernel_cap_t  cap_effective;    /* caps we can actually use */
22   kernel_cap_t  cap_bset;         /* capability bounding set */
23   kernel_cap_t  cap_ambient;      /* Ambient capability set */
24 #ifdef CONFIG_KEYS
25   unsigned char  jit_keyring;     /* default keyring to attach
26                                    * requested keys to */
27   struct key __rcu *session_keyring; /* keyring inherited over fork */
28   struct key  *process_keyring;   /* keyring private to this process */
29   struct key  *thread_keyring;    /* keyring private to this thread */
30   struct key  *request_key_auth;  /* assumed request_key authority */
31 #endif
32 #ifdef CONFIG_SECURITY
33   void    *security;              /* subjective LSM security */
34 #endif
35   struct user_struct *user;       /* real user ID subscription */
36   struct user_namespace *user_ns; /* user_ns the caps and keyrings are
37                                    * relative to. */
38   struct group_info *group_info;  /* supplementary groups for
39                                    * euid/fsgid */
40   struct rcu_head  rcu;           /* RCU deletion hook */
41 };
```

Listing 1: Credentials structure in the Linux kernel

As shown in Listing 1, the `cred` structure contains various user ID (`uid`) and group ID (`gid`) fields. In Linux, a `uid/gid` value of 0 (zero) means that the process has root (superuser) privileges and can access all system resources. Furthermore, Android, which is based on Linux, typically reserves `uid/gid` in the range of 1 to 10,000 for system privileged daemons, which are granted more privileges than normal applications.

PrivWatcher protects all instances of the `cred` structure that are referred to by at least one `task_struct`. This includes all `cred` instances that are used by the kernel to represent process privileges. PrivWatcher particularly focuses on protecting the various `uid/gid` fields of each of these instances. This prevents attackers from escalating privileges by influencing any of the monitored fields to take values that are reserved for privileged processes. The same technique can be extended to protect other fields, such as the `security` pointer which defines the contexts of custom security modules (e.g., SELinux [51]). In addition, the `cred` structure contains various types of capability fields that determine if the process is permitted to do certain actions. For instance, the kernel allows processes that have the `setuid` capability more freedom to change their (`uid`) and escalate their privilege.

PrivWatcher requires the kernel to organize its PCB in a canonical layout where data fields relevant to process credentials are separated into a standalone structure, as is the case with Linux (i.e., the `cred` structure). This is because structures occupy contiguous memory addresses at runtime, and thus a standalone credentials structure allows dedicated use of the safe region. Otherwise the entire PCB would need to be protected, which is infeasible because it is updated frequently by multiple subsystems of the kernel.

It is worth noting that on earlier versions of the Linux kernel, data fields related to process credentials (e.g., `uid/gid`) are lumped directly into `task_struct` and therefore does not meet PrivWatcher's requirement of canonical PCB layout. However, by using static code analysis techniques, such as [39, 43], we can easily rearrange the PCB and automate refactoring of the source code.

3.1.2 Moving Process Credentials to the Safe Region

The kernel stores runtime instances of the `cred` structure in dynamically allocated heap memory. The memory allocator for small kernel objects such as `cred` typically acts as a cache on top of the kernel's page-level allocator. Thus, in order to aggregate and protect all instances of the `cred` structure, the kernel should be modified to allocate them in memory pages belonging to the safe region. As mentioned, PrivWatcher guarantees that the safe region is never mapped writable to the kernel.

However, one subtlety arises if we choose to rely on the kernel's allocator. This is because some allocators keep internal states in the same memory pages where objects are allocated from. In such cases, the allocator itself needs write access to the safe region. For example, the Linux kernel's default allocator Slub [13] organizes free objects into a linked list. Whenever objects are deallocated, this free list needs to be updated. Therefore, PrivWatcher would then be forced to mediate the allocator's list maintenance operations.

To avoid this extra complexity in mediating kernel allocator's maintenance operations, as an alternative, allocation from the safe region can also be carried out by PrivWatcher. In order to minimize changes to the kernel, we adopt the following strategy: The kernel continues to use its original allocator for the `cred` structure, but whenever an instance of the `cred` structure is about to be assigned to a `task_struct`, it requests PrivWatcher to make a duplicate from the safe region. We modify the kernel so that it uses the duplicate returned by PrivWatcher, and frees the original unprotected one as appropriate. This approach also has the nice property of allowing us to focus on exactly those `cred` instances that describe process privileges.

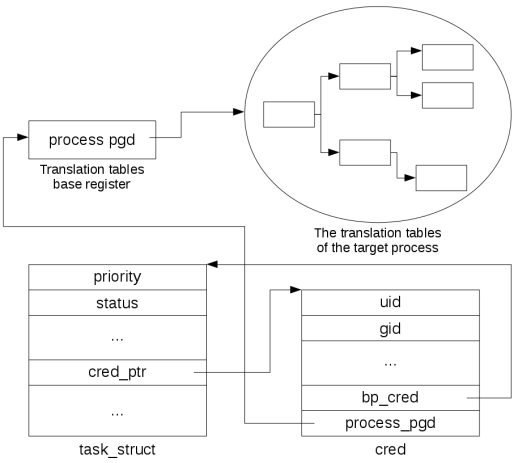

Figure 3: A three-way interconnection that binds together the credentials structure, the PCB and the process address space.

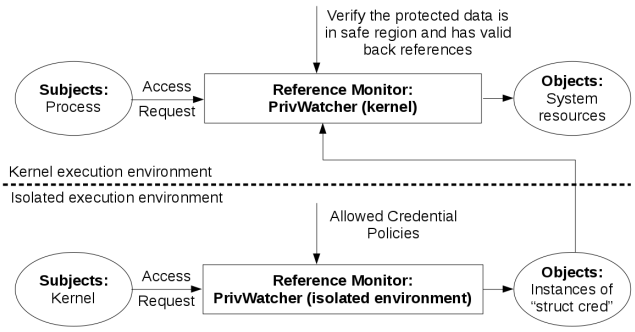

Figure 4: PrivWatcher's dual reference monitor. The kernel-side reference monitor ensures TOCTTOU consistency of process credentials; the isolated environment reference monitor verifies changes to process privileges.

3.2 Ensuring TOCTTOU Consistency

The previous setup prevents direct overwrites to data fields of existing verified credentials (i.e., *overwrite attacks* in our threat model). However, to fully address the challenge of ensuring TOCTTOU consistency, we still need to guarantee that the context of use is the same as the context of verification. A lack of this property leads to potential *forging attacks*, *reuse attacks*, or *address space injection attacks*, as mentioned in Section 2.

To address this challenge, PrivWatcher first creates a secure, unforgeable path that links together process credentials and their presumed context of use (i.e., the process that they are supposed to describe). Then, by adopting a dual reference monitor, PrivWatcher actively verifies the validity of this link to ensure that TOCTTOU consistency for process credentials is preserved. In the following, we present the details of our approach.

3.2.1 Creating a Secure Verification Path

Non-control data fields typically describe entities. These entities in turn determine the context of use. In the case of process credentials, they describe the privileges of a process. There are two contexts, rooted in the current hardware states that represent the currently executing process:

- *The kernel representation:* The kernel represents processes as instances of the `task_struct` structure. In the Linux kernel, the `task_struct` of the currently executing process is obtained by first reading the stack register and locating the top of the current kernel-mode stack, where a pointer gives the address of the current `task_struct`.
- *The hardware representation:* The hardware views processes as individual address spaces, which is defined by a set of address translation page tables (PGT). The base address of the current first level PGT (known in Linux as Page Global Directory, or `pgd`) is loaded into a hardware register (e.g., `cr3` in Intel x86 and `ttbr0` in ARM). To facilitate process switch, the `pgd` of a process is linked to `task_struct`, as shown in Figure 1.

PrivWatcher enforces a strict one-to-one relationship between a process and its credentials. As shown in Figure 3, this is achieved by using a three-way interconnection that links each instance of the `cred` structure to the `task_struct` and address space contexts of the process that it describes. For convenience, we refer to these links as the "back references". Note that the (forward) reference from `pgd` to `cred` is implicit, and that the current hardware `pgd` (e.g., `cr3`) should always agree with the one back referenced by the `cred` of the current process.

Looking further into the implementation of these techniques, PrivWatcher introduces two new fields into the `cred` structure to record back references. This is shown in Figure 3 as `bp_cred` and `process_pgd`. Also, when PrivWatcher decides that a set of process credentials violates the kernel's access control policy, it invalidates the back references by setting them to an invalid value, such as `NULL`, and thus denying the offending process any future accesses to sensitive resources. We next discuss mechanisms for verifying the `cred` back references.

3.2.2 Dual Reference Monitor

Having established the secure link between process credentials and its usage contexts, PrivWatcher uses a dual reference monitor to enforce integrity verification as well as TOCTTOU consistency of process credentials, as shown in Figure 4. PrivWatcher interposes security checks at two different points: 1) when the privileges of a process are being set, and 2) when the kernel performs security checks for a process to access sensitive resources. In the former, process privilege changes are requested by the kernel and are verified in the isolated environment. We present details of this verification in Section 3.3. The latter is carried out in the kernel, and it enforces TOCTTOU consistency of process credentials by augmenting all access control calculations in the kernel to verify the following conditions:

- The process credential actually belongs to the safe region, which guarantees that it has been previously checked by PrivWatcher's reference monitor in the isolated execution environment.

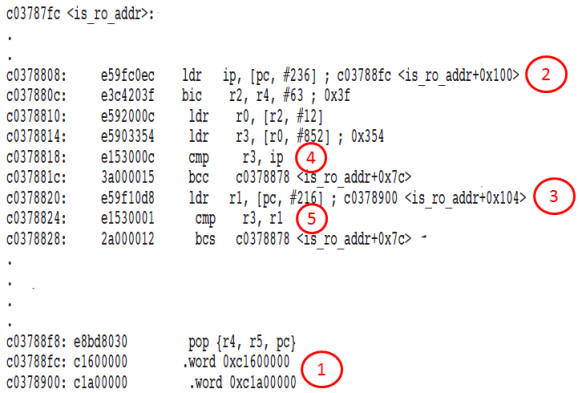

Figure 5: Deterministic verification of input address without relying on data fields. 1-The start and end address of the read-only memory is encoded within the code section. 2&3-The read-only data is read into registers. 4&5-The comparison is done between the input in r3 and values read from the read-only data fields.

- The process credentials correctly back reference the current `task_struct` and that the current hardware page tables base address (e.g., given by registers `cr3` on Intel x86 and `ttbr0` on ARM) also matches the back-referenced value.

If either of these conditions fail, then the access control calculation would return error status (e.g., no permission), thus denying any sensitive accesses.

One technicality needs to be addressed in order to adapt the Linux kernel to using this strategy. In the Linux kernel, access control calculations are not performed at a centralized location but rather scattered throughout the kernel source code. Manually identifying and patching all of them would prove to be infeasible. Instead, we leverage the Linux Security Modules (LSM) framework [55, 57]. LSM places a set of hooks in the kernel code right before sensitive resources are accessed, so that customized access control checks can be implemented. For example, Security Enhanced Linux (SELinux) is implemented on top of the LSM framework [51]. In PrivWatcher, we augmented all LSM hooks to implement the kernel-side reference monitor. We note that on kernel versions without LSM, as an alternative one could also augment all system calls. This still provides strong security guarantees since no action will be performed on behalf of userspace if the process credentials violate TOCTTOU consistency.

3.2.3 Deterministic Data Verification

To guarantee the security of the kernel-side security checks, they have to be conducted without relying on any data fields that can be subjected to memory corruption.

Validating that process credentials belong to the safe region requires PrivWatcher to communicate the memory address ranges of the safe region securely to the kernel. This can be achieved by allocating the safe region from a fixed memory range and exporting the start and end addresses of these pages as symbols to the kernel. These symbols are embedded in the kernel's code sections, and given our assumption of code integrity protection; they are secure from memory corruption. The kernel can then carry out the range check with simple arithmetic operations. Figure 5 shows an example ARM assembly instructions resulting from the verification function. It shows that the verification is done without relying on corruptible data fields.

The kernel typically needs to traverse a chain of memory dereferences in order to retrieve the current `task_struct`. Attackers could therefore interfere with any link in this chain to influence the final result. To prevent this, we adopt the following strategy: we maintain a secure mapping between the CPU core and its current `task_struct`. On process switches, PrivWatcher is invoked to synchronize this secure mapping, which, just like the safe region, is guaranteed to be tamper-proof against the kernel. The secure mapping can be implemented as an array that maps CPU core number to its current `task_struct` (core number can be obtained using hardware-specific instructions that doesn't depend on memory contents, e.g., x86's `cpuid` [36]). Note that since we assume kernel control data to be protected, attackers cannot divert the kernel control flow to reuse this update operation.

3.3 Enforcing Integrity Protection

Up to this point, we have established PrivWatcher's secure allocation of protected data and its non-bypassable monitoring techniques. This framework allows PrivWatcher to enforce policies that preserve the kernel's original semantics of its runtime operations. PrivWatcher can adapt any type of policy that defines the credential of user processes. Hence, PrivWatcher is a very useful tool for high security systems where each process is linked to a predefined task at a particular privilege. In addition, PrivWatcher can be perfectly adapted to app-based systems like Android, where a predefined set of processes run as root and system privilege levels; each has its defined `uid` and `gid`. Meanwhile, all apps run with less privileged credentials. In the case of Android, a straightforward policy would be to prevent unprivileged apps from using credentials that grant root or system privileges. This should in fact be the preferred policy since normal operations of Android does not require its users to exercise administrator privileges [17].

In this section, we present a sample policy that targets general-purpose Linux systems. Our sample policy prevents generic privilege escalation attacks. PrivWatcher enforces that process privileges as defined by the DAC `uid`/`gid` cannot be escalated (i.e., non-root → root), except for processes that have `exec`'d whitelisted root-setuid binaries (e.g., `passwd`). This policy can be easily extended, e.g., to protect the range of `uid`/`gid` values reserved for privileged Android system daemons. Our approach described next can also be adapted to other access control models (e.g., SELinux [51]), and we will discuss more in Section 3.4 about this.

3.3.1 Monitoring for Privilege Escalation

PrivWatcher monitors for privilege escalation, which happens when an unprivileged process attempts to manipulate its `cred` structure to get elevated privilege. We therefore need a precise notion of a "privileged" `cred` structure that correctly reflects the DAC model enforced by the kernel.

As mentioned, the Linux `cred` structure includes four user identifiers: `uid`, `euid`, `suid`, and `fsuid`. The group identifiers are similar in concept and we omit their description for brevity. We view a `cred` structure as privileged if any one

of the `uid`, `euid`, or `suid` fields is equal to the root user ID (i.e., zero). This is also the same semantics as defined by the Linux kernel [24]. For simplicity, we do not include checks to the capability fields defined in the `cred` structure. We make a straightforward assumption that unprivileged processes does not have the `setuid` capability and hence are not allowed to escalate their privilege.

3.3.2 Legitimate Privilege Escalation

Given the description of a `cred` structure that grants elevated privileges, it is easy to enforce a strict policy that only allows process privileges to be dropped (i.e., root → non-root) but not the other way around (i.e., non-root → root). This prevents all privilege escalation exploits that manipulate process credentials. As mentioned, this should be the preferred policy on systems such as Android. On the other hand, the DAC model does indeed allow for, but needs to strictly control, privilege escalation from non-root to root. This is done by using root-setuid binaries [24]. They are useful in exporting to unprivileged users a small, fixed set of functionalities that require elevated privileges to complete (e.g., the password-setting program `passwd`).

Allowing for legitimate root-setuid binaries essentially requires PrivWatcher to authenticate processes based on the binaries they are executing, which is in turn determined by their physical address spaces. Our three-way interconnection among the `cred` structure, the `task_struct` structure, and the process address space (Section 3.2.1) readily allows this authentication to take place. Previous works (e.g., [18, 42]) have already provided detailed techniques on authenticating process binaries based on their physical address spaces. We only remark that to support legitimate privilege escalation via root-setuid binaries, only the initial process that performed the escalation needs to be authenticated. Afterwards, its descendant processes inherit privileges from it. In particular, this allows root-setuid binaries such as `sudo` to function properly (the command "`sudo cmd`" allows `cmd` to inherit privileges from `sudo` and run as root).

3.4 Security Analysis

PrivWatcher's design allows it to address the attack vectors identified in Section 2. Firstly, the read-only mapping of the safe region prevents validated credentials from being modified directly by the kernel (overwrite attack). The unique interconnection linking together the `cred` structure, the `task_struct` structure, and process address space allows PrivWatcher to achieve TOCTTOU consistency. In particular, preventing the use of forged credentials can be achieved by confirming that the credentials indeed belong to the safe region (forging attack); also, attackers cannot reuse existing privileged credentials because the back-referenced `task_struct` would be different (reuse attack); finally, replacing the address space of a privileged process can be detected by verifying that the back-referenced `pgd` matches the value given by the hardware registers, such as `cr3` on Intel and `ttbr0` on ARM (address space injection attack).

In the rest of this section, we examine the security guarantees provided by PrivWatcher. Afterwards, we discuss the current state of OS kernel security solutions and how PrivWatcher can advance real world systems security. We then discuss the effectiveness of PrivWatcher against existing kernel attacks. Finally, we present a plan for future work to enhance and generalize PrivWatcher's protection.

3.4.1 Security Guarantees

Security researchers predominantly agree that protecting the security of the kernel requires a reference monitor that runs in isolation from the kernel itself. The goal is to prevent potential memory corruption vulnerabilities in the kernel from compromising the reference monitor, which has its own memory address space.

PrivWatcher introduces a set of novel techniques that exclusively relies on the fact that the kernel code section is integrity protected. Thus, it guarantees that access control checks performed at the kernel side cannot be circumvented. This ensures strict enforcement of TOCTTOU consistency so that the context in which protected data fields are verified is the same where they are used.

Utilizing existing reference monitor and kernel code integrity techniques allows PrivWatcher to raise the bar of kernel security protection. Nevertheless, it tied the security of PrivWatcher to the underlying tool that provides the assumed security properties. If the isolated domain that hosts PrivWatcher is compromised, then PrivWatcher will be as well. Similarly, PrivWatcher will be bypassed if kernel code integrity protection is compromised, allowing an attacker to nullify the kernel side TOCTTOU consistency checks.

Enforcement of the DAC policy semantics, presented in Section 3.3, relies on the amount of control the reference monitor has over the kernel. The presented policy assumes the reference monitor is aware of which applications are allowed to raise their privileges and that it can detect malicious binaries injected in userspace code using any of the plentiful existing research in this direction [18, 42, 46, 37]. These proposals often rely on techniques that monitor and protect the entire page tables' hierarchy of the system. Hence, it is reasonable to assume that tying the credential information to the base of the process page tables, as described in Section 3.3, is sufficient to identify the binary that a process is executing.

It is important to note that *adopting an orthogonal system to identify running processes is a requirement of the sample policy, but not a requirement of PrivWatcher.* This policy was selected to support the argument that PrivWatcher can be adaptable to any general-purpose system. We anticipate that app-based systems should require a simpler policy. For instance, Android typically assigns fixed `uid` values between 0 and 10,000 to its root and system processes and `uid` values greater than 10,000 for regular apps. Hence, an Android privilege escalation prevention policy can state that regular apps, identifiable by process `uid` greater than 10,000, cannot change their `uid` to a value reserved for privileged applications. This policy will be effective in preventing privilege escalation assuming that app processes (including attacker-controlled ones) are assigned the proper unprivileged `uid`s by an uncompromised system.

3.4.2 Kernel Security Landscape

The majority of existing kernel security solutions focused on two problems: preventing kernel code modification and injection, and preventing corruption of control data. The landscape of existing solutions made it critical to adopt a solution like PrivWatcher for the following reasons:

First, the is an absence of solutions that provide non-control data protection for the kernel. There are few exceptions to this absence, with the most notable being Kenali [52], which adopts data flow integrity techniques to pro-

tect non-control data fields. Nevertheless, Kenali lacks the notion of preserving TOCTTOU consistency of the kernel's non-control data, particularly regarding the context of use. Section 6 presents a more detailed discussion about the difference between PrivWatcher and Kenali.

Second, existing solutions that target either code or control-data integrity have critical limitations:

Existing kernel code integrity solutions [18, 42, 46, 37, 28, 50] can be deemed irrelevant if an attack escalates the privilege of a malicious process. The malicious process can then have access to all system resources despite the fact that the kernel maintains its code integrity. Unfortunately, existing tools cannot be easily adaptable to prevent such attacks because they lack the required semantic knowledge about the location, context of use, and proper verification techniques of credentials data. PrivWatcher extends these tools to prevent privilege escalation, which is a great threat to systems protected by these existing techniques.

Existing control-data integrity solutions [27, 26, 45, 40, 15, 44, 31] focus on coarse grained rather fine grained control-flow integrity. As a result, their effectiveness is limited to preventing sophisticated code reuse attacks, which opens the door to simpler form of code reuse, such as passing malicious arguments to legitimate function entry point to corrupt non-control data. Davi et al. [29] proved that many of the existing control-data protection techniques can be bypassed by a Turing complete attack, which testifies to the imperfect status of these solutions and how far they are from stopping simple code-reuse attacks. A Turing complete code reuse attack could be as powerful as privilege escalation. Nevertheless, improving existing control-data integrity solutions to eliminate that threat is out of the scope of this paper.

3.4.3 Effectiveness Against Attacks

In this section, we argue that recent kernel attacks necessitates the adoption of a privilege escalation prevention tool like PrivWatcher. They also emphasize the gap in existing solutions discussed in the previous section.

PingPong Root [11, 56], which relies on CVE-2015-3636, uses a typical use-after-free kernel bug to control a dangling file descriptor. After the exploit, the attacker controls the function pointer that closes the file and can use it to jump to arbitrary location in the kernel. Privileged Execute Never (PXN) protection limits the attack from returning to userspace code, so attackers resolve to jump-oriented programming to overwrite the `addr_limit` kernel data field so they can have access to and overwrite process credentials.

QuadRooter [12] uses a similar approach. It starts by exploiting the race condition vulnerability CVE-2016-2059 to control a function pointer that will eventually call the kernel function `commit_creds` with malicious parameters to set the `UID` of the attacking process to 0.

KNOXout [10] and iovyroot [7] both exploit CVE-2015-1805 to overwrite the `ptmx_fops->check_flags` kernel function. The function pointer would either be used to jump to a gadget to overwrite `addr_limit` and directly rewrite the credentials, in the case of iovyroot, or to jump to the kernel function `override_creds` to override the credentials with a privileged `uid`, in the case of KNOXout.

PingPong Root, QuadRooter, KNOXout and iovyroot share the same exploitation technique. All of these exploits work in roughly two stages: 1) they use the memory corruption vulnerability to divert kernel control flow and execute some attacker-specified shell code; and 2) the shell code then manipulates process credentials (e.g., overwrites them) in order to give elevated privileges to attacker processes. They all failed to overcome kernel code integrity protection and they all did not need to mount sophisticated code reuse exploitation to compromise the system. In fact, directly calling a legitimate kernel functions with malicious parameters, like QuadRooter, would likely bypass all existing code reuse defenses. Even if there exist a tool that can completely eliminate code reuse attacks, the underlying memory corruption vulnerabilities would still allow attackers to directly carry out the second stage attack without triggering code integrity or CFI protection.

All these exploits perfectly fall within the threat model defined in Section 2. Given that PrivWatcher implements a proper policy that prevents malicious values to be written to the credential data fields, it will be able to stop all these exploits. It can successfully detect and prevent the second stage when attackers attempt to manipulate process credentials (Section 3.4). It therefore would have prevented all of these exploits from successfully gaining root privileges.

These exploits also demonstrate that, in practice, process credential is the preferred target of attackers. We can expect this to remain a significant threat as more defense techniques get enabled by default for the kernel (e.g., [9]) and correspondingly attackers begin carrying out pure non-control data attacks.

3.4.4 Future Work

We have demonstrated how PrivWatcher can stop existing kernel attacks. Nevertheless, once PrivWatcher is widely adopted, attackers are likely to either corrupt other non-control data fields, or to find ways to overcome the adopted security policy.

Applying PrivWatcher to other types of non-control data, such as the credentials of `inode` structures, the secure link between them and their usage contexts need to be precisely defined. As a future work, we will explore how to utilize automated techniques, such as static source code analysis (e.g., [38]), to achieve this goal.

In terms of supporting other access control policies enforced by the kernel, one potential candidate is SELinux. SELinux is a mandatory access control (MAC) policy that assigns every subject (e.g., processes) and every object (e.g., files) a security label. A policy file, written by the administrator or generated automatically, gives the allow and deny rules based on labels. The policy file is loaded into the kernel for enforcement. PrivWatcher can be easily adapted to protect SELinux credentials and enforce their semantics by consulting the policy file. In this regard, we plan to investigate applying PrivWatcher mechanisms to protect more policy-specific credentials.

4. PROTOTYPE IMPLEMENTATION

We implemented a PrivWatcher prototype for Ubuntu Linux running on an x86_64 platform. Our prototype uses QEMU with KVM hardware acceleration as the securely isolated execution domain (QEMU/KVM). Being a hypervisor, QEMU/KVM already provides the ability to intercept guest kernel's MMU operations. We note that adopting a hypervisor is only an implementation decision: as long as PrivWatcher's assumptions are met, any isolated execution domain can be used.

Apart from our description in Section 3, here we cover additional kernel-side changes handling several technicalities in the Linux kernel. Overall, our prototype added about 750 lines of code (LoC) to the kernel source code.

Reserving the Safe Region: We modified the kernel linker script for a large enough slab of contiguous virtual memory addresses. It supports the maximum number of processes in the system, as reported by `/proc/sys/kernel/pid_max`. The corresponding physical address for this slab can be statically determined and PrivWatcher guarantees it is never mapped writable to the kernel.

Mediating Updates to Process Credentials: PrivWatcher needs to change the way process privileges are updated so that it can mediate them. The Linux kernel documentation states that a process can only change its own privileges and does so through a read-copy-update (RCU) approach: the updated credential is a modified copy of the current one, and is committed to the process by using three well-defined APIs (i.e., `commit_creds`, `override_creds`, and `revert_creds`). These are therefore the only places at which the privilege of a process is updated, and as mentioned we insert a call to PrivWatcher for verification at these points.

Managing Protected Credentials: We need to handle two technicalities for the `cred` structure. First, the `cred` structure is reference counted, and the reference counter is an integer field within the `cred` structure. PrivWatcher would therefore forbid the kernel from updating the counter. We solve it by replacing it with a pointer to an external counter and then refactor the kernel code accordingly (only two API functions handle counter operations, i.e., `get_cred` and `put_cred`). Second, Linux allows a single `cred` instance to be shared among multiple processes. However, given that PrivWatcher needs to ensure a one-to-one relationship between credentials and the process it describes, we disabled this sharing. We require every `task_struct` (i.e., the PCB) to have it own copy of the `cred` structure.

The `init` Process: The `pgd`, as well as the `task_struct` and `cred` structures of the `init` process are not allocated during runtime but are instead hard-coded. We handle this corner case by hard-coding the back references of `init`'s `cred` structure to their proper values. We also modify the linker script so that `init`'s `cred` structure resides in a separate page which is considered as part of the safe region.

Kernel Threads and Interrupt Context: When the Linux kernel is running in the context of a kernel thread or handling an interrupt, the interconnection between the active process PCB and mapped page tables does not hold. A kernel thread is a task that is running only in the kernel context, without a userspace counterpart. Hence, the kernel does not need to update the page tables base pointer when a kernel thread is scheduled. Similarly, when an interrupt is received, the kernel switches to a different context to handle the interrupt, while the page tables base pointer would be still pointing to the previously interrupted process. PrivWatcher handles this corner case while implementing the LSM hooks discussed in Section 3.2. The LSM verifies that the kernel is not running in either a kernel thread or an interrupt context before enforcing the interconnection between the active process and the mapped page tables.

5. PERFORMANCE EVALUATION

In this section, we present micro-benchmarks and application benchmarks for performance evaluation of PrivWatcher. We focus on evaluating the additional overhead introduced (i.e., on top of the assumed security properties). The measurements were made on a system with Intel Core i7-2600 CPU at 3.4GHz and 16GB RAM.

We note here that because of PrivWatcher's assumptions (Section 2), the end-to-end performance overhead relative to an unmodified kernel should be understood as that of PrivWatcher, plus those introduced by code integrity and control data (or CFI) protection. Nevertheless, as we establish in this section, the *additional* overhead of PrivWatcher, relative to the assumed protections, is negligible.

To best capture PrivWatcher's overhead, we chose to evaluate using a comparative approach. First, we measure its overhead relative to an unmodified kernel without any assumed protections. We expect a higher overhead in this case, given that the runtime cost of PrivWatcher monitoring is not shadowed by other security tools. We then measure against a rather naive version of code integrity protection: a system that traps *every* page table update to ensure that 1) only the virtual addresses of the kernel's code section can be mapped with supervisor (e.g., Ring 0) execution privileges, and 2) the kernel's code section is never mapped writable. Additionally, we ensure that the `SMEP` bit in `cr4` is always set, so that instruction fetches from the userspace are forbidden in kernel mode [36]. This naive implementation provides basic code integrity protection as required by our threat model (Section 2), and the technique used is similar to many previous works (e.g., [50, 19, 28]). We expect PrivWatcher's additional overhead relative to this implementation of code integrity protection to be smaller.

We note that our code integrity implementation is purely for the purpose of measuring PrivWatcher performance and is not intended to be bullet-proof. Its performance is also not optimized and the results should not be interpreted as a prediction of the end-to-end aggregate overhead of PrivWatcher plus the assumed protections.

We omit evaluation of PrivWatcher overhead relative to CFI protection for two reasons: 1) currently Linux kernel space ASLR only randomizes the kernel code load location at boot time [9] and therefore does not introduce runtime overhead; 2) since most CFI protection mechanisms rely on (or ensure) code integrity protection, and that as we demonstrate the overhead of PrivWatcher itself is small, the additional performance cost of CFI will dominate and dwarf that of PrivWatcher.

In the following we refer to the overhead of only running PrivWatcher as "PW NoTrap", the overhead of our implementation of code integrity protection as "PGT Trap", and finally the combined overhead of PrivWatcher and code integrity protection as "PW Trap". The results shown are overhead values relative to the benchmark scores of an unmodified kernel.

5.1 Micro-benchmarks

Table 1 shows the results of evaluating PrivWatcher using UnixBench [14]. For all test cases except `execl`, process creation, and shell scripts, there was no significant difference in performance. These test cases involve intensive computation or I/O operations and do not trigger events that need to be monitored (i.e., changes to process privileges and PGT up-

	PW NoTrap %	PGT Trap %	PW Trap %
Dhrystone2	-0.1	-0.1	-0.2
Whetstone	-1.1	-0.7	-3.7
Execl	23.3	91.5	94.4
File Copy 1024B	0.0	-0.2	-2.2
File Copy 256B	0.0	0.2	1.1
File Copy 4096B	-0.9	0.1	-2.7
Pipe	1.3	0.3	1.3
Process Creation	22.3	93.3	95.1
Shell Scripts (8)	11.0	85.4	90.1
System Call	-0.5	-0.3	3.7

Table 1: UnixBench results, showing overhead in percentage relative to an unmodified kernel, the lower the better.

	PW NoTrap %	PGT Trap %	PW Trap %
bzip2	1.2	-1.0	1.6
gzip	1.1	1.3	2.2
ApacheBench	0.0	-0.7	-1.0
kcbench	1.5	56.1	56.8

Table 2: Application benchmark results, showing overhead relative to an unmodified kernel, the lower the better. The values shown are averages over 10 runs, with standard deviation less than 5% for all benchmarks.

dates). The differences shown are likely within the margin or error of our experiments.

In the case of `execl` and process creation, we see an overhead of about 22% for PW NoTrap, and about 94% for PW Trap. Both of these operations involve installing a new `cred` structure to processes and therefore PrivWatcher needs to be invoked to check for any potential malicious privilege escalations. As can be seen, most of the overhead for PW Trap is due to trapping PGT updates (PGT Trap). The relative overhead of PW Trap to PGT Trap is only 3.2% and 1.9% for `execl` and process creation, respectively.

For both PW NoTrap and PW Trap, the overhead of the shell script test closely mirrors that of `execl` and process creation. This is expected, since shell scripts typically need to invoke multiple processes. Consequently, the cost of process creation and program loading becomes a dominant factor.

5.2 Application Benchmarks

To evaluate PrivWatcher's impact on overall system performance, we experimented with two file compression applications (`bzip2` and `gzip`), `ApacheBench` [1] (a web benchmarking tool), as well as a kernel compilation benchmark (`kcbench` [8]). These applications offer a mixture of computation, I/O and network operations, process/thread creation, and program loading that mimics a typical real-world workload. Specifically, `bzip2` and `gzip` were used to compress a 107MB file. ApacheBench was used to run 100 `GET` requests to the Google main page. The kernel compilation benchmarking tool `kcbench` was run on the Linux mainline kernel source tree with a concurrency level of 1 (`-j1`).

Table 2 shows the application benchmark results. As expected from micro-benchmark, for compute, I/O, or network-intensive applications (`bzip2`, `gzip`, and ApacheBench), neither did PW NoTrap, PGT Trap, or PW Trap introduce significant overhead. On the other hand, it is worth mentioning that even for `kcbench`, the overhead of PW NoTrap is not distinguishable from the I/O and network benchmarks, and the relative overhead of PW Trap to PGT Trap is only about 1%. Kernel compilation requires spawning a large number of processes to compile each individual source files (`gcc`), as well as to link the resulting object files together (`ld`). It therefore needs to frequently trigger operations monitored by PrivWatcher. This indicates that the overhead of PrivWatcher is negligible at the application level.

6. RELATED WORK

Data-Flow Integrity: Data-flow integrity (DFI) was proposed as a generic defense against software attacks [23]. Using static program analysis, it computes a data-flow graph (DFG) that represents define-use relationships. To enforce the DFG, runtime checks are inserted before read operations to verify that the memory location is defined by legitimate instructions. However, its protection accuracy is limited by the precision of static analysis, and complete DFI enforcement incurs a high overhead. Recent works applied DFI to kernel space. Kenali [52] uses static source code analysis to identify sensitive data fields that are relevant to security checks; any pointers that can refer to the sensitive fields are also considered sensitive. Instead of checking on memory reads, integrity verification is done only on write operations (a variation of Write Integrity Testing [16]), in order to avoid full DFI enforcement overhead.

The main differences between PrivWatcher and Kenali are: 1) *How write accesses are mediated*: Kenali instruments all write instructions so that only intended memory *locations* (as specified by the DFG) can be modified, while PrivWatcher explicitly verifies that the new *values* being written do not violate the security policy. 2) *How TOCTTOU consistency is preserved*: Kenali protects all pointers that can potentially refer to sensitive data fields, hence it prevents simple redirection attacks that violate TOCTTOU consistency (e.g., the pointer in `task_struct` cannot be redirected to refer to another `cred` structure). Nevertheless, Kenali does not protect the saved `pgd` address, which is critical to preventing address space injection attacks, as mentioned in Section 2. Likewise, although Kenali randomizes the location of kernel-mode stack, the saved `sp` register (in this case an index into a redirection table) is not protected. Consequently, attackers can still influence the current `task_struct` (and thus the current `cred`) retrieved by the kernel (see Section 3.2.3). On the other hand, PrivWatcher explicitly links together the context of verification and the context of use and actively verifies this link before protected data fields are used, providing a strong guarantee for TOCTTOU consistency.

Data Integrity Protection: Sentry [53] is another related work that aims to integrity protect critical kernel data fields. It introduced the idea of partitioning kernel data structures so that their security-sensitive fields are aggregated into a single `struct` definition, and replaced in the original structure with a pointer reference. The new structure is then allocated in protected memory pages that are non-writable to the kernel. To verify write accesses, Sentry checks whether

the value of the current program counter is within the kernel address space. This check is inadequate against kernel-level memory corruption. Unlike PrivWatcher, there is no guarantee for TOCTTOU consistency. In particular, attackers are free to modify the (after partitioning) referencing structure, which is not protected.

Data Integrity in Rootkit Detection: Previous works [21, 22, 30, 41, 47] have also applied non-control data protection to detecting rootkits. All of these techniques are based on scanning memory snapshots for anomalies in kernel data. As such, they can only detect attacks after-the-fact, and suffer from TOCTTOU inconsistencies whereby an attacker scrubs traces of intrusion between two consecutive scans. PrivWatcher guarantees the TOCTTOU consistency of the protected data fields. Its event-driven nature also means PrivWatcher can *actively* prevent attacks.

In addition, previous approaches proposed policy based approaches to protect the semantic integrity of kernel data structures. However, the invariants proposed by previous work, such as [47] and [21], are best suited for expressing the relationships between data structures, e.g., that differences between the process list and CPU run queue are forbidden. None of the previous approaches can be used to describe malicious privilege escalation, as required by the sample DAC policy presented in Section 3.3.

7. CONCLUSION

In this paper we presented PrivWatcher, a framework that allows integrity verification and protection of process credentials against memory corruption attacks. We proposed techniques to overcome the challenges of isolating, verifying, and ensuring the TOCTTOU consistency of credentials data fields. We applied PrivWatcher to ensure the correct DAC semantics enforced by the Linux kernel and prevent unauthorized processes from elevating their privileges. An implementation of PrivWatcher on Intel x86-64 Ubuntu is given. Our experimental evaluation showed that the overhead of PrivWatcher monitoring is small. It is therefore suitable for practical deployment. Our future work will involve adapting PrivWatcher to other forms of kernel access control policies and applying PrivWatcher principles to protect other types of critical kernel non-control data.

8. REFERENCES

[1] ApacheBench. https://httpd.apache.org/.

[2] CVE-2013-2596. http://www.cvedetails.com/cve/CVE-2013-2596.

[3] CVE-2013-6282. http://www.cvedetails.com/cve/CVE-2013-6282.

[4] CVE-2014-3153. http://www.cvedetails.com/cve/CVE-2014-3153.

[5] CVE-2015-3636. http://www.cvedetails.com/cve/CVE-2015-3636.

[6] CVE-2016-0728. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2016-0728.

[7] iovyroot. https://github.com/dosomder/iovyroot.

[8] kcbench. https://github.com/knurd/kcbench.

[9] Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[10] KNOXout. http://www.vsecgroup.com/single-post/2016/09/16/KNOXout---Bypassing-Samsung-KNOX.

[11] PingPong Root. http://pingpongroot.co/.

[12] QuadRooter. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Adam-Donenfeld-Stumping-The-Mobile-Chipset.pdf.

[13] The SLUB allocator. http://lwn.net/Articles/229984.

[14] UnixBench. https://github.com/kdlucas/byte-unixbench.

[15] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[16] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[17] Android. System and kernel security. http://source.android.com/devices/tech/security/overview/kernel-security.html.

[18] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. HIMA: A hypervisor-based integrity measurement agent. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 461–470. IEEE, 2009.

[19] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: real-time kernel protection from the ARM TrustZone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[20] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 38–49. ACM, 2010.

[21] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 77–86. IEEE, 2008.

[22] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 555–565. ACM, 2009.

[23] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160. USENIX Association, 2006.

[24] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *USENIX Security Symposium*, pages 171–190, 2002.

[25] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security*, volume 5, 2005.

[26] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. Deng. ROPecker: A generic and practical approach for defending against ROP attack. In *Network and Distributed System Security Symposium*, 2014.

[27] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity

operating system kernels. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 292–307. IEEE, 2014.

[28] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, V. Adve, S. K. Sahoo, C. Geigle, B. Ding, Y. He, Y. Wu, et al. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206. ACM, 2015.

[29] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, 2014.

[30] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577. ACM, 2009.

[31] I. Fratric. ROPGuard: runtime prevention of return-oriented programming attacks (2012).

[32] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 179–194. IEEE, 2016.

[33] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing kernel code integrity on the TrustZone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[34] A. Grünbacher. POSIX access control lists on Linux. In *USENIX Annual Technical Conference, FREENIX Track*, pages 259–272, 2003.

[35] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.

[36] Intel. Intel 64 and and IA-32 architectures software developer's manual, Volume 2.

[37] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM, 2007.

[38] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[39] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[40] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208. ACM, 2010.

[41] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[42] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, pages 243–258, 2008.

[43] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228. Springer, 2002.

[44] V. Pappas. kBouncer: Efficient and transparent ROP mitigation. *Apr*, 1:1–2, 2012.

[45] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security*, pages 447–462, 2013.

[46] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 233–247. IEEE, 2008.

[47] N. L. Petroni Jr, T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Usenix Security*, 2006.

[48] N. L. Petroni Jr and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115. ACM, 2007.

[49] Samsung. White paper: An overview of Samsung KNOX, 2013.

[50] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *ACM SIGOPS Operating Systems Review*, 41(6):335–350, 2007.

[51] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1(43):139, 2001.

[52] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[53] A. Srivastava and J. Giffin. Efficient protection of kernel data structures via object partitioning. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 429–438. ACM, 2012.

[54] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou. SecPod: a framework for virtualization-based security systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pages 347–360. USENIX Association, 2015.

[55] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, 2002.

[56] W. Xu and Y. Fu. Own your Android! yet another universal root. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[57] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *USENIX Security Symposium*, pages 33–48, 2002.