PrivAnalyzer: Measuring the Efficacy of Linux Privilege Use

John Criswell
University of Rochester
criswell@cs.rochester.edu

Jie Zhou
University of Rochester
jzhou41@cs.rochester.edu

Spyridoula Gravani
University of Rochester
sgravani@cs.rochester.edu

Xiaoyu Hu
BitFusion.io Inc.
hxy9243@gmail.com

***Abstract*—Operating systems such as Linux break the power of the root user into separate privileges (which Linux calls capabilities) and give processes the ability to enable privileges only when needed and to discard them permanently when the program no longer needs them. However, there is no method of measuring how well the use of such facilities reduces the risk of privilege escalation attacks if the program has a vulnerability.**

This paper presents PrivAnalyzer, an automated tool that measures how effectively programs use Linux privileges. PrivAnalyzer consists of three components: 1) AutoPriv, an existing LLVM-based C/C++ compiler which uses static analysis to transform a program that uses Linux privileges into a program that safely removes them when no longer needed, 2) ChronoPriv, a new LLVM C/C++ compiler pass that performs dynamic analysis to determine for how long a program retains various privileges, and 3) ROSA, a new bounded model checker that can model the damage a program can do at each program point if an attacker can exploit the program and abuse its privileges. We use PrivAnalyzer to determine how long five privileged open-source programs retain the ability to cause serious damage to a system and find that merely transforming a program to drop privileges does not significantly improve security. However, we find that simple refactoring can considerably increase the efficacy of Linux privileges. In two programs that we refactored, we reduced the percentage of execution in which a device file can be read and written from 97% and 88% to 4% and 1%, respectively.

I. Introduction

Commodity operating systems such as Windows [1] and Linux [2] mediate a process's access to system objects such as files, pipes, and sockets. Significant research has been devoted to designing access controls and privilege models which reduce the damage that an application can cause should an attacker exploit a vulnerability within the application. Examples of such access controls include Unix discretionary access control [2], [3], fine-grained privileges first developed for compartmented mode workstations [4] and now deployed in Windows and Linux [1], [2], and capabilities such as those found in Capsicum on FreeBSD [5].

Linux privileges (called capabilities) [6] are intended to reduce the risk of privilege escalation attacks, and they are increasingly being used in the deployment of inter-application isolation mechanisms such as Linux containers [7]. Docker [8], a major container platform, uses Linux privileges to reduce the likelihood of a bug in a container leading to a privilege escalation exploit. Docker removes the need for privileged containers by assigning only the necessary capabilities for launching execution and allows for dynamic addition/removal as necessary [8]. Unfortunately, it is unclear how to effectively use capabilities and benefit from such a design.

Currently, programmers manually reason about how long their programs retain privileges and what damage an attacker could do with those privileges if the program has an exploit. Such reasoning must be done for every possible type of attack and must account for the process's privileges, user IDs, group IDs, and the owners, groups, and permissions of directories, files, devices, and other objects. Manual analysis is error-prone and fails to scale. It also fails to provide a quantifiable metric that can be used to compare different software designs. While there are tools that evaluate the efficacy of mandatory access controls [9], [10], there are no tools, to the best of our knowledge, that evaluate how much security improves when real programs use Linux privileges. Consequently, the value of Linux privileges is unknown. This may be why many developers forgo using them.

To address this problem, we have developed an automated tool named *PrivAnalyzer*. PrivAnalyzer consists of three components: 1) AutoPriv, 2) ChronoPriv, and 3) ROSA. AutoPriv [11] is an existing compiler that computes which privileges are still usable by a program at each program point and transforms the program to permanently drop privileges when no longer needed. We feed AutoPriv's output to the second component, ChronoPriv, our new compiler pass that records the number of instructions executed with a specific privilege set along with the process's user IDs and group IDs. The third component, ROSA (Rewrite of Objects for Syscall Analysis), is a new bounded model checker, written in Maude [12], that models the Linux system call API and its privileges. Given a set of system calls that an application is allowed to make, the number of times it is allowed to make each system call, and the privileges each system call is allowed to use, ROSA can determine whether an application could put the system into a specified compromised state. ROSA models processes, directories, files, sockets, and a subset of system calls that operate upon these objects. ROSA takes ChronoPriv's output as its input along with a description of a compromised system state and decides whether the program, if it were compromised, could put the system into the compromised state. Altogether, PrivAnalyzer can determine whether a vulnerability could put the system into a compromised state and for how long the program poses such a risk. Programmers can use PrivAnalyzer to determine the ramifications of using

various privileges within their programs.

Using PrivAnalyzer, we modeled four privilege escalation attacks on five open-source programs that were modified to use Linux privileges. Our results show that simply adding code to enable privileges when needed and disabling them permanently when no longer needed is insufficient; programs such as `passwd` and `su` retain powerful privileges for over 88% of their execution that an attacker could use to read and write device files such as `/dev/mem`. Subsequently, we manually analyzed two of these programs and discovered that the poor results are primarily due to design decisions motivated by the fact that most Unix systems only have a root user and no privileges. With some simple refactoring, we modified `passwd` and `su` so that they used these powerful privileges for only 4% of their execution, significantly decreasing the window of opportunity in which an attacker could mount a privilege escalation attack on these programs.

To summarize, our contributions are as follows:

- We present an automated tool, PrivAnalyzer, that measures how effectively programs use Linux privileges. It aims to help security-critical software developers to minimize privileges use. We describe the design and implementation of the ChronoPriv dynamic analysis and the ROSA bounded model checker which we added to AutoPriv [11] to create PrivAnalyzer.
- We evaluate the performance of the ROSA bounded model checker.
- We use PrivAnalyzer to quantitatively show that merely using the minimum set of Linux privileges is insufficient in mitigating privilege escalation attacks. Programs such as `passwd` and `su` maintain powerful privileges for 88% or more of their execution.
- We study and refactor two programs to use Linux privileges more effectively. PrivAnalyzer shows that these refactored programs use powerful privileges for 4% of their execution.
- We summarize two lessons learnt from refactoring `passwd` and `su` that should help programmers write software that better resists privilege escalation attacks.

The rest of this paper is organized as follows: Sections II, III, and IV describe Linux privileges, our attack model, and background on building model checkers in Maude. Sections V and VI describe the design and implementation of PrivAnalyzer. Section VII presents our evaluation on security improvements when using Linux privileges and how to refactor applications to improve the efficacy of Linux privileges. Section VIII evaluates PrivAnalyzer's performance. Sections IX and X describe related and future work; Section XI concludes.

II. Linux Privileges

Linux divides the power of the root user into separate privileges (which Linux calls *capabilities*) [2]. Each privilege bypasses a subset of the access control rules which the root user on a traditional Unix system can bypass. For example, the `CAP_CHOWN` privilege allows a process to change the owner of a file to any user even if the process is owned by a non-root user. Likewise, the `CAP_SETUID` privilege allows a process to set its effective, real, and saved user IDs to any value.

Each Linux process has three sets of privileges [6]:

- *Effective*: This is the privilege set that the operating system kernel checks when making access control decisions.
- *Permitted*: This is the set of privileges that the process is allowed to use in its effective set.
- *Inheritable*: This privilege set limits the privileges that a process can acquire when executing a new program.

The Linux kernel provides system calls which allow a process to change its effective and permitted sets [2]. The effective privilege set must always be a subset of the permitted privilege set, thereby making the permitted set the feature that limits the privileges that can be enabled and used within the effective set [6]. A process can only change its permitted set to a subset of its current value i.e., a process can remove privileges from its permitted set but cannot add privileges to its permitted set [6].

Borrowing terminology from the compartmented mode workstation [4], we say a process *raises* a privilege when it turns it on in the effective set and *lowers* a privilege when it disables it in the effective set. The PitBull Foundation system [13] *remove* operation disables a privilege in both the effective and permitted sets; a removed privilege is no longer in the permitted set and can never again be acquired by the process until it executes a new program [6].

We use three wrappers around the Linux system calls that manipulate a process's privileges taken from the AutoPriv compiler project [11]:

- **priv_raise**: Enable one or more privileges in the effective privilege set.
- **priv_lower**: Disable one or more privileges in the effective privilege set.
- **priv_remove**: Disable one or more privileges in both the effective and permitted privilege sets.

III. Attack Model

Our attack model is a modified version of the attack model from the AutoPriv compiler [11]. We assume a strong attacker that can use memory safety attacks against applications. Consequently, our model allows an attacker to corrupt the arguments to system calls and to call system calls in an order not permitted by the original application's control flow [14], [15]. Our attack model also permits an attacker to inject code into an application. However, since there are defenses that can limit the system calls invoked by a process [16], our model assumes that attackers can only use system calls used by the original program [17].

As Hu et al. [11] explain, a consequence of the AutoPriv attack model is that attackers can use system calls to enable any privilege in the effective set that still remains in the process's permitted set [2]. We assume that Linux kernel is part of the trusted computing base (TCB), i.e., once a privilege is removed from the permitted set, it can not be added back. Our work therefore measures the amount of damage that an attacker can perform with the privileges in the permitted set.

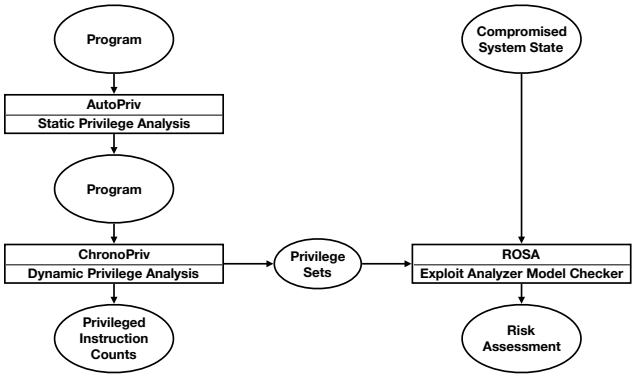

Fig. 1: PrivAnalyzer Architecture

IV. Maude Background

Maude [12] is a term rewriting language for creating formal verification tools. Maude programs specify the format of abstract syntax trees representing *terms* and rules that *rewrite*, or transform, abstract syntax trees from one form to another. Maude supports both equational rewrite rules and term rewrite rules. Equational rewrite rules are used to model deterministic behavior; repeated application of equational rewrite rules must eventually yield a term that has a unique *normal form* that equational rewrite rules cannot rewrite further. Term rewrite rules permit Maude to model non-deterministic computation and do not need to yield a unique normal form.

Maude [12] provides a search command that takes a term representing the initial state of a system and a pattern for what could be the end state of the system and searches for whether a set of rewrite rules will take the system from the initial term to a term matching the search criteria. This feature allows Maude programs to be used as model checkers: if terms model the state of a system and the rewrite rules model transitions between those states, then a user can search the space of terms for states matching a specified criteria.

To ease modeling of concurrent systems, Maude provides syntactic sugar for expressing a term rewriting system as an associative set of objects that can consume messages from an associative set of messages [12]. While primarily designed for modeling concurrent systems, we leverage this syntactic sugar to build our bounded model checker with less Maude code.

V. Design

Figure 1 shows the PrivAnalyzer system. PrivAnalyzer first analyzes the program with AutoPriv [11]. AutoPriv takes a program that uses Linux privileges, uses static analysis to find the points in the program that use privileges, and computes the program points at which each privilege becomes *dead*, i.e., points after which the program will never use the privilege again. AutoPriv computes this information so that it can safely insert code into a program that removes privileges from the permitted set when they are no longer needed, making the privileges inaccessible to attackers.

Once the AutoPriv component has finished, PrivAnalyzer feeds the transformed program from AutoPriv into *ChronoPriv*. ChronoPriv measures the number of instructions a program executes dynamically while a certain privilege set is live i.e., the privileges are in the permitted set. ChronoPriv also records the user and group IDs that the process uses while various privileges are live. ChronoPriv creates a report indicating which privilege sets are live and for how long during a program's execution.

Finally, PrivAnalyzer feeds the privilege sets and user/-group ID credentials observed during execution to the ROSA bounded model checker. As Section V-B describes, ROSA determines if the desired result of an attacker (the *Compromised System State* in Figure 1) can be reached with the program's credentials, privilege sets, and system calls. ROSA outputs a *risk assessment* indicating whether the attacker could abuse the program's privileges to reach the compromised system state should the program have a vulnerability.

By determining *what* damage an attacker can do with each privilege set in the program (determined by ROSA), and by determining *for how long* each privilege set is available during execution (determined by ChronoPriv), PrivAnalyzer can provide a quantitative metric expressing the risk a privileged program poses if it has vulnerabilities. Developers can use PrivAnalyzer's results to evaluate the security impact of changes to a program's privilege use. For example, if a developer modifies a program to add a new feature or to fix a bug, PrivAnalyzer can measure how much the program's security posture has changed due to the modification.

We now describe the design of the ChronoPriv dynamic analysis and the ROSA model checker in more detail.

A. ChronoPriv Vulnerability Analyzer

Developers ask two questions when writing programs that must bypass the access controls: how long does a program use different combinations of privileges, and what damage can be done with those privileges should the program have an exploitable vulnerability? To answer the first query, we designed *ChronoPriv*. As Figure 1 shows, ChronoPriv instruments a program to record and report the number of instructions executed with each combination of privileges present in the program's permitted privilege set and the process's credentials when those privileges are available in the permitted privilege set. On Linux, ChronoPriv records the permitted privilege set, the real, saved, and effective user IDs and group IDs of the process, and the number of instructions executed with that combination of privileges and user and group IDs. When the program terminates, ChronoPriv outputs its results.

When using PrivAnalyzer, developers run their programs with various inputs to measure how many instructions are executed dynamically with each permitted privilege set and user/group credentials. When the developer finishes collecting dynamic privilege and user/group credential information, ChronoPriv feeds its results into the ROSA bounded model checker. For each combination of privilege set and user/group ID credentials, ROSA determines whether attacks of interest to the developer could occur if the program has a vulnera-

bility that the attacker can exploit while those privileges and user/group ID credentials are in effect.

B. ROSA Bounded Model Checker

ROSA is a Maude program built using the Object Maude extension [12] that models a Linux system with processes, users, groups, directories, files, and sockets. Writing ROSA in Maude allows ROSA to be easily enhanced to model new (existing or hypothetical) access controls. ROSA models a Linux system as a set of objects and messages. Processes, files, and sockets are represented as objects while messages sent to each process object represent system calls that the process can execute. ROSA then defines a set of rewrite rules that specifies how the state of objects in the system changes when a process object "receives" a message, i.e., when the process executes a system call. Given a set of objects and messages, ROSA uses Maude term rewriting [12] to determine if a set of objects matching a specified pattern i.e., the description of a compromised state, can be created by applying rewrite rules on the initial set of objects and messages which represent the system's initial state. In other words, ROSA searches for states that are reachable from the initial state of the system that match the description of a compromised system. If ROSA cannot find a reachable state matching the compromised state, ROSA concludes that the program, when running with the specified privileges and user/group IDs, cannot put the system into the comprised system state if it were exploited by an attacker.

Process objects represent one task in Linux; a Linux task is either a process or a thread [2]. Each process object has an effective, real, and saved user ID (UID) and an effective, real, and saved group ID (GID). Processes also have a state attribute indicating whether they are running or have been terminated. Additionally, a process object has two sets of object IDs, named `rdfset` and `wrfset`, respectively, that contain the IDs of objects (files and sockets) that the process has opened for read access and write access, respectively.

File objects use unique integers for their object identifiers. Each file object also contains attributes for the file name, the file permissions, the file owner, and the file group. File names are for human readability; rewrite rules do not use them.

To simulate system calls that modify directory entries, ROSA provides a directory object which is nearly identical to a file object: it has attributes for its owner, group, permission bits, and a human-readable name. Additionally, it contains an inode attribute which indicates the object ID of the file object to which the directory entry refers. In this way, ROSA can model system calls such as `unlink()` and `rename()` as messages that, when consumed by a process, modify directory entry objects. ROSA also models basic pathname lookup e.g., checking for search permission on a file's parent directory, but (without loss of generality) only on a single parent directory.

Sockets are objects that have a unique object identifier and an integer port attribute. Socket objects represent TCP sockets that can be bound to TCP ports.

System call messages specify the system call name, the process which is allowed to execute the system call, the arguments to the system call, and the set of privileges that the system call can use. Certain arguments (such as file object identifiers) can be wildcard values, allowing ROSA to try different values for the argument taken from the set of objects e.g., files, given in its input. This feature allows ROSA to model both attacks that modify system call inputs and attacks that do not. Making privileges an attribute of a system call (instead of an attribute of the process object) allows ROSA to model attacks which only use specific privileges with specific system calls as well as attacks that utilize any of a process's privileges with any system call.

ROSA also provides user objects and group objects which contain a user ID or group ID, respectively. These objects allow a PrivAnalyzer user to denote which users and groups can be used to replace wildcard values in system calls that take user ID and group ID arguments e.g., `setresuid()`. Restricting UID and GID values to only those specified in user and group objects allows PrivAnalyzer users to constrain ROSA's search space.

To use ROSA, a user provides a set of processes, a set of user and group objects representing the users and groups relevant to the attack being analyzed, and a set of objects upon which the system calls may operate; PrivAnalyzer tailors the input with the set of system calls and relevant privileges reported by ChronoPriv. PrivAnalyzer can then query ROSA to see if a compromised state can be reached from the initial state of objects and messages.

As an example, suppose that there is a program that can execute the following system calls in any order provided that each system call is only executed once:

1) `open()` for read-only access using no privilege
2) `setuid()` with `CAP_SET_UID` privilege
3) `chown()` with `CAP_CHOWN` privilege; the user ID value is unconstrained, but the group ID value must be 41.
4) `chmod()` with no privilege

The user wants to know if the program can open the file `/etc/passwd`, which has owner 40 and group 41, for reading. The process currently has effective, real, and saved user and group IDs that do not match the file's owner and group.

Figure 2 shows the initial state for this query. Note that the arguments to `chmod()` turn on all permissions on a file (an attacker would want to make a file as accessible as possible, and the arguments to `chmod()` do not affect which privileges are needed for `chmod()` to succeed). Note also that various arguments to system calls e.g., the file identifier and owner arguments to `chown()`, are specified as -1 which tells ROSA that they are wildcard arguments.

Next, the user specifies a pattern representing the state for which she is looking. This is any state where the file object (Object ID 3) is in the read set of the process. Figure 3 shows that we are looking for a term (denoted by `Z:Configuration`) that represents the state for which to search. As we do not care if the attacker changes the user and group IDs of the process, they are left as variables (A

```
(search in UNIX :
  < 1 : Process | euid : 10 , ruid : 11 , suid : 12 ,
                  egid : 10 , rgid : 11 , sgid : 12 ,
                  state : run ,
                  rdfset : empty , wrfset : empty >
  < 2 : Dir | name : "/etc" ,
              perms : r w x r w x r w x ,
              inode : 3 , owner : 40 , group : 41 >
  < 3 : File | name : "/etc/passwd" ,
               perms : - - - - - - - - - ,
               owner : 40 , group : 41 >
  < 4 : User | uid : 10 >
  open(1,3,r - -,empty)
  setuid(1,-1,CapSetuid)
  chown(1,-1,-1,41,CapChown)
  chmod(1,-1,r w x r w x r w x,empty)
```

Fig. 2: ROSA Start State Example

through F) with no constraints. Putting it all together, we get the Maude `search` command in Figure 4.

For this example, ROSA finds the following solution, indicating that the process can put the system into the compromised state:

- The process successfully uses `chown()` to change the file's owner to match the effective user ID of the process.
- The process then uses `chmod()` to change the permission bits of the file to make it readable by the owner.
- The process opens the file.

A limitation of ROSA is that the user must specify the number of times that an attacker can use a given system call. We find this limitation acceptable. Many attacks do not use a particular system call many times; they merely use a few system calls with the necessary privileges [17], [18], [15].

VI. Implementation

We implemented ChronoPriv as an LLVM [19] 3.7.1 Intermediate Representation (IR) pass which adds code to each basic block to record the number of IR instructions executed dynamically within the block, the permitted privilege sets available when each basic block executes, and the user/group ID credentials. ChronoPriv omits `unreachable` instructions in its instruction counts as executing an `unreachable` instruction terminates the program [20].

We implemented ROSA using 1,151 lines of Maude code with the Full-Maude system [12] on Maude 2.7. ROSA currently models simple processes and threads, files, a file system with a single level of directories, and TCP sockets. We also built a simple test suite for ROSA that verifies that a subset of

```
=>* Z:Configuration
< 1 : Process | euid : A:Int , ruid : B:Int ,
                suid : C:Int ,
                egid : D:Int , rgid : E:Int ,
                sgid : F:Int , state : G:procState ,
                rdfset : H:Set{Int} ,
                wrfset : I:Set{Int} >
such that (3 in G:Set{Int}) .)
```

Fig. 3: ROSA End State Example

```
(search in UNIX :
  < 1 : Process | euid : 10 , ruid : 11 , suid : 12 ,
                  egid : 10 , rgid : 11 , sgid : 12 ,
                  state : run ,
                  rdfset : empty , wrfset : empty >
  < 2 : Dir | name : "/etc" ,
              perms : r w x r w x r w x ,
              inode : 3 , owner : 40 , group : 41 >
  < 3 : File | name : "/etc/passwd" ,
               perms : - - - - - - - - - ,
               owner : 40 , group : 41 >
  < 4 : User | uid : 10 >
  open(1,3,r - -,empty)
  setuid(1,-1,CapSetuid)
  chown(1,-1,-1,41,CapChown)
  chmod(1,-1,r w x r w x r w x,empty)
  =>* Z:Configuration
  < 1 : Process | euid : A:Int , ruid : B:Int ,
                  suid : C:Int ,
                  egid : D:Int , rgid : E:Int ,
                  sgid : F:Int , state : G:procState ,
                  rdfset : H:Set{Int} ,
                  wrfset : I:Set{Int} >
  such that (3 in G:Set{Int}) .)
```

Fig. 4: ROSA Query

the system calls that it supports exhibit the expected behavior for privileged and unprivileged operation.

ROSA supports system calls for processes (`setuid`, `seteuid`, `setresuid`, `setgid`, `setegid`, `setresgid`, `kill`), files and directories (`open`, `chmod`, `fchmod`, `chown`, `fchown`, `unlink`, `rename`), and TCP sockets (`socket`, `bind`, `connect`). ROSA models operations that modify the effective, real, and saved user ID and group ID values of a process as these are used by the Linux access controls [2]. It does not model the file system ID feature of Linux [2] as that feature is seldom used and not applicable to other Unix-like systems. ROSA only supports TCP sockets and only a subset of socket operations; adding support for other socket features is straightforward. ROSA also lacks support for system calls that create new threads and processes and the `exec()` family of system calls; the attacks that we model do not rely on using these system calls.

ROSA does not yet model Linux namespaces or system calls that modify the file system namespace e.g., `chroot()`, `mount()`, and `clone()`. While it supports system calls that remove links to files e.g., `unlink()` and `rename()`, it does not support system calls, such as `creat()` and `link()`, that create new files and new links to existing files. It also lacks support for newer system calls such as `openat()`. Even so, ROSA can model powerful attacks that steal and corrupt sensitive data stored in files, masquerade as critical services, and disrupt availability of critical system services.

VII. Security Evaluation

We now use PrivAnalyzer to evaluate the security of Linux applications that execute as root. We then refactor these applications to improve their security posture and use PrivAnalyzer to measure the improvement.

A. Modeled Attacks

As Table I states, we model the following attacks:

TABLE I: Modeled Attacks

Attack	Description
❶	Read from `/dev/mem` to steal application data
❷	Write to `/dev/mem` to corrupt application data
❸	Bind to a privileged port to masquerade as a server
❹	Send a SIGKILL signal to kill the `sshd` server

TABLE II: Programs for Experiments

Program	Version	SLOC	Description
thttpd	2.26	8,922	Small single-process web server
passwd	4.1.5.1	50,590	Utility to change user passwords
su	4.1.5.1	50,590	Utility to log in as another user
ping	s20121221	12,202	Test reachability of remote hosts
sshd	6.6p1	83,126	Login server with encrypted sessions

1) **Reading** `/dev/mem`: Opening the `/dev/mem` device file for reading allows a process to read any memory location on the system [21], allowing the process to read any data stored within any process on the system.
2) **Writing** `/dev/mem`: Opening `/dev/mem` for writing allows a compromised process to alter the data within any process.
3) **Binding to a privileged port:** Binding a socket to a privileged TCP port allows a compromised process to masquerade as a trusted server (e.g., the remote login server).
4) **Sending** `SIGKILL` **to a critical server:** Sending signals to a system-owned server permits an attack to disrupt the availability of system services. We model the sending of a signal to a server owned by another user.

As Section III describes, our attack model assumes that an exploited vulnerability, if it exists within the program, permits an attacker to enable any privilege that is in the process's permitted privilege set and to use that privilege with any system call used by the application. This attack model permits sophisticated attacks such as code-reuse attacks that misuse indirect control transfer instructions to redirect execution to any of the system calls available to the program [18], [22], [23]. To model such attacks with ROSA, we identified the system calls used by the application and created, for each attack, an input file to ROSA that contains the processes and files needed to perform the attack and the list of system calls that the attack can utilize.

As the programs in our evaluation reduce their maximum privilege sets during execution, the possible privileges that an attack can use changes over time. We used ChronoPriv to record a process's real, effective, and saved user ID and group ID along with privileges sets as the Linux access controls use the effective privileges, user IDs, and group IDs to decide if an operation, such as opening a file, can succeed [2]. We therefore created, for each possible combination of privilege sets and IDs, an input file that permits any system call to use the entire maximum privilege set and asked ROSA if, with those privileges and IDs, an attack was possible.

B. Test Programs

We use the test programs from Hu et al.'s AutoPriv project [11]. These programs, described in Table II, typically run as the root user and are examples of different types of privileged programs (some are network servers; others are setuid root utilities). Hu et al. [11] modified these programs to add calls to `priv_raise` and `priv_lower` around system calls or library function calls that need privileges. We installed the programs so that they start up with the correct permitted set instead of starting up as a setuid root executable.

We used `sloccount` [24] to count the lines of C, C++ and assembly code (excluding comments) in each test program. For `passwd` and `su`, we counted the lines of code in the entire shadow utility suite. For `sshd`, we counted the lines of code in the whole OpenSSH suite. We ran all experiments on a 64-bit Ubuntu 16.04 system.

We compiled the test programs with PrivAnalyzer with ChronoPriv enabled. The compiler inserts calls to `priv_remove()` to remove dead privileges, inserts a `prctl()` call [6] into the program to disable kernel backward compatibility features (such as enabling privileges in the effective set when the process's effective user ID is zero), and adds our dynamic instruction counting instrumentation into the program. We ran each program as follows: for `ping`, we configured it to send requests 10 times to the localhost network interface using the `-c 10` flag. For `passwd`, we ran it to change the current user's password. For `su`, we ran it to execute the `ls` program as another user. For `thttpd`, we used ApacheBench [25] with concurrency level 1 and request number 1 to fetch one 1 MB file. For `sshd`, we started it in the foreground with the `-d` flag and ran `scp` to fetch one 1 MB file stored in another user's account.

The *Privileges* column of Table III describes all the privilege set combinations observed by ChronoPriv's dynamic analysis when we executed each program, starting with the full privilege set available to the program when it starts execution and ending with the smallest privilege set upon program exit. Table III also presents the real, effective, and saved UIDs and GIDs observed with each privilege set in columns *UID* and *GID* respectively. UID 1000 corresponds to the user that starts the execution of the program. There is another regular user in the system with UID 1001. For `su`, this UID's username is the username argument to the program; for `sshd`, user 1000 starts the program and runs `scp` to transfer files from user 1001. The second column of Table III provides a short name for the combination of privileges and effective, real, and saved UID/GID values. For every attack, we use ROSA to analyze the attack under each combination of privileges and process credentials listed in Table III.

C. Efficacy Evaluation

Table III summarizes the results of our analysis. For each program, the *Vulnerability* column of Table III shows whether the attacks we modeled (summarized in Table I) were successful. To evaluate how long each program spends potentially vulnerable to a particular privilege escalation attack, we ran

TABLE III: Security Efficacy Results. A ✔ denotes vulnerability to an attack and ✘ denotes invulnerability to an attack. Column *Name* shows a short name for each combination of privileges and process credentials.

Program	Name	Privileges	UID ruid, euid, suid	GID rgid, egid, sgid	Dynamic Instruction Count	Vulnerability 1	2	3	4
passwd	passwd_priv1	CapDacReadSearch,CapDacOverride, CapSetuid,CapChown,CapFowner	1000,1000,1000	1000,1000,1000	2,654 (3.81%)	✔	✔	✘	✔
	passwd_priv2	CapSetuid,CapDacOverride,CapChown, CapFowner	0,0,0	1000,1000,1000	43 (0.06%)	✔	✔	✘	✔
	passwd_priv3	CapSetuid,CapDacOverride,CapChown, CapFowner	1000,1000,1000	1000,1000,1000	41,255 (59.15%)	✔	✔	✘	✔
	passwd_priv4	CapChown,CapFowner,CapDacOverride	0,0,0	1000,1000,1000	25,630 (36.75%)	✔	✔	✘	✘
	passwd_priv5	(empty)	0,0,0	1000,1000,1000	162 (0.23%)	✘	✘	✘	✘
ping	ping_priv1	CapNetRaw,CapNetAdmin	1000,1000,1000	1000,1000,1000	194 (1.36%)	✘	✘	✘	✘
	ping_priv2	CapNetAdmin	1000,1000,1000	1000,1000,1000	204 (1.43%)	✘	✘	✘	✘
	ping_priv3	(empty)	1000,1000,1000	1000,1000,1000	13,844 (97.21%)	✘	✘	✘	✘
sshd	sshd_priv1	CapChown,CapDacOverride, CapDacReadSearch,CapKill,CapSetgid, CapSetuid,CapNetBindService, CapSysChroot	1000,1000,1000	1000,1000,1000	196,181 (0.31%)	✔	✔	✔	✔
	sshd_priv2	CapChown,CapDacOverride, CapDacReadSearch,CapKill, CapSetgid,CapSetuid,CapSysChroot	1000,1000,1000	1000,1000,1000	62,374,249 (98.94%)	✔	✔	✘	✔
	sshd_priv3	CapChown,CapDacOverride, CapDacReadSearch,CapKill, CapSetgid,CapSetuid,CapSysChroot	1001,1001,1001	1001,1001,1001	468,197 (0.74%)	✔	✔	✘	✔
	sshd_priv4	CapChown,CapDacOverride, CapDacReadSearch,CapKill, CapSetgid,CapSetuid,CapSysChroot	1000,1000,1000	1001,1001,1001	1,738 (0.00%)	✔	✔	✘	✔
su	su_priv1	CapDacReadSearch,CapSetgid, CapSetuid	1000,1000,1000	1000,1000,1000	38,880 (82.10%)	✔	✔	✘	✔
	su_priv2	CapSetgid,CapSetuid	1000,1000,1000	1000,1000,1000	2,449 (5.17%)	✔	✔	✘	✔
	su_priv3	CapSetgid,CapSetuid	1000,1000,1000	1001,1001,1001	133 (0.28%)	✔	✔	✘	✔
	su_priv4	CapSetuid	1000,1000,1000	1001,1001,1001	82 (0.17%)	✔	✔	✘	✔
	su_priv5	CapSetuid	1001,1001,1001	1001,1001,1001	43 (0.09%)	✔	✔	✘	✔
	su_priv6	(empty)	1001,1001,1001	1001,1001,1001	5,768 (12.18%)	✘	✘	✘	✘
thttpd	thttpd_priv1	CapChown,CapSetgid,CapSetuid, CapNetBindService,CapSysChroot	1000,1000,1000	1000,1000,1000	323 (0.00%)	✔	✔	✔	✔
	thttpd_priv2	CapSetgid,CapNetBindService, CapSysChroot	1000,1000,1000	1000,1000,1000	4,685,943 (9.82%)	✔	✘	✔	✘
	thttpd_priv3	CapSetgid,CapNetBindService	1000,1000,1000	1000,1000,1000	361 (0.00%)	✔	✘	✔	✘
	thttpd_priv4	CapSetgid	1000,1000,1000	1000,1000,1000	7,199 (0.02%)	✔	✘	✘	✘
	thttpd_priv5	(empty)	1000,1000,1000	1000,1000,1000	43,008,606 (90.16%)	✘	✘	✘	✘

each instrumented program created by ChronoPriv with the sample inputs described in Section VII-B and recorded the number of LLVM instructions executed with each privilege set and IDs. Table III records the results in the *Dynamic Instruction Count* column. The results in Table III show that solely reducing the available Linux privileges helps reduce vulnerability to attacks which bind to a privileged port (Attack 3). However, it often fails to mitigate the other attacks.

`ping` is not vulnerable to any attack we modeled for all its executed instructions. It needs `CAP_NET_RAW` to call `socket` with `SOCK_RAW` to create a raw socket. It does this only once at the very beginning of the program, allowing `ping` to drop `CAP_NET_RAW` early in its execution. It also needs `CAP_NET_ADMIN` to use the `SO_DEBUG` and `SO_MARK` options in `setsockopt` in case the `-d` or `-m` flags are specified on the command line. This is done in a setup function also executed early during program execution. Therefore, `ping` can drop all its privileges very early.

Similar to `ping`, `thttpd` also uses privileges early in its execution (e.g., to bind to a privileged port and to set the server's root directory). After all the configuration work is done, `thttpd` drops all its privileges.

`sshd` is vulnerable to attacks 1, 2, and 4 for its entire execution. It drops `CAP_NET_BIND_SERVICE` after binding to a privileged port but retains all its other privileges. The problem is twofold. First, some of `sshd`'s signal handlers use privileges. As signal handlers can be called at any time, any privileges they use remain live during execution [11]. Second, we believe that implementation limitations within the AutoPriv compiler are also responsible. AutoPriv [11] uses a conservatively correct call graph when propagating information about privilege use inter-procedurally. When `sshd` creates a child process to handle a client connection, the child process enters a loop that continually reads and processes data from the client. The privileges remain live during this loop. We have found an indirect function call within this loop. Since AutoPriv creates an over-approximation of the targets of the indirect function call [11], it probably thinks that all the functions which raise privileges are targets of this indirect call and, consequently, keeps these privileges alive during the loop. The privileges are dead after the loop, but `sshd` doesn't exit the loop until the client connection closes. A more accurate call graph analysis may improve AutoPriv's ability to identify when privileges can be safely removed using `priv_remove()`.

TABLE IV: Lines of Code Changed for Refactored Programs

	shadow library code	passwd.c	su.c
Added	7	23	35
Deleted	76	13	6

The `passwd` program is vulnerable to attacks on `/dev/mem` and our denial of service attack (attacks 1, 2, and 4) for 63% of its execution. `passwd` needs `CAP_DAC_READ_SEARCH` to retrieve the user's password entry from the `/etc/shadow` password database using `getspnam()`. It also uses `CAP_SETUID` to call `setuid(0)` to set its *real* and *saved* user ID to root to ignore unexpected signals (Linux requires a process's effective or real UID to match either the real or saved UID of a target process when sending a signal [2]). The `passwd` program then needs `CAP_DAC_OVERRIDE` so that it can replace the old shadow database with a new one and to lock a lock file, preventing concurrent executions of the `passwd` program from interfering with each other. It needs this privilege because, as written, the `passwd` program makes minimal assumptions about which user owns the `/etc` directory and the `/etc/shadow` file (it explicitly uses `stat()` to find the owner of `/etc/shadow` and then uses `chown()` to ensure the new `/etc/shadow` file it creates is owned by the same user). CAP_SETUID is kept for 63% of `passwd`'s execution and CAP_DAC_OVERRIDE is used near the end of the program, so the program is vulnerable to attacks 1, 2, and 4 for over half its execution.

Similar to `passwd`, `su` also needs `CAP_DAC_READ_SEARCH` to call `getspnam()` to read passwords from the `/etc/shadow` shadow password database. If the operating system has a `sulog` file, `su` then needs `CAP_SETGID` to change the effective group ID to the group ID of `sulog` so that `su` can write to the `sulog` file. The `su` program then needs `CAP_SETUID` and `CAP_SETGID` to change the current process's user IDs, groups IDs, and supplementary group list to the IDs of the target user to which it is switching. These two operations occur very late in execution, and this is why `su` is vulnerable to attacks 1, 2, and 4 for 88% of its execution.

D. Security Refactoring Process

The results in Table III show that simply dropping Linux privileges when no longer needed may not suffice. While `ping` is invulnerable to all the attacks we modeled and `thttpd` is invulnerable to all attacks for more than 90% of its execution, `passwd`, `su` and `sshd` remain vulnerable to most of the attacks for most of their execution time. PrivAnalyzer reveals the similarity among these programs; they retain powerful privileges until late in their execution. This observation led us to investigate whether we could improve their resistance to privilege escalation by refactoring their code. We chose two of the programs, `passwd` and `su`, and we re-evaluated their privilege use and vulnerability under our attack model with our tool. We chose these two programs because they are relatively small but still use powerful Linux privileges, such as `CAP_SETUID` and `CAP_CHOWN`. We describe our results for each program below. We refactored the versions of the two programs that use `priv_raise` and `priv_lower` (not the original versions that run as root). Our refactoring requires very minor source code changes. Table IV shows the amount of source code we changed.

1) Refactored passwd: As Table III shows, `CAP_SETUID` is available for 63% of `passwd`'s execution, and `CAP_OWNER`, `CAP_FOWNER`, and `CAP_DAC_OVERRIDE` are available for more than 99% of executed instructions. These four privileges are extremely powerful. With `CAP_SETUID`, a process can change its *effective* user ID to match the owner of any file. It can then change the file's permission bits and, subsequently, open the file for reading and writing without using any other privileges [2]. `CAP_SETUID` also allows a process to change its *real* or *effective* user ID to match the real or saved user ID of a victim process, allowing it to send a `SIG_KILL` signal to the victim [2]. With `CAP_OWNER`, a process can change a file's owner to be any user. With `CAP_FOWNER`, a process can change the permission bits of any file. With `CAP_DAC_OVERRIDE`, a process can gain read, write, and execute access to any file. We aimed to reduced the number of instructions executed with these four privileges.

We devised two changes to `passwd` to permit it to remove privileges earlier in execution. First, we noticed that, when using privileges, `passwd` can call `setuid()` much earlier in its execution (namely, after it has determined the real UID of the user that executed it). Moving the `setuid()` call to an earlier point allows the process to drop `CAP_SETUID` earlier.

Second, allowing `passwd` to execute with an effective UID of zero still allows it to open `/dev/mem` as the root user owns many system files, including device files and the shadow database files, on Ubuntu. This allows `passwd` to read and write `/dev/mem` even though we have reduced its privilege use. However, there is no reason for root to own the shadow database. As Section VII-C states, the shadow suite source code does not assume that root owns the password database. Since these shadow-related files are located in the `/etc` directory, we can create a new special user named *etc* (UID number 998 in our case) and set the owner of the `/etc` directory and the shadow password file to be *etc*. By doing so, `passwd` can change its effective UID to *etc* and effective GID to *shadow* (the group owner of the shadow file on Ubuntu 16.04) to eliminate the use of `CAP_OWNER`, `CAP_FOWNER`, and `CAP_DAC_OVERRIDE`, which are responsible for updating the password database. To ignore unexpected signals, `passwd` can set its *real* and *saved* user IDs to *etc* as well.

We ran PrivAnalyzer on the refactored `passwd` program with these changes. Our results in Table V show that `passwd` is invulnerable to all of our modeled attacks for 96% of its execution. Going back to the results in Table III, we see that using `passwd_priv4` instead of `passwd_priv3` decreases the vulnerability of `passwd`. In particular, PrivAnalyzer determines that dropping `CAP_SETUID`, i.e. the only privilege that makes the two sets differ as shown in Table III, makes one of the four attacks infeasible. We believe that highlighting these changes in privilege sets would help

TABLE V: Results for Refactored Programs. ✔ denotes vulnerability to an attack, ✘ denotes invulnerability to the attack, and 🕒 denotes that ROSA timed out for the attack/privilege set combination.

Program	Name	Privileges	UID ruid, euid, suid	GID rgid, egid, sgid	Dynamic Instruction Count	Vulnerability 1	 2	 3	 4
passwd	passwdRef_priv1	CapSetuid,CapSetgid	1000,1000,1000	1000,1000,1000	2,633 (3.82%)	✔	✔	✘	✔
	passwdRef_priv2	CapSetuid,CapSetgid	998,998,1000	1000,1000,1000	42 (0.06)	✔	✔	✘	✔
	passwdRef_priv3	CapSetgid	998,998,1000	1000,1000,1000	49 (0.07%)	✔	✘	✘	✘
	passwdRef_priv4	CapSetgid	998,998,1000	1000,42,1000	42 (0.06%)	✔	🕒	✘	✘
	passwdRef_priv5	empty	998,998,1000	1000,42,1000	66,165 (95.99%)	✘	✘	✘	✘
su	suRef_priv1	CapSetuid,CapSetgid	1000,1000,1000	1000,1000,1000	264 (0.56%)	✔	✔	✘	✔
	suRef_priv2	CapSetuid,CapSetgid	1000,998,1001	1000,1000,1000	42 (0.09%)	✔	✔	✘	✔
	suRef_priv3	CapSetgid	1000,998,1001	1000,1000,1000	42 (0.09%)	✔	🕒	✘	✘
	suRef_priv4	CapSetgid	1000,998,1001	1000,998,1001	126 (0.27%)	✔	🕒	✘	✘
	suRef_priv5	empty	1001,1001,1001	1001,1001,1001	5,766 (12.21%)	✘	✘	✘	✘
	suRef_priv6	empty	1000,998,1001	1000 998,1001	40,951 (86.69%)	🕒	🕒	✘	✘
	suRef_priv7	empty	1000,998,1001	1001,1001,1001	43 (0.09%)	🕒	🕒	✘	✘

developers identify powerful privileges and help guide them in refactoring their programs to reduce privilege use.

2) Refactored su: `su` is vulnerable also because `CAP_SETUID` is live for too long. We observed that the process determines the target user early during execution. Therefore, we can modify `su` to change the supplementary group ID list much earlier and to use `CAP_SETUID` and `CAP_SETGID` to set the *saved* user ID and *saved* group ID to the target user ID and group ID, respectively. When `su` needs to switch user IDs and group IDs, it can call `setresuid()` and `setresgid()` to change the effective user ID and group ID to the saved user ID and group ID *without* using privileges [2]. For the `sulog` file, we can change its owner to `etc` and set the effective group ID to `etc` when `CAP_SETGID` is available. In this way, `su` can drop these two privileges much earlier. We can also eliminate `CAP_DAC_READ_SEARCH` by setting the effective user ID to the owner of `/etc/shadow` when `CAP_SETUID` is available. The change of the effective user ID and saved user ID doesn't affect the flow of the program because all the identification work is done by checking the *real* user ID, which remains unchanged.

We used PrivAnalyzer to measure the refactored `su`'s security improvement. The results in Table V show that this new `su` cannot launch our four modeled attacks for at least 12% of its execution. Furthermore, since ROSA is unable to deliver a verdict on attacks 1 and 2 for privilege sets 6 and 7 within our 5 hour limit, we assume that our refactored `su` is invulnerable to all the attacks for an additional 87% of its execution time. This is because, as Section VIII discusses, ROSA's analysis often takes longer when attacks are impossible as ROSA must search the entire state space. We believe our refactored `su` is invulnerable for nearly 99% of its execution.

Looking at Table III, we see that `su` is vulnerable with all the privilege sets except the empty one. The last privilege to remain live is `CAP_SETUID` (Table III). Similar to `passwd`, the PrivAnalyzer results help identify which privilege increases the exposure to privilege escalation, helping guide the developer on where to focus refactoring efforts.

E. Security Refactoring Lessons

Our refactoring work provides two key insights into writing privileged applications on Linux:

*a) **Change Credentials Early**:* We observed that many privileged operations in privileged programs simply require that a process running as one user create or manipulate resources e.g., processes and files, owned by *one* other user. However, Linux privileges such as `CAP_DAC_OVERRIDE` allow a process to manipulate files owned by *any* user.

A better approach is to use `CAP_SETUID` and `CAP_SETGID` to provide the process with two sets of credentials: one in the real UID and GID and another in the saved UID and GID.

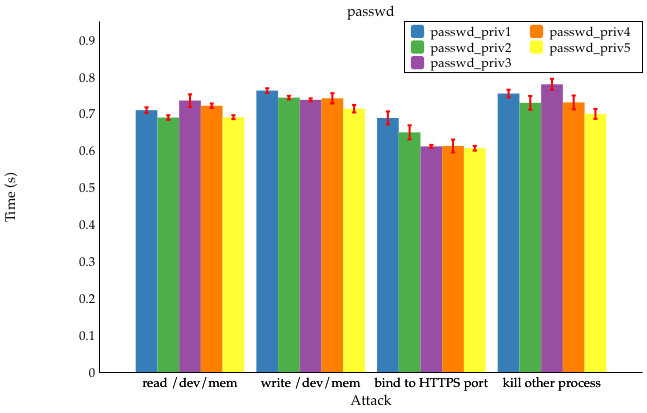

Fig. 5: Search time for `passwd`.

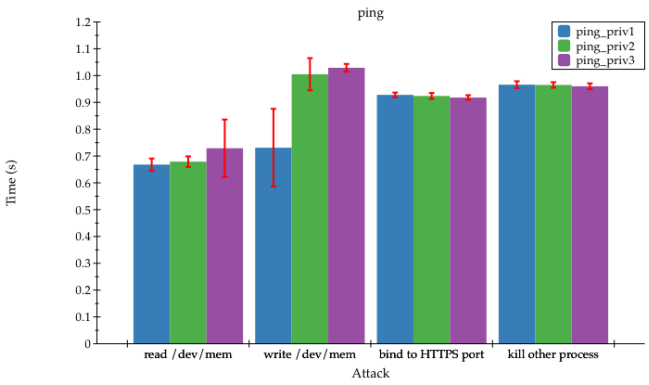

Fig. 6: Search time for `ping`.

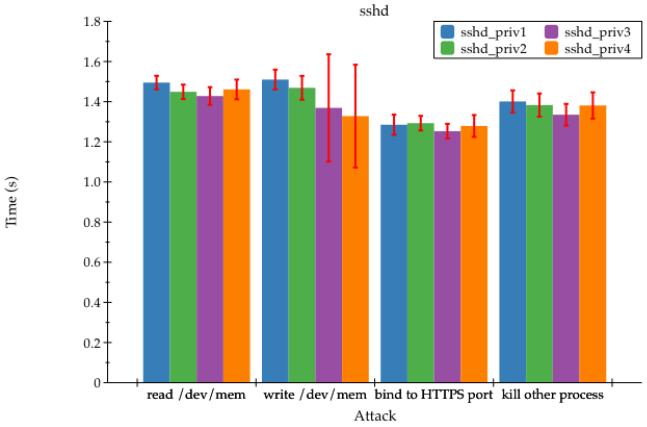

Fig. 7: Search time for `sshd`.

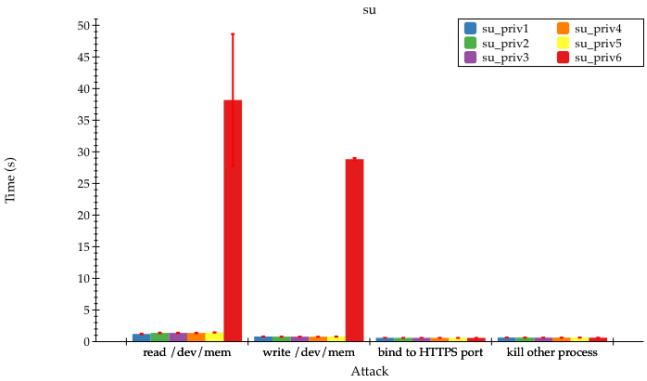

Fig. 8: Search time for `su`.

The process can then switch the effective UID/GID between the real UID/GID and the saved UID/GID *without* privilege. This allows privileges such as `CAP_SETUID` and `CAP_SETGID` to be removed early in execution and often allows privileges such as `CAP_CHOWN`, `CAP_DAC_READ_SEARCH`, and `CAP_DAC_OVERRIDE` to be eliminated entirely.

b) ***Create Special Users for Special Files****:* On Ubuntu, root owns many of the system configuration files. Consequently, running as one user provides access to nearly all sensitive system files. For example, a password-changing program shouldn't be able to read and write device files, but it can on Ubuntu. Having different special users own different files allows privileged programs to configure themselves to only access the files that they need (as we did by changing `/etc/shadow` to be owned by a special `etc` user).

VIII. Performance Evaluation

We now evaluate the performance of ROSA, PrivAnalyzer's bounded model checker, using the programs in Table II. We perform our experiments on a Dell Precision 3620 workstation with a 3.6 GHz Intel® i7-7770 processor, 16 GB of RAM, and a 1 TB TOSHIBA DT01ACA1 hard disk running Ubuntu 16.04. We use the `time` system call to report the sum of user and system time in seconds that ROSA takes to reach a verdict.

We run each test 10 times to compute the average search time and the standard deviation. Figures 5, 6, 7, 8, and 9 show that, in most of the experiments, ROSA needs less than 2 seconds to reach a verdict. However, in one case (the one in which `su` has dropped all of its privileges), ROSA takes almost 40 seconds to decide that `su` cannot read or write `/dev/mem`. We believe this is due to the search space: for attacks like those on `/dev/mem`, numerous system calls such as `open()`, `setresuid()`, `chown()`, and `chmod()` are relevant. For attacks that fail, ROSA must determine that all combinations of executing these system calls does not lead to compromise. In contrast, fewer system calls are relevant to attacks 3 and 4 which kill off other processes or bind to privileged ports, providing ROSA a smaller state space to search.

Figures 10 and 11 show ROSA's performance when analyzing the refactored `passwd` and `su` programs. In general, ROSA takes longer to reach a verdict for the refactored programs. For example, ROSA takes almost 12 hours to determine that the refactored `passwd` cannot write to `/dev/mem` when it executes with privilege set 3 (as Table V shows). There are cases in which the operating system kills ROSA due to high memory consumption (3 days of execution), and we do not get a response. We believe this behavior is due to the large state space to explore. Attacks 1 and 2 involve a larger

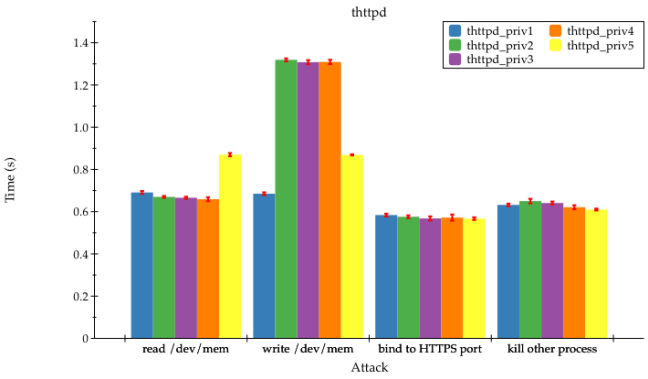

Fig. 9: Search time for `thttpd`.

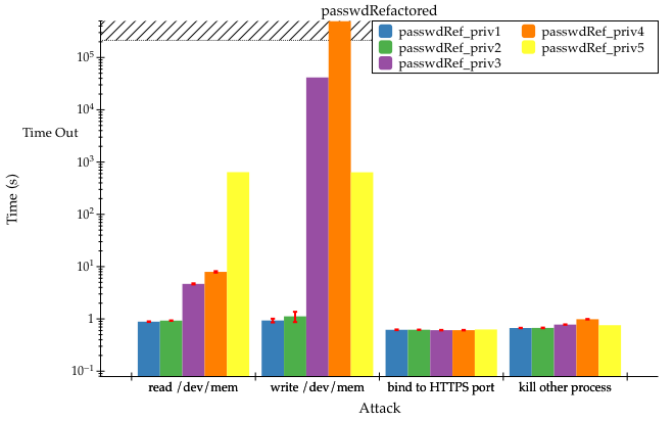

Fig. 10: Search time for refactored `passwd`.

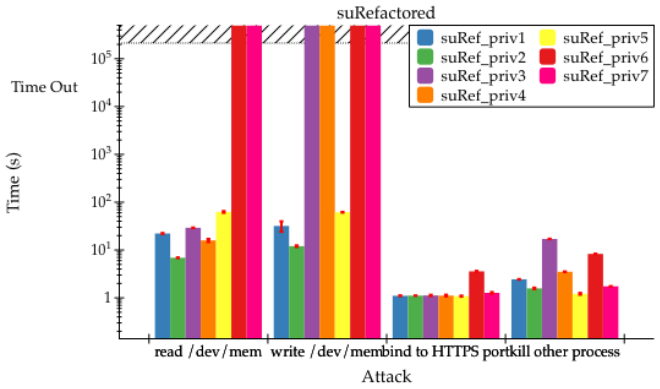

Fig. 11: Search time for refactored `su`.

number of system calls and UID/GID values. ROSA must try every combination of UID/GID pairs in each system call to determine that no combination of process UID/GID settings and file ownership settings will permit the attack to succeed. Given how quickly ROSA can find solutions when attacks are possible, we believe that the attacks are likely to fail when ROSA cannot return an answer in reasonable time.

IX. Related Work

Zanin et al. [9] propose an automatic tool for evaluating security policy configurations for SELinux. SELinux [26] is a system that allows an administrator to specify a set of rules which the Linux kernel enforces when making access control decisions. These rules enable the generation of security policies which are flexible yet difficult for administrators to reason about. To evaluate the efficacy of an SELinux security policy configuration, Zanin et al. [9] formalize the semantics of the configuration constructs and expose the relationships that occur among the configuration rules. Based on the proposed formal model entitled `SELAC` [9], they develop a tool that decides whether access of a specific object by a given subject under the configured policy is possible. Zanin et al. propose a tool [9] that, similar to PrivAnalyzer, can reach verdicts about the possibility of attacks allowed under a specific set of configuration rules. However, unlike our work, they do not quantify the vulnerability window of the system under a specific configuration. In this work, we show that PrivAnalyzer can evaluate hypothetical attacks at different points during the dynamic execution of a program under different sets of privileges. This evaluation led us to refactor a subset of these programs and show that we can improve their security.

Chen et al. develop `VulSAN` [10], a tool that quantifies the quality of protection offered by mandatory access control systems. Given an attack objective and the attacker's initial resources, `VulSAN` identifies minimal sets of programs that, if compromised, can lead to the attack succeeding. VulSAN maps the security policies and the state of the system to Prolog facts. `VulSAN` then generates a graph for each attack scenario that is modeled. Each path in the graph is a sequence of different program executions. `VulSAN` uses the graph to extract the minimal sets of programs that can lead to the modeled attack. Although `VulSAN` can identify chains of compromises between programs in a system, it cannot identify the damage that a single program could cause on its own if exploited. PrivAnalyzer analyzes the behavior of individual programs to determine whether their use of privileges mitigates the risks of privilege escalation attacks.

Vijayakumar et al. [27] evaluate the attack surface of a program, i.e. the program entry points that are accessible to an adversary. They use the operating system's Mandatory Access Control (MAC) policy to automatically separate program data into two sets: 1) trusted, and 2) adversary controlled data. Using runtime analysis, they collect the set of entry points that access objects outside of the constructed trusted set. The proposed method measures the attackability of a program, i.e. how likely it is that a program gets attacked in any way. PrivAnalyzer measures the vulnerability window of a program to an attack, i.e. how long the program is vulnerable to a specified attack. While both systems evaluate important security metrics, the former relies on a MAC scheme and cannot model dynamic changes of access rights. PrivAnalyzer evaluates the vulnerability of a program in a discretionary access control system, accounting for dynamic changes in the access control policy.

X. Future Work

Several interesting directions exist for future work. First, we plan to enhance PrivAnalyzer to model additional operating system privilege models, allowing us to compare their efficacy. For example, PrivAnalyzer could model Solaris privileges [28] and Capsicum [5] and investigate whether they can provide greater protection than Linux privileges.

Second, we will model additional attacks and defenses. For example, our current evaluation assumes that attacks can corrupt application control-flow and data-flow. With enhancements to PrivAnalyzer, we can model attacks that are weakened due to defenses such as control-flow integrity [29] and code-pointer integrity [30] and determine what types of attacks such a weakened attacker can perform.

XI. Conclusions

This paper presented PrivAnalyzer, an automated tool that measures the risk that privileged programs can pose. PrivAnalyzer adds two new components to the AutoPriv compiler [11]: the ChronoPriv vulnerability analyzer and the ROSA bounded model checker. Using PrivAnalyzer, we measured the efficacy of using Linux privileges and showed that enabling, disabling, and removing privileges from otherwise unmodified applications does not significantly improve their security posture. We then refactored two privileged Linux programs to better leverage Linux privileges and used PrivAnalyzer to measure the security improvement. From this exercise, we learned two key approaches for using Linux privileges more effectively.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. This work was funded by NSF award CNS-1463870.

References

[1] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Redmond, WA, USA: Microsoft Press, 2004.

[2] D. P. Bovet and M. Cesati, *Understanding the LINUX Kernel*, 2nd ed. Sebastopol, CA: O'Reilly, 2002.

[3] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Addison-Wesley Professional, 2014.

[4] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings, "Compartmented mode workstation: Prototype highlights," *IEEE Trans. Softw. Eng.*, vol. 16, no. 6, pp. 608–618, Jun. 1990.

[5] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for UNIX," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3.

[6] S. E. Hallyn and A. G. Morgan, "Linux capabilities: Making them work," in *Proceedings of The Linux Symposium*, Ottawa, Canada, July 2008.

[7] linuxcontainers.org, "Linux Containers," https://linuxcontainers.org/, [Online; accessed 28-March-2019].

[8] D. Documentation, "Docker," https://docs.docker.com/engine/security/security/, [Online; accessed 28-March-2019].

[9] G. Zanin and L. V. Mancini, "Towards a formal model for security policies specification and validation in the selinux system," in *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '04. New York, NY, USA: ACM, 2004, pp. 136–145. [Online]. Available: http://doi.acm.org/10.1145/990036.990059

[10] H. Chen, N. Li, and Z. Mao, "Analyzing protection quality of security-enhanced operating systems," in *Proceedings of the 10th Annual Information Security Symposium*, ser. CERIAS '09. West Lafayette, IN: CERIAS - Purdue University, 2009, pp. 8:1–8:1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2788357.2788369

[11] X. Hu, J. Zhou, S. Gravani, and J. Criswell, "Transforming code to drop dead privileges," in *2018 IEEE Cybersecurity Development (SecDev)*, vol. 00, Sept 2018, pp. 45–52. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SecDev.2018.00014

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Berlin, Heidelberg: Springer-Verlag, 2007.

[13] Argus Systems Group, Inc., "Security features programmer's guide," Savoy, IL, September 2001.

[14] Solar Designer, "return-to-libc attack," August 1997, https://insecure.org/sploits/linux.libc.return.lpr.sploit.html [Online; accessed 11-March-2019].

[15] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data Attacks Are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium (SEC)*, Baltimore, MD, 2005, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251410

[16] N. Provos, "Improving host security with system call policies," in *12th USENIX Security Symposium*, August 2003.

[17] A. One, "Smashing the Stack for Fun and Profit," *Phrack*, vol. 7, November 1996, http://www.phrack.org/issues/49/14.html [Online; accessed 11-March-2019].

[18] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October 2007, pp. 552–561.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO'04. Palo Alto, CA: IEEE Computer Society, 2004, pp. 75–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[20] C. Lattner *et al.*, "LLVM Language Reference Manual," January 2016. [Online]. Available: http://releases.llvm.org/3.7.1/docs/LangRef.html

[21] "Linux programmer's manual: mem(4)," November 1992.

[22] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in *Proceedings of the 17th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Chicago, IL, October 2010.

[23] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2013, pp. 574–588. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.45

[24] D. A. Wheeler, "SLOCCount Version 2.26," 2004.

[25] "Apachebench: A complete benchmarking and regression testing suite. http://freshmeat.net/projects/apachebench/," July 2003.

[26] S. Smalley, "Configuring the selinux policy," NSA, Tech. Report, February 2005.

[27] H. Vijayakumar, J. Schiffman, and T. Jaeger, "Integrity walls: Finding attack surfaces from mandatory access control policies," in *7th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, May 2012.

[28] J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Architecture*. Prentice Hall PTR, 2000.

[29] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information Systems Security*, vol. 13, pp. 4:1–4:40, November 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960

[30] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 147–163. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685061