Secure and Privacy-Preserving Shell as a Service

Arpitha Raghunandan, Shivram Nakkeeran Gowtham
University of Illinois at Urbana-Champaign
{arpitha2,gowtham6}@illinois.edu

Abstract

In a multi-user system, especially one with service(bot) users, there is often a need to share statistics and aggregations over file data owned by one user with other users. Providing execute access to shell scripts or binaries to other users is also a common scenario.

A quick but insecure way to do this is by providing read and execute access to the relevant files using ACL, which results in revealing the entire file/shell script. Sharing *setuid* executables with other users could be an alternative, but *setuid* has multiple security risks, requiring the executables to be carefully designed. As another alternative, adding user privilege specifications to the sudoers file can allow one user to execute a specific shell command as another user. However, this requires an admin user intervention every time such a need arises. On top of that, this approach reveals the command itself which exposes the directory structure and file names of the owning user at the least. Other approaches like writing a REST-API server to expose file data require additional effort to code, run and maintain such services.

We propose a novel mechanism that allows a user to expose privileged shell commands and scripts as a service to other users of the system in a secure and privacy-preserving manner. The operating system will provide system calls for users to register "services", which are then offered to other users in a privacy-preserving manner. The users calling a service will attain no additional information about the service as the shell command/script is hidden from them. In this paper, we discuss this mechanism in detail, addressing various design features that make it secure. We also compare our mechanism to *setuid* executables in Linux, which is known to have multiple security risks.

Keywords: Operating Systems, Privacy, Security.

1 Introduction

Production servers are often used to host multiple microservices that run as different users. The users could be service accounts owned by different teams in the organization. For instance, consider a MySQL server running on a machine as user *sql*, managed by the database team. A script located at */home/sql/scripts/restart.sh* is used to restart the server in case it goes unresponsive for whatsoever reason. The databases team wishes to give "restart" access to the user *systems* (managed by the systems team) by giving execute access to the script. Straightforward solutions discussed above either have major security risks or require additional effort. Specifically, adding user privilege specification to the sudoers file allows one user act as another user and run a specific command. For our use case, the specification would look like *systems@server: sudo -u sql /home/sql/scripts/restart.sh*. However, the *systems* user gains knowledge about the directory structure of the user *sql*, as they can now guess that the directory */home/sql/scripts/* is used to store critical scripts. Also, if neither *sql* nor *systems* users have root access, modifying the sudoers file is not possible without admin user intervention.

The privacy concerns and importance of not revealing the command is better illustrated by the following example. Suppose the requirement is to share the number of times a particular user, say *appdev* (owned by the app development team), has accessed the SQL server in the current day. Assuming the the access logs of the current day are stored in */home/sql/logs/access.log*, along with certain assumptions about the structure of the access log file, the shell command *cat /home/sql/logs/access.log | grep "user: appdev key: secret-api-key" | wc -l* retrieves what we need. Sharing the command directly with *systems* user reveals the secret API-key of *appdev* user, which is clearly a privacy concern. Even if the command did not contain any secret key, revealing it to *systems* user is clearly revealing more information than what the original requirement was, to let *systems* know the number of accesses to the MySQL server by *appdev*. Sharing specific shell commands between users is also common in a normal multi-user system. Finding occurrences of a specific pattern in a file, listing a subset of files in a directory, giving file metadata access (e.g. last modified timestamp of object files) are some examples.

In this paper, we propose a easy-to-use novel mechanism, called **privacy-preserving shell (PPS)**, to share specific shell commands between users. PPS is an API-like kernel module implemented as multiple system calls. Users can offer **services** to other users, wherein a service masquerades a shell command. Other users can see the name of the service and its description, but the actual shell command that is executed while calling the service stays hidden. This way, PPS provides maximum possible privacy to the user offering the service. For the use case mentioned above, user *sql* creates a service named “Fetch number of accesses by a user *appdev* to

the mysql server", and the user *systems* can call the service to retrieve the information. The command which contains the secret key of *appdev* is not revealed to *systems*.

Rest of the paper is organized as follows. In section 2, we discuss about *setuid* executables in Linux and its security risks. We use this as a motivation for designing PPS ensuring that it's free of these security issues. In section 3, we describe PPS and it's implementation in detail. In section 4, we discuss various security aspects and threat models. In section 5, we show the results of some performance tests that we ran for PPS. Finally, we discuss some related work and conclude in sections 6 and 7 respectively.

2 Linux SUID and Motivation

SUID (Set owner User ID up on execution) (or *setuid*) is a special type of file permissions in Linux. Executables in linux with "setuid" bit allow other users to execute them with the file-system permissions of the executable's owner. It is implemented as a part of the *exec* system call, by setting the effective user ID (euid) of the current task to the executable owner's userID, in case the *setuid* bit is set. *setgid* is a similar access-rights flag that operates on the group ID.

An alternative to our mechanism is to create a binary wrapping the shell command and use setuid flag to share it with other users. This way, the command is hidden from other users. To add the setuid flag to an executable, we can do *chmod u+s /path/to/executable*. In addition to the effort required for creating and maintaining binaries, the following security issues of setuid make this solution impractical. Though shell scripts can be directly shared (without wrapping) as they can be made executable, many operating systems ignore the setuid attribute when applied to executable shell scripts due to the underlying security risks.

2.1 Shebang - Path injection attack

Shell scripts are executed using an interpreter (such as bash) identified by the kernel using shebang(#!). When *exec* is called on a shell script, the kernel first opens it to identify the interpreter using shebang. The kernel then closes the shell script, and executes the interpreter with *argv*[1] as the path to the script. In case of a *setuid* executable, the credentials are changed during the first step itself. An exploiter can invoke an arbitrary script by creating a symlink to a setuid script. They can execute the symbolic link and modify the link after the kernel closes the shell script but before it executes the interpreter with *argv*[1] as path to the script. Race conditions can be exploited favorably using *nice* (scheduling priority modifier). *argv*[1] can be set to any script the attacker desires and the script will be executed with higher privileges [8].

2.2 Environment variables

Creating a binary wrapper around the shell script may thwart the symbolic link attack upto some extent. However, *setuid* executables are no different from normal executables when it comes to command line arguments and environment variables. The invoker has full control over the execution environment. A poorly designed setuid binary can be "fooled" to execute arbitrary code by tweaking variables like $*PATH*. An attacker can use a tool like *strings* to identify environment variables used in the executable, and modify their values to point to arbitrary locations. The attacker can also execute arbitrary code by altering *libc.so* in $LD_LIBRARY_PATH to exploit dynamically linked executables. For example, GNU libc was vulnerable to this exploit in the past [5].

Due to the above security issues and other known issues like buffer-overrun attack, most Linux based OSes don't allow *setuid* on bash scripts. Prior research works [3] [4] summarize security issues in setuid and provide guidelines to carefully design *setuid* executables. Simply wrapping bash scripts using an executable (e.g. C-program) doesn't solve the problem, as environment variables still need to be sanitized [2]. Creating an executable ensuring these issues are taken care of is a hard task in itself. Because of this, some interpreters like bash have additional safety measures to disallow wrapped execution of shell scripts with *setuid*. In addition to providing privacy, another motivation behind PPS is to come up with a secure alternative for *setuid*. We argue (in section 4) that PPS doesn't have any of the above security issues.

3 PPS - Design and Implementation

The objective of PPS is to let users share specific shell commands (including executables) in a privacy-preserving manner. In particular, we only want to share the final result of the command, i.e, *stdout* and *stderr*. We define a **service** as a facade to share a shell command, hiding it with a name and verbose description. For instance, a user can offer a service named "attendance" with description "Number of students who attended the conference". The *awk* command that sums a particular column of a file *attendance.csv* is hidden by the service from other users. When executed by other users, the service only reveals the final output according to its description.

As we wish to avoid root user intervention, services are managed by the kernel. PPS is essentially a kernel module that provides system calls to register a new service, list services offered by other users, and to execute a particular service. Services also include authorization methods to provide restrictive access to the service. We allow password based authorization and a user ID whitelist for each service listing all users that can execute the service without a password.

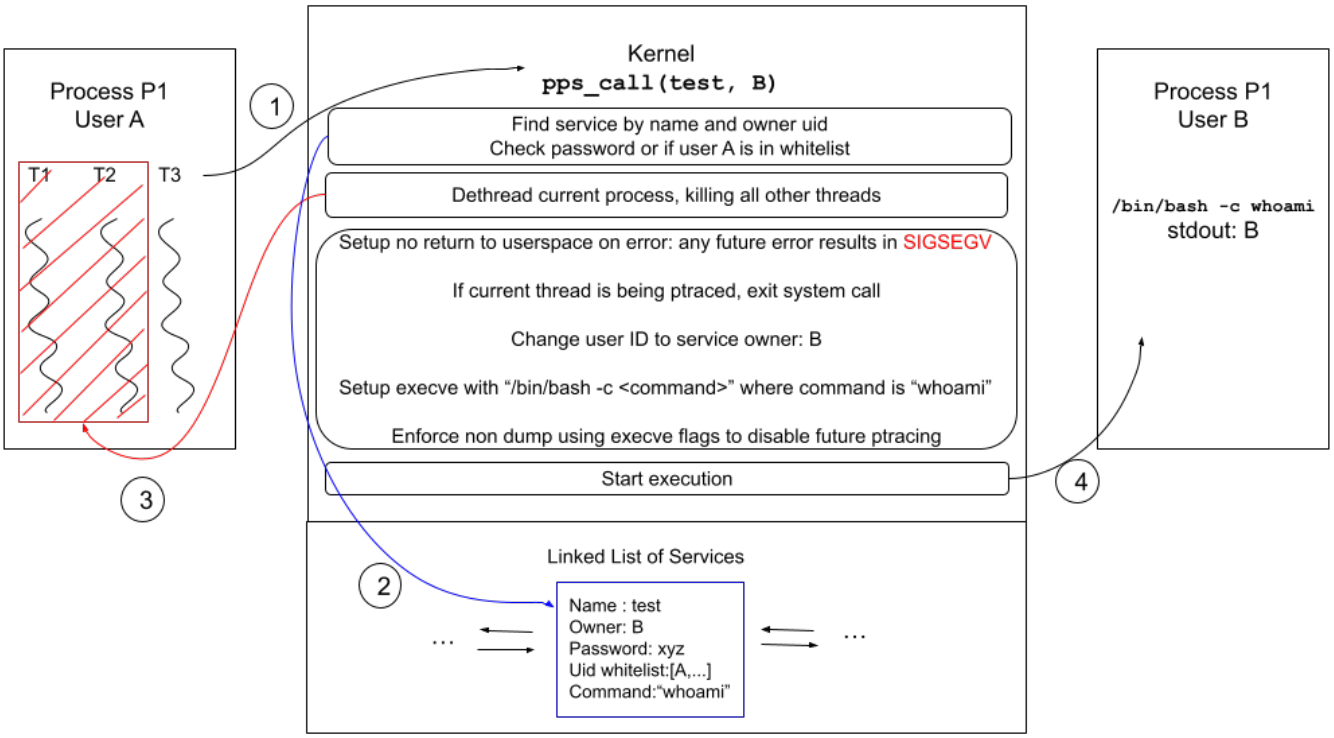

Figure 1. Step-by-step execution of pps_call

Services also have a list of environment variables that are set upon executing the service, giving full control of the execution environment to the user creating the service. Services are executed with the file-system credentials of the owner, and not the user executing it. The following code snippet shows the structure of a service in PPS.

```
// structure of a pps service
struct ppshell_service {
uid_t owner_euid; // owner's user id
char *name; // name for the service, unique per user
char *description; // verbose description of the service
char *command; // command to be executed (hidden from other users)
char *auth_pwd;// password based auth (NULL = disabled)
uid_t *auth_uid_list; // uids whitelist (NULL = disabled)
char **environ; // env vars passed to service upon execution
};
```

The current version of PPS provides 4 system calls and a user-space wrapper library which can be used to create, run and manage services: *pps_create, pps_call, pps_list* and *pps_show*. Each of these system calls are described below.

3.1 Creating a service

pps_create, as the name suggests, is used to create a new service. The system call takes all it's arguments in a C structure as shown in the above code snippet. Each service is uniquely identified by its name and its owner's user ID. All newly created services are added to a doubly-linked list in kernel address space that maintains all PPS services. The user-space wrapper for the *pps_create* system call allows the user to input all information required to create a service. The wrapper then calls the underlying system call with the user entered details packed into a C struct argument.

As explained in Section 3.2, *pps_call* spawns a bash process with the service's command as an argument, as in, */bin/bash -c <service_command>*. Even though the service hides the command from other users, the */proc* filesystem does not hide command line arguments, as they are available at */proc/PID/cmdline*. Other users can see the service's command using *ps aux* while it is executing. The proc filesystem supports a mount option *hidepid* [7] which defaults to 0, enabling access for all users to all */proc/PID/* files. Setting *hidepid* = 1 will solve our issue as it restricts access to only processes owned by the current user. However, we chose an alternative solution, our user-space wrapper for *pps_create* will encapsulate the command in a shell script and store it at */homedir/.pps/<service_name>.sh*. The service's command will be replaced by this shell script before making the system call. During execution, the command line argument to */bin/bash* will be the encapsulating script, hiding the actual command from the */proc* filesystem.

3.2 Executing a service

pps_call is an exec-type system call. Like *execve(/path/to/exec, argv[], envp[])*, *pps_call* replaces the existing process with a

new program. Therefore, the ideal way to use this system call is $fork() + pps\_call()$, which our user-space wrapper does. pps_call takes two parameters, the name and owner's user ID of the service to execute. Figure 1 gives a brief outline of the steps performed in pps_call. We first authorize the user performing the call based on user ID whitelist or password. After that, we destroy all other threads of the current process and setup "point-of-no-return" to userspace, forcing a segmentation fault on any future errors in the kernel space. If the process is currently being ptraced, we do not allow service execution. The reasoning behind these steps are explained in the next section. The user ID of the process is changed to the service owner's uid, so that the service can run with file-system credentials of the owner. Finally, pps_call performs an $execve$ of */bin/bash* with the service command passed as $argv[]$ and the service's environment variable list passed as $envp[]$. This is in contrast to the usual way $execve$ works, where the user making the system call has control over the arguments and environment variables. We use the *BINPRM_FLAGS_ENFORCE_NONDUMP* flag of $execve$ to enforce non dumpability for the new process as a security measure, explained in the next section.

The user-space wrapper for pps_call performs a $fork()$ and redirects the $stdout$ and $stderr$ streams of the child process to a pipe from which the parent process can read them. After that, the child process does pps_call, replacing it with the bash process described above. Note that the child will run with user ID of the service owner, while the parent will still run as the user calling the service. The parent waits for the child to complete and reads the output of the service from the pipe, displaying it to the user calling the service.

3.3 Listing services

pps_list lists all the services the user either owns or is authorized to execute. It displays the services' name, description and its owner's user id. A service is included in the result of *pps_list* in any of the three following cases: the user calling *pps_list* owns the service, the service can be run with a password, or the user ID of the calling user is in the list of authorized users for the given service. The wrapper for *pps_list* makes the underlying system call and displays the list in a human-readable form. *pps_list* is the only way a user can learn about services offered by other users. If a user is not authorized to run a service, then that service will not be listed when the user calls *pps_list*. This provides restrictive visibility for a service in PPS.

3.4 Service information

pps_show can be used to get all information about a particular service. Only the owner of a service will be able to use *pps_show* to get information regarding the service. The system call takes the service name as it's argument. If a service is found with the given name and with the caller's user id as the service's owner id, the information about the service, i.e. it's description, command, password, list of authorized users and list of environment variables is returned by the system call. The user-space wrapper calls the underlying system call and parses the information it receives and displays the service details in a human-readable form. The command returned to the wrapper is the path to the encapsulating shell script as described in Section 3.1. The wrapper reads the script's content and displays the original user entered command.

3.5 Summary

Apart from the above mentioned system calls and wrappers that form the essence of PPS, another helper system call named *ppshell_get_num_services* is implemented. As the name suggests, it returns the total number of services currently stored in the system.

We implemented the system calls for PPS on *Linux v*5.14 and ran our tests on *ARM*64 architecture. PPS can be easily supported for x86 by adding entries to the master syscall table located at *arch/x86/entry/syscalls/syscall_64.tbl* [12] in the Linux source tree. Figure 2 shows the changes we made to the kernel source code for PPS. Clearly, PPS is a very light module with ~ 1000 lines of code in the kernel space and ~ 600 lines of code in the user space.

```
shivramgowtham@Shivrams-MacBook-Pro privacy-preserving-shell % git diff --stat HEAD~30 HEAD
 fs/exec.c                          |  79 ++++++++++++++-
 include/linux/binfmts.h            |   4 +
 include/linux/syscalls.h           |  13 +++
 include/uapi/asm-generic/unistd.h  |  14 ++-
 include/uapi/linux/ppshell.h       |  41 ++++++++
 kernel/sys.c                       | 825 ++++++++++++++++++++++++++++++++++++++++++++++++++++
 kernel/sys_ni.c                    |   6 ++
 ppswrappers/call.c                 | 209 ++++++++++++++++++++++++++++++++++++++++
 ppswrappers/create.c               | 158 ++++++++++++++++++++++++++++++
 ppswrappers/list.c                 |  90 +++++++++++++++++
 ppswrappers/num_services.c         |  30 ++++++
 ppswrappers/show.c                 | 139 ++++++++++++++++++++++++++
 12 files changed, 1606 insertions(+), 2 deletions(-)
shivramgowtham@Shivrams-MacBook-Pro privacy-preserving-shell %
```

Figure 2. Source code changes for PPS

4 Security Issues and Threats

In this section, we summarize various possible security issues and threats in PPS and how our design avoids them.

4.1 Avoiding issues in setuid

$setuid$ is known to have various security risks, described in Section 2. pps_call does not rely on any executable path provided by the calling user, so there is no way to perform any sort of path injection attacks as in Section 2.1. Also, any environment variables set by calling-user will not be used by pps_call. We ensure that a service execution environment is

fully controlled by the owner of the service, thereby avoiding the attack described in Section 2.2.

4.2 Protection against malicious service

The security measures mentioned in this section are implemented in the user-space wrapper of *pps_call*. This way, the user calling the service has full control over these measures.

4.2.1 File descriptors

A user does $fork() + pps\_call()$ to call services as described in Section 3.2. Exec only preservers open file descriptors, rest of the process address space is completely replaced by the new program. The user can close all file descriptors in the forked process before doing *pps_call*. They have full control over which of their open descriptors are shared to the child process. Ideally, the user will close all file descriptors, create a pipe and use that as the *stdout* of the child process and read from the pipe in the parent process. Our user-space wrapper does this by default.

4.2.2 Wait with timeout

It is possible that a malicious service owner created a service that runs indefinitely. To avoid waiting forever, *pps_call* wrapper uses the WNOHANG option of *waitpid* and periodically checks if the service call has finished until a timeout is reached. Since the service runs with user ID of the owner, an indefinitely running service does not affect any system limits (such as nproc) of the user calling the service.

4.2.3 Current working directory

execve does not change the working directory. To avoid the child process(bash that runs as service owner) from learning the current directory of the calling user, we use *chdir* (set cwd to "/tmp" for instance) in the wrapper before doing *pps_call*.

4.3 Protection against malicious caller

4.3.1 Privilege escalation

PPS ensures that there is no way for the user calling the service to run arbitrary code with the user ID of the service owner. Before changing the user ID in *pps_call*, we perform dethreading, which kills all other threads of the current process as shown in Figure 1. This will help avoid privilege-escalation issues, in case the current thread in the system call is interrupted by another thread just after changing uid allowing arbitrary code to run as service owner. Any errors on or after the point of dethreading are not returned to user space, again to avoid the same issue. This is done by forcefully sending SIGSEGV to kill the process.

4.3.2 Dumpability and ptrace

Each task(thread) in Linux has a dumpability flag, which governs whether the current process can be core-dumped or be traced using ptrace. If the process is currently being ptraced by the user calling the service, we do not allow service execution as we shouldn't let the caller control the service. Immediately after checking this, we change credentials of the task to service owner (see Figure 1). Any credentials change in the kernel space automatically disables dumpability for the current task [6], disabling any further core-dump or ptrace-attach. To make this change permanent, we use the *execve* flag *BINPRM_FLAGS_ENFORCE_NONDUMP*, ensuring the service cannot be ptraced/core-dumped after start of execution. This ensures that the child process that executes the service can't be controlled by the parent process (running as the user calling the service).

4.3.3 Complex commands

We tested some commands with pipes such as *cat xxx | grep abc | wc -l*. As expected, only the final output, which goes to *stdout* is visible to the calling user. The intermediate anonymous pipes are newly opened file descriptors that are only accessible in the child process. There is no way for the calling user to read these pipes as the child process runs as a different user (owner of the service).

5 Evaluation

In a system with bot users, PPS services may be created in an automated fashion, where each user owns many services. Even in this scenario, it is unrealistic to have thousands of services as each service exposes a privileged shell command to other users. Having too many services indicates a poor ACL for files and directories. However, to test performance under unrealistic load, we create upto 13,000 services and measure the average time taken to create a service. We used a Ubuntu 20.04 ARM 64 virtual machine with 8 CPU cores and 8GB RAM for these tests. The code and results of the tests are available in our repository in the *pps_test/* directory.

Existing Services Count	N	M	Avg. Time (ms)
0	5	50	1.4
250	5	100	1.44
500	10	50	7.6
2000	10	200	7.95
3000	10	300	3.23
0	1	6000	7.88
0	20	200	41.639
0	30	300	102.650
4000	30	300	337.07

Table 1. Average time taken for pps_create

We performed stress testing for *pps_create* by running N parallel threads, each of which sequentially perform *pps_create* M times, creating $N * M$ services in total. We measured the

average time taken per system call, precise upto the nanosecond level. The first step in *pps_create* is to iterate through all existing services to check for uniqueness violation. Therefore, time taken to create a service would depend on the number of services that already exist in the system. We test for different values of N, M and number of pre-existing services in the system. Also, any access to the linked list data structure that stores the services is protected by a mutex lock. Since we have parallel threads in our tests, we are measuring the locking overhead as well.

Table 1 shows the results of our tests. *pps_create* is fairly quick upto creating 6000 services, and also when the system already has 3000 services and we add 3000 new services. In the worst case, where we have 4000 existing services and we create 9000 new services, *pps_create* takes around $337ms$ per service. Note that the times measured may be a little misleading, as we are measuring time in user-space, which has a additional system call overhead to get current time. Also, since our tests have upto 30 threads on a 8-core CPU, the time includes context switching overhead as well. This overhead is clearly visible as the single threaded case with 6000 services ($N = 1, M = 6000$) is much faster than the multi-threaded test with 4000 services ($N = 20, M = 200$). However, the aim was to stress test *pps_create* under unrealistic scenarios, rather than to measure the exact time taken by the system call.

6 Related Work

Privacy as an operating system service has been discussed in [11]. They discuss about deploying privacy mechanisms in kernel and system libraries and argue that it provides transparency and ease of management. Among the different deployment architectures discussed in the paper, PPS can be thought of as a "white box" deployment, as it completely resides in the kernel space.

There has been prior work on limiting privilege escalation issues in *setuid* such as policy-based privilege elevation [9] and protection domains within setuid binary [10]. There is also work on pruning unnecessary setuid binaries in Linux distributions. Fedora [14] and Ubuntu [13] have filesystem capabilities as an alternative to reduce the need for certain setuid binaries. Protego [1] deprivileges kernel code and provides a framework to migrate policies from setuid binaries into the kernel, thereby getting rid of them.

7 Conclusion

Privacy-preserving shell or PPS is a convenient, novel method that allows users to share specific shell commands and scripts as a service to other users of a system in a secure manner. PPS is an in-kernel module implemented as multiple system calls. As explained in the previous sections, PPS is designed to handle various security issues and threats, especially related to privilege escalation, making it a viable and secure alternative to *setuid* executables in Linux. The PPS design takes into consideration the various drawbacks of using *setuid* mechanism and delivers a solution that overcomes them. We hope to pitch PPS as an alternative to *setuid* to the Linux community. As a future research direction, we will look into using these concepts in cloud systems.

References

[1] Bhushan Jain, Chia-Che Tsai, J. J., and Porter, D. E. Practical Techniques to Obviate Setuid-to-Root Binaries. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)* (Apr. 2014).

[2] Brown, N. Ghosts of Unix past, part 4: High-maintenance designs. https://lwn.net/Articles/416494/, 2010.

[3] Chen, H., Wagner, D., and Dean, D. Setuid Demystified. In *11th USENIX Security Symposium (USENIX Security 02)* (2002).

[4] Dittmer, M. S., and Tripunitara, M. V. The UNIX Process Identity Crisis: A Standards-Driven Approach to Setuid. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)* (2014).

[5] Edge, J. Two glibc vulnerabilities. https://lwn.net/Articles/412048/, 2010.

[6] Linux source code: Credential change disables dumpability. https://elixir.bootlin.com/linux/v5.14/source/kernel/cred.c#L468.

[7] proc(5) — Linux manual page. https://man7.org/linux/man-pages/man5/proc.5.html#DESCRIPTION.

[8] Mascheck, S. Setuid support: The #! magic, details about the shebang/hash-bang mechanism on various Unix flavours. https://www.in-ulm.de/~mascheck/various/shebang/#setuid.

[9] Provos, N. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium* (Aug. 2003).

[10] Shinagawa, T., and Kono, K. Implementing A Secure Setuid Program. In *Proceedings of the Conference on Parallel and Distributed Computing and Networks* (2004).

[11] Sotiris Ioannidis, Stelios Sidiroglou, A. D. K. Privacy as an Operating System Service. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Security (HotSec'06)* (2006).

[12] Linux source code: x86 syscall table. https://elixir.bootlin.com/linux/v5.14/source/arch/x86/entry/syscalls/syscall_64.tbl.

[13] Ubuntu wiki: Filesystem Capabilities. https://wiki.ubuntu.com/Security/Features#fscaps.

[14] Walsh, D. Fedora wiki: Features/RemoveSETUID. https://fedoraproject.org/wiki/Features/RemoveSETUID, 2011.