Decap: Deprivileging Programs by Reducing Their Capabilities

Md Mehedi Hasan
Stony Brook University
Stony Brook, USA
mdhasan@cs.stonybrook.edu

Seyedhamed Ghavamnia
Stony Brook University
Stony Brook, USA
sghavamnia@cs.stonybrook.edu

Michalis Polychronakis
Stony Brook University
Stony Brook, USA
mikepo@cs.stonybrook.edu

ABSTRACT

Linux enables non-root users to perform certain privileged operations through the use of the setuid ("set user ID") mechanism. This represents a glaring violation of the principle of least privilege, as setuid programs run with full superuser privileges—with disastrous outcomes when vulnerabilities are found in them. Linux capabilities aim to improve this situation by splitting superuser privileges into distinct units that can be assigned individually. Despite the clear benefits of capabilities in reducing the risk of privilege escalation, their actual use is scarce, and setuid programs are still prevalent in modern Linux distributions. The lack of a systematic way for developers to identify the capabilities needed by a given program is a contributing factor that hinders their applicability.

In this paper we present Decap, a binary code analysis tool that automatically deprivileges programs by identifying the subset of capabilities they require based on the system calls they may invoke. This is made possible by our systematic effort in deriving a complete mapping between all Linux system calls related to privileged operations and the corresponding capabilities on which they depend. The results of our experimental evaluation with a set of 201 setuid programs demonstrate the effectiveness of Decap in meaningfully deprivileging them, with half of them requiring fewer than 16 capabilities, and 69% of them avoiding the use of the security-critical `CAP_SYS_ADMIN` capability.

CCS CONCEPTS

• **Security and privacy → Software and application security**; *Operating systems security*.

KEYWORDS

Linux capabilities, setuid programs, privileges

ACM Reference Format:
Md Mehedi Hasan, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2022. Decap: Deprivileging Programs by Reducing Their Capabilities. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3545948.3545978

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10…$15.00
https://doi.org/10.1145/3545948.3545978

1 INTRODUCTION

Unix systems split privileges into two categories, with processes being either privileged or unprivileged. Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checks. This all-or-nothing approach is in stark contrast with the principle of least privilege [45], which states that users, processes, and programs should only have the bare minimum privileges necessary to complete their task. Having only root or non-root privileges complicates cases in which unprivileged users must perform a *specific* privileged operation. For example, unprivileged users who want to change their password must be able to make modifications in two files owned by root, `/etc/passwd` and `/etc/shadow`, but the permissions of these two files prevent any unprivileged process from modifying them.

The "set user ID" mechanism [42] was introduced to address the issues caused by this coarse-grained split of privileges, and allow unprivileged processes to perform operations that otherwise are accessible solely by privileged processes. Executable files contain a `setuid` bit in their attributes, which denotes that when launched, the process will be granted the privileges of the file's owner, i.e., superuser privileges in case of setuid executables owned by root. Although the `setuid` bit offers a pragmatic solution to the above issues, allowing *unprivileged* users to execute programs with superuser privileges can lead to disastrous outcomes. Vulnerabilities in setuid programs are prevalent [13, 26, 28, 48, 51], allowing attackers to achieve arbitrary code execution with root privileges.

Providing superuser privileges to a process only to perform a very narrow-scope privileged operation is an overkill, and has been a necessity due to the existence of only two privilege levels. Starting with kernel v2.2, Linux provides a more flexible solution by dividing the privileges associated with the superuser account to more fine-grained distinct units called *capabilities*. This allows a process to acquire only the specific capabilities associated to the privileged operations it needs to perform.

Initially, superuser privileges were divided into 27 capabilities. While some capabilities remain overloaded (e.g., `CAP_SYS_ADMIN`, which is essentially equivalent to superuser permissions), additional capabilities are still being introduced—their number has increased to 41 in the most recent Linux kernel (v5.17 at the time of writing). Granting distinct capabilities allows programs to perform privileged operations without the risk of running them with superuser privileges. It would then be expected to observe a declining number of programs using the `setuid` bit in more recent Linux distributions, and an increasing transition to the use of capabilities.

Surprisingly, despite the important security benefits of choosing just the necessary capabilities for a given task, the transition from setuid-based to capability-based programs has been extremely slow. As we discuss in Section 2.3, among 201 setuid programs in the Ubuntu 18.04 distribution (released in April 2018), only *seven* have

transitioned to the use of capabilities in the latest Ubuntu version (21.10 at the time of writing, released in October 2021). This is a striking observation that highlights the underutilization of capabilities as a defense-in-depth mechanism in modern systems, and is the main motivation behind our work.

We have identified two important factors that hinder the applicability of capabilities in real-world programs. First, there is no authoritative and systematized reference for developers to infer the capabilities required for a given privileged operation. The relevant information is currently scattered across various man pages, and for several capabilities it is incomplete or missing. This makes it quite challenging to identify the right set of capabilities for a given privileged operation, or even inferring whether an operation requires any capability in the first place. Second, given the large number of existing setuid programs in Linux distributions, converting them to become capability-aware would be a tedious and time-consuming effort if performed manually.

In this work we present *Decap*, a generic and practical tool for taking advantage of the full potential of Linux capabilities towards *deprivileging* programs. Given the lack of ground truth for inferring the capabilities required by a given privileged operation, in the initial phase of our work we performed a thorough investigation of all available Linux capabilities and the system calls they affect, to derive a detailed and complete mapping between all system calls related to privileged operations and their respective capabilities. Using this pre-generated mapping, Decap performs static binary code analysis of a given program and its libraries to identify the set of all possible system calls it may invoke (in a conservative, best-effort way), and then derives the set of capabilities required by the program.

Among the available capabilities, `CAP_SYS_ADMIN` is quite overloaded and heavily used as a "catch all" capability—so much so that it is known as the new *root* [22]. Removing this capability from the minimal set extracted for a given program is thus crucial for achieving meaningful attack surface reduction. During our initial investigation, however, we observed that `CAP_SYS_ADMIN` is associated with the highest number of system calls among all capabilities, including commonly used ones such as `clone`. This would lead to its inclusion in all over-privileged programs in our data set, essentially eliminating the security benefits of deprivileging them.

Fortunately, several system calls depend on `CAP_SYS_ADMIN` only under certain (and often rare) circumstances, associated with specific argument values. For these system calls, we have extended our mapping to also consider the particular argument values that depend on `CAP_SYS_ADMIN`. Decap then performs argument value analysis on the invocation sites of these system calls to determine whether this security-critical capability is actually required.

We evaluated the effectiveness of Decap in deprivileging programs by collecting a set of 201 setuid programs extracted from more than 75K Ubuntu packages. Our results show that Decap achieves a significant reduction in the privileges required by existing setuid programs, with half of them requiring fewer than 16 capabilities for their correct operation. More importantly, Decap removes the security-critical `CAP_SYS_ADMIN` capability from 69% of the programs in our dataset. We also demonstrate how deprivileging mitigates the effects of exploiting previously disclosed vulnerabilities in these programs.

The main contributions of our work include:

- We present the first systematic effort in deriving a complete mapping between system calls related to privileged operations (and in certain cases their arguments), and the corresponding capabilities on which they depend.
- We implemented Decap, a tool that automatically deprivileges programs by identifying all system calls they may invoke using binary code analysis, and then deriving the set of required capabilities using the above mapping.
- We experimentally evaluated Decap using a set of 201 setuid programs, and demonstrate its effectiveness in meaningfully deprivileging them, with half of them requiring fewer than 16 capabilities, and 69% of them avoiding the use of the security-critical `CAP_SYS_ADMIN` capability.

Our Decap prototype is publicly available as an open-source project from https://github.com/hasanmdme/decap.

2 BACKGROUND AND MOTIVATION

Privilege separation enhances security by restricting critical operations only to the users or processes that need to perform them. Before the introduction of capabilities, Linux essentially provided only two privilege levels, corresponding to the superuser (root) and standard user accounts. Although standard users do not have the required privileges to perform critical operations, they often do need to access critical resources that are otherwise accessible only by the root account. Traditionally, this need has been addressed through the use of the `setuid` attribute of executable files, which essentially allows regular users to invoke certain programs with superuser privileges—a powerful capability that often leads to misuse.

Although their code can perform only a few specific critical operations, setuid programs have full privileges to perform *any* critical operation. Consequently, attackers can exploit vulnerabilities in setuid programs to escalate their privileges and fully compromise the system. To mitigate this issue, Linux divides superuser privileges into distinct *capabilities* that can be independently assigned to non-privileged programs. Instead of providing blanket access to any critical operation through `setuid`, capabilities provide fine-grained access to only the specific privileges a program requires.

2.1 Linux Privileges

When a new process is created, it inherits the real user ID and group ID of its parent. A process whose effective user ID is 0 has all the privileges of the superuser, and is called a *privileged* process. When a setuid ("set user ID") executable is invoked, the created process automatically runs with the privileges of the file's owner (correspondingly, `setgid` allows users to run an executable with the privileges of the executable's group) [21]. Setting the `setuid` bit on executables owned by root (user ID 0) allows users to invoke programs with superuser privileges. For example, the `passwd` program for changing a user's password is owned by root and has the `setuid` bit enabled, so that it can read and modify the `/etc/passwd` and `/etc/shadow` files when invoked by non-privileged users.

When running a setuid program, the created process can perform any critical operation supported by the kernel, because privileged processes bypass all kernel permission checks. To mitigate this security risk, the typical solution is to drop the effective privileges

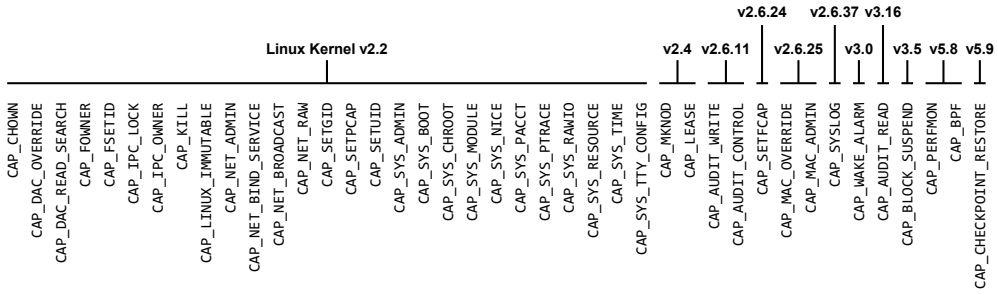

Figure 1: The 41 capabilities available in the current Linux kernel, and the kernel version in which they were introduced.

of the process once the required critical operation has completed, and temporarily reacquire them only when needed. Attackers, however, can always elevate privileges that have been only temporarily dropped. To provide meaningful security, a process should thus drop its privileges *permanently*. This is a common design pattern especially for server applications, which often need to perform privileged operations only during their initialization phase (e.g., bind an IPv4 socket to a port number lower than 1024) [37].

2.2 Linux Capabilities

Linux *capabilities* aim to mitigate the risk posed by overprivileged programs. When performing security checks in the kernel, instead of making a binary decision based on whether superuser privileges are wielded, the superuser privilege is divided into different capabilities [27]. A process can carry out a privileged operation only if it has the appropriate capability. Each capability is typically associated with a specific privileged operation (or group of related operations), which allows it to bypass the corresponding access control rules [3]. For example, the `CAP_NET_BIND_SERVICE` capability allows a non-root process to bind a network socket to privileged ports—a very specific need for which `setuid` is an overkill.

At the time of writing, the most recent Linux kernel (v5.17) provides 41 capabilities, listed in Figure 1 along with the kernel version in which they were introduced. A process may temporarily raise its privileges by moving capabilities from its *permitted* to its *effective* set, and lower its privileges by removing them from the effective set. As a general practice, capability-aware programs should initially have all their capabilities disabled in the effective set, and wield them only when needed.

2.3 The Incapability of Using Capabilities

The benefits of capabilities compared to setuid programs in terms of reducing the risk of privilege escalation are clear. A setuid program containing an exploitable vulnerability can lead to privilege escalation and full system compromise. Reducing process privileges to those absolutely necessary through the use of capabilities limits significantly the harmful operations an attacker may perform.

Surprisingly, although capabilities were introduced in Linux more than two decades ago (with the v2.2 kernel release in 1999), their actual use is scarce, and setuid programs are still prevalent in modern Linux distributions. As we discuss in Section 7.1, we identified 201 setuid programs as part of the Ubuntu 18.04 distribution, which was released in 2018. Our intuition for choosing a relatively older distribution was that it would contain many more setuid programs—which we needed for our experimental evaluation—compared to more recent distributions. When we analyzed the subsequent versions of these programs in later Ubuntu distributions, however, we found that just *seven* of them have become capability-aware in the latest (at the time of writing) Ubuntu 21.10 release, as shown in Table 1. Moreover, across more than 59K software packages in the same Ubuntu 21.10 release, we found just 29 capability-aware programs (the details of which are provided in Table 5 in the Appendix).

Table 1: Out of 201 setuid programs in Ubuntu 18.04, only seven have transitioned to the use of capabilities in the latest Ubuntu release (✗ == setuid, ✓ == capabilities).

Program	Ubuntu 18.04 (April 2018)	Ubuntu 20.04 (April 2020)	Ubuntu 21.10 (October 2021)
`ping`	✗	✓	✓
`ping6`	✗	✓	✓
`noping`	✗	✗	✓
`traceroute6.iputils`	✗	✓	✓
`arping`	✗	✓	✓
`oping`	✗	✗	✓
`pinger`	✗	✓	✓

We can only speculate on the reasons for the scarce use of capabilities, but the lack of a systematic way for developers to identify the capabilities needed by a given program is certainly a contributing factor. By automating the process of identifying the set of capabilities needed by a given program, Decap aims to ease the transition of existing setuid programs to become capability-aware, and to facilitate the use of capabilities when developing new applications.

3 THREAT MODEL

Our attack model assumes adversaries who can exploit vulnerabilities in setuid programs. These include both remote attackers who have achieved arbitrary remote code execution in a vulnerable network-facing setuid program, and local (non-privileged) users who can invoke any (vulnerable) setuid program available on the system. The outcome in both cases is full system access with superuser privileges, assuming that the corresponding executable file is owned by root (in which case the spawned process gains superuser privileges). Decap generates a deprivileged, capability-aware version of a given input program. Attackers can still use any capability included in a process' permitted capability set by moving it into

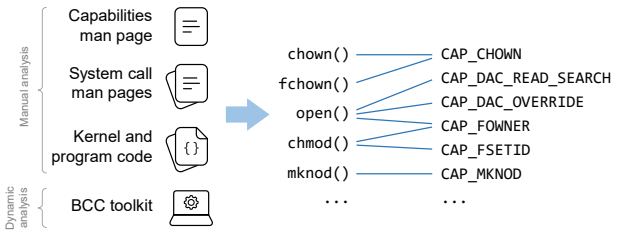

Figure 2: The different sources of information used to map system calls to capabilities.

the effective set (if not already included). As a defense-in-depth approach, Decap thus strives to limit the set of permitted capabilities for a given program.

4 SYSTEM CALL TO CAPABILITY MAPPING

User-space programs perform privileged operations by invoking certain system calls. Currently, however, there is no generic and systematic way to derive the relationship between a given system call and the capabilities it requires (if any). To that end, the initial phase of our work involved a thorough investigation of all available capabilities and the system calls they affect. The result of this investigation is a mapping between the system calls and respective capabilities available in Linux, a summary of which is provided in Table 2 (for space reasons in its reverse representation, i.e., mapping capabilities to system calls). We plan to maintain a publicly available on-line version of this mapping, and continue maintaining it and refining it as part of our future work in this area. Given this pre-generated mapping, Decap performs static binary code analysis of a target program and its libraries to extract the set of all possible system calls it may invoke, and then to derive and enforce the corresponding set of required capabilities.

The ultimate ground truth about the capabilities required by a given system call is reflected in the source code of the kernel. Unfortunately, automatically mapping system calls to their capabilities by performing static analysis on the Linux kernel source code is a daunting task. Extracting the complete kernel control flow graph and identifying all possible paths that are guarded by capability checks, while not impossible, is extremely challenging and results in severe overapproximation [24, 54]. Instead, we derived the mapping between each system call and the capabilities it requires through a combination of manual and automated analysis.

As summarized in Figure 2, we manually gathered information about dependencies between system calls and capabilities by systematically examining the relevant Linux man pages, and by carefully analyzing the Linux kernel's source code. Based on this information, we generated an initial system call to capability mapping, which we then further verified and refined through dynamic analysis using custom capability-specific test programs.

Among the available capabilities, `CAP_SYS_ADMIN` is the most powerful, essentially granting root privileges to the application. Although one would expect such a powerful capability to be rarely used, the current situation is actually the opposite. Due to the lack of a central authority for determining how capabilities should be assigned to privileged operations, kernel developers tend to overuse

Table 2: The resulting mapping of our effort to identify the system calls that depend on each capability.

Capability	System Calls
CAP_AUDIT_CONTROL	sendto, recv, recvfrom, recvmsg
CAP_AUDIT_READ	bind
CAP_AUDIT_WRITE	sendto
CAP_BLOCK_SUSPEND	epoll_ctl
CAP_SYS_ADMIN	bpf, perf_event_open, syslog, mount, umount, pivot_root, swapon, swapoff, setdomainname, vm86, setns, fanotify_init, unshare, lookup_dcookie, io_submit, prctl, clone, quotactl, msgctl, setrlimit, shmctl, ioprio_set, keyctl, madvise, ioctl, seccomp, ptrace, sethostname
CAP_BPF	bpf
CAP_PERFMON	perf_event_open
CAP_SYSLOG	syslog
CAP_CHECKPOINT_RESTORE	clone
CAP_CHOWN	chown, fchown, lchown, fchownat
CAP_DAC_READ_SEARCH	open, openat, openat2, open_by_handle_at, linkat
CAP_DAC_OVERRIDE	utime, utimensat, utimes, open, openat, openat2
CAP_FOWNER	chmod, fchmod, fchmodat, utime, utimes, utimensat, unlink, unlinkat, open, openat, openat2, fcntl, rename, renameat, renameat2, rmdir, ioctl
CAP_LEASE	fcntl
CAP_FSETID	chmod, fchmod, fchmodat
CAP_IPC_LOCK	mlock, mlock2, mlockall, mmap, memfd_create
CAP_IPC_OWNER	msgrcv, msgsnd, semop, semtimedop, shmat, shmdt, msgctl, msgget, shmctl
CAP_KILL	kill, ioctl
CAP_LINUX_IMMUTABLE	ioctl
CAP_MAC_ADMIN	setxattr, lsetxattr, fsetxattr
CAP_MAC_OVERRIDE	socket
CAP_MKNOD	mknod, mknodat, renameat2
CAP_NET_ADMIN	setsockopt, ioctl
CAP_NET_BIND_SERVICE	bind
CAP_NET_BROADCAST	
CAP_NET_RAW	socket
CAP_SETGID	setgroups, setfsgid, setgid, setregid, setresgid
CAP_SETFCAP	clone
CAP_SETPCAP	capset, prctl
CAP_SETUID	setuid, setreuid, setresuid, setfsuid, keyctl
CAP_SYS_BOOT	reboot, kexec_file_load, kexec_load
CAP_SYS_CHROOT	chroot, setns
CAP_SYS_MODULE	finit_module, init_module, create_module, delete_module
CAP_SYS_NICE	sched_setscheduler, sched_setparam, sched_setattr, migrate_pages, setpriority, sched_setaffinity, nice, ioprio_set, move_pages, spu_create, mbind
CAP_SYS_PACCT	acct
CAP_SYS_PTRACE	ptrace, userfaultfd, kcmp, set_robust_list, process_vm_readv, process_vm_writev
CAP_SYS_RAWIO	iopl, ioperm
CAP_SYS_RESOURCE	send, sendto, sendmsg, prctl, msgctl, setrlimit, fcntl, prlimit, mq_open, ioctl
CAP_SYS_TIME	settimeofday, stime, adjtimex, clock_adjtime, ntp_adjtime
CAP_SYS_TTY_CONFIG	vhangup, ioctl
CAP_WAKE_ALARM	timer_create, timerfd_create

`CAP_SYS_ADMIN` as a "catch-all" capability for newly introduced kernel features [22]. This was reflected in our preliminary results, as our analysis indicated that *all* `setuid` programs in our data set would have to retain `CAP_SYS_ADMIN` due to the large number of system calls mapped to this capability, essentially making the whole effort pointless. Fortunately, we observed that for many of these system calls, the dependency on `CAP_SYS_ADMIN` is actually needed only for exceptional cases, associated with specific argument values (e.g., certain flags). We thus performed a second round of manual and automated analysis for the set of 28 system calls requiring `CAP_SYS_ADMIN`, this time to derive a more fine-grained mapping between certain system call *argument values* and the `CAP_SYS_ADMIN` capability.

4.1 System Calls

4.1.1 Man Pages. To build our initial mapping, we used information extracted from both the man page of each system call and the `capabilities` man page. The latter provides a description of each

capability and a (partial) list of relevant system calls. For example, the capabilities man page mentions that CAP_CHOWN, which allows a process to arbitrarily modify the owner and group of a file, is associated to the chown system call, which is used to perform this privileged operation.

Nevertheless, the capabilities man page is not complete, and the affected system calls of some capabilities are not explicitly mentioned. Instead, for some capabilities, the page only briefly mentions the privileged operations that a process can perform. For example, the documentation of CAP_SETGID only mentions that a process requires this capability to make arbitrary modifications to the process GIDs, without explicitly specifying the system calls that can be used to perform this operation. Since system calls (or groups of related system calls) have their own separate man pages, which (often, but not always) list their required capabilities, we used the man pages of system calls to complement the initial mapping derived by analyzing the capabilities man page. For example, the man page of the setgroups system call explicitly mentions CAP_SETGID as a requirement.

4.1.2 Source Code. Although the capabilities and system call man pages contain extensive information regarding system call capability requirements, relying solely on this information would result in an incomplete mapping. Therefore, we examined the source code of user-space programs and the kernel to identify additional requirements. For example, the man page of the sendto system call does not mention any capabilities, but points to additional man pages of related utilities. One of them is auditctl, which is used to control the kernel's audit system. Its man page mentions that the -m flag (used for sending user-space messages to the audit system) requires the CAP_AUDIT_WRITE capability. By examining the part of auditctl's source code related to this feature, we observed that it uses the sendto system call to perform this operation, and we thus augmented our mapping accordingly.

Finally, we inspected the kernel's source code to identify additional capability requirements. For a given capability, we first identify any conditional branches that depend on the result of the corresponding capability check. We then find the callers of the function containing the check, and continue iteratively until we reach the code of a system call. We follow this approach mostly for cases in which the capabilities man page provides only a generic description of a given capability, without explicitly mentioning any related system calls.

An indicative example is CAP_DAC_OVERRIDE, for which no system calls are mentioned in its documentation. By searching through the kernel code, we identified corresponding capability checks related to generic permission checks when a process attempts to open a file by invoking the open, openat, or openat2 system calls. If a process does not have permission to open a requested file (e.g., because the process is run by a different user than the file's owner), CAP_DAC_OVERRIDE can be used to *override* that permission.

4.1.3 Dynamic Analysis. As a final verification step, we used dynamic analysis to validate the mappings derived from the previous steps, and refine them as needed. To that end, we used the capable [12] tool from the BPF Compiler Collection (BCC) toolkit [35] to trace the capability checks performed by the kernel during a program's execution. We implemented many small test programs, each tailored to performing a certain privileged operation, and launched them using capable to identify the corresponding capability checks that get triggered.

For example, to test the mapping of the bind system call to the CAP_NET_BIND_SERVICE capability, we implemented a program that binds a socket to a privileged port. Running this program as a regular (non-root) user on top of capable raises an error when attempting to bind to a privileged port, and CAP_NET_BIND_SERVICE is returned as the missing capability. We followed a similar approach to validate several other undocumented mappings, including open to CAP_DAC_OVERRIDE, socket to CAP_NET_RAW, clone to CAP_SYS_ADMIN, and chmod to CAP_FOWNER.

Table 3: System calls that conditionally depend on CAP_SYS_ADMIN according to certain values of the highlighted arguments.

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg,
    ... /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */);
int prctl(int option, unsigned long arg2, unsigned long
    arg3, unsigned long arg4, unsigned long arg5);
int quotactl(int cmd, const char *special, int id, caddr_t addr);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int setrlimit(int resource, const struct rlimit *rlim);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
int syscall(SYS_ioprio_set, int which, int who, int ioprio);
long syscall(SYS_keyctl, int operation, unsigned long arg2,
    unsigned long arg3, unsigned long arg4, unsigned long arg5);
int madvise(void *addr, size_t length, int advice);
int ioctl(int fd, unsigned long request, ...);
int syscall(SYS_seccomp, unsigned int operation, unsigned
    int flags, void *args);
long ptrace(enum __ptrace_request request, pid_t pid,
    void *addr, void *data);
```

4.2 System Call Arguments

Among the capabilities available in Linux, CAP_SYS_ADMIN is the single most security-sensitive one, as it provides a superset of multiple privileges to a process. Therefore, removing this capability from a given binary is more important than removing any other capability. Our initial mapping identified 28 system calls requiring CAP_SYS_ADMIN, including some very commonly used ones such as clone, ioctl, keyctl, prctl, and madvise. Upon closer inspection, however, we discovered that 12 of them, listed in Table 3, require it *only* under special circumstances, related to specific system call arguments. For example, the clone system call requires CAP_SYS_ADMIN only if its third argument (flags) is set to CLONE_NEWIPC.

Based on this observation, we extended our system call to capability mapping for these 12 system calls by capturing the particular argument values that require the CAP_SYS_ADMIN capability. We followed a similar approach as discussed in Section 4.1, which involved manual analysis of man pages and source code to identify the specific argument values that necessitate the inclusion of CAP_SYS_ADMIN. In some cases, the argument values are explicitly mentioned in the man pages. For example, the man page of the msgctl system call mentions that if the value of the second argument (cmd) is IPC_SET or IPC_RMID, it requires CAP_SYS_ADMIN.

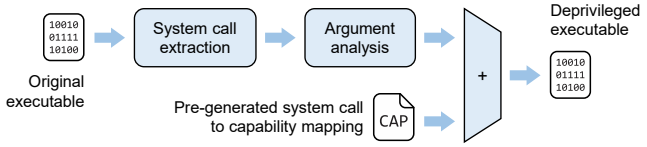

Figure 3: Overview of Decap's operation for identifying the capabilities required by a given program.

When inspecting the kernel source code, we first identified all functions that perform checks for `CAP_SYS_ADMIN`, and then recursively identified the callers of these functions. While traversing the callers, we looked for call sites that are guarded by a system call argument check. If the capability check can only be reached under certain circumstances when the system call argument has a specific value, we consider the value as *sensitive* and map the capability to both the system call and the particular argument value. For example, `ioctl` requires `CAP_SYS_ADMIN` if the value passed as its second argument (`request`) is `EXT4_IOC_CHECKPOINT`. This is because there is a `CAP_SYS_ADMIN` check in the `ext4_ioctl_checkpoint` function, the call site of which is only reachable if the `request` argument has been set to `EXT4_IOC_CHECKPOINT`.

5 DECAP DESIGN

Our main goal is to deprivilege programs by providing them only with the essential capabilities necessary for their correct operation. This requires the precise identification of the privileged operations a program may perform during its entire lifetime. Dynamic analysis by tracing the capability checks performed by the kernel [12] is a useful way to derive this information during the development of new applications, or while testing existing applications. Dynamic analysis, however, is not adequate for developing a *generic* approach to perform this task automatically for arbitrary programs, as exercising all possible program paths (especially for complex applications) is infeasible.

Aiming to provide a *practical* solution to this problem, readily applicable to existing over-privileged Linux programs, Decap relies solely on static code analysis to identify the set of capabilities required by a given program. Relying on static analysis means that Decap operates in a best-effort way by ensuring *soundness* but not *completeness*. Given the set of all available Linux capabilities, our primary aim is to ensure that the subset of removed capabilities is sound, i.e., that the deprivileged version of the program will continue to operate correctly. On the other hand, due to the inherent overapproximation of static code analysis (and for some system calls, of our own pre-generated mapping as well, as discussed in Section 8), the set of removed capabilities may not be complete, i.e., some retained capabilities may not be actually needed and could be also removed.

Figure 3 presents an overview of Decap's operation. Given a target application, Decap relies on existing tools to identify the set of system calls required by the main executable and all its libraries. For the set of 12 system calls that conditionally require `CAP_SYS_ADMIN`, Decap then analyzes all their previously identified call sites to extract the values passed to the specific arguments that require this capability. Finally, based on the pre-generated mapping of system calls to capabilities, Decap identifies and enforces the set of capabilities required by the application. Decap runs its analysis on the program binary and does not require its source code. We only require the source code of the libc library to build its callgraph.

5.1 System Call Identification

The first step in deprivileging a setuid program is to identify its required system calls. The problem of identifying the required system calls of a given application has been extensively explored by prior works, several of which provide open-source tools for extracting the system call requirements of a given application by performing static analysis [8, 10].

Confine [10] is a container hardening framework that performs static analysis on the applications running in a container to extract their required system calls. For a given application, Confine considers both system calls invoked directly and those invoked through libc functions. Direct system call invocations are identified through binary code analysis, while the required libc functions are inferred from the import table of each binary. Libc functions are mapped to their respective system call(s) by analyzing the libc source code to build a precise callgraph. This leads to a more precise libc callgraph compared to other binary-only approaches. However, Confine suffers from overapproximation because it does not use library specialization [1, 40] as part of its analysis, and instead includes *all* system calls used across *all* linked libraries. Consequently, any system calls *exclusively* required by library functions that are not imported by the target application are still considered as required. For example if an application depends on the Openssl library, Confine considers all the libc functions imported by Openssl regardless of their reachability.

Sysfilter [8], on the other hand, relies on a purely binary-level approach for identifying the system calls required by a given application. It uses Nibbler [1] to perform library specialization, and as a result, it excludes more system calls from loaded libraries compared to Confine. However, its binary-only analysis of libc results in a less accurate callgraph, leading again to overapproximation.

Although both Confine and Sysfilter suffer from overapproximation that results in the inclusion of more system calls than actually needed, the underlying reasons are different. Based on our experience with both tools, any system calls that are only identified by one of them and not the other are indeed unneeded by the application, and are included solely as a result of overapproximation in their analysis. Given that 126 out of the 334 system calls available in Linux v.5.4 may impose capability requirements (as discussed in Section 7.3), any additional system call that Decap mistakenly considers as required can significantly hinder its effectiveness, especially in case this leads to the inclusion of a critical capability such as `CAP_SYS_ADMIN`.

Until a more accurate system call identification approach becomes available, and given that the results of both Confine and Sysfilter are still *sound*, Decap analyzes the target application with both tools, and considers the *intersection* of the two reported sets of required system calls for deriving the corresponding capabilities. This is an optimal solution compared to using solely one of the two tools, as it takes advantage of the benefits of both tools to increase the precision of system call extraction.

5.2 System Call Argument Identification

As we mentioned in Section 4.2, the CAP_SYS_ADMIN capability is required by multiple system calls but only when invoked with a limited set of specific argument values. Therefore, once the set of required system calls has been extracted, Decap performs argument-level analysis for the 12 system calls that conditionally require CAP_SYS_ADMIN, and attempts to extract the concrete values passed to the arguments that determine whether CAP_SYS_ADMIN is required (listed in Table 3), across all their call sites.

The first step in this process is to identify the invocation sites of these system calls. Decap performs binary analysis on the target application and its libraries to identify both direct invocations and invocations through libc functions. Once all call sites are identified, Decap performs a single-level backwards inter-procedural data flow analysis, starting from each call site, to extract the concrete value passed to each argument related to CAP_SYS_ADMIN.

Besides wrapper functions (which provide an interface for invoking system calls), libc also provides complex functions that internally invoke multiple system calls (e.g., printf). For these complex libc functions in which a system call is invoked by another libc (wrapper) function, we perform a one-time analysis to identify whether any sensitive arguments are used at the system call invocation site. For example, fork() internally invokes the clone system call with concrete argument values, hardcoded in its function body. In this case, our analysis concludes that no sensitive argument is used at this invocation site, and therefore importing fork() does not lead to the inclusion of CAP_SYS_ADMIN.

For a given system call and sensitive argument combination, if *at least one* of the concrete values passed across *at least one* call site matches one of the *sensitive* values defined in our mapping for that particular argument, Decap assumes that CAP_SYS_ADMIN is required. Similarly, if the concrete value of a given argument cannot be identified by our static analysis at one or more call sites, this again leads to the inclusion of CAP_SYS_ADMIN.

5.3 Capability Enforcement

Setuid programs are launched as privileged processes, and as a result have complete access to all capabilities provided by the kernel. Decap reduces these privileges by first deprivileging the target application entirely by removing its setuid bit, and then granting *only* the capabilities that the program actually requires.

After identifying the required capabilities, Decap generates a capability profile tailored to the program. This profile is then applied to the program by using the setcap tool to modify the extended attributes of its binary file, and adding the capabilities in the "permitted" and "effective" sets of the binary. From that point and on, whenever the deprivileged version of the program is launched, the created process will only acquire the privileges specified in the permitted set.

6 IMPLEMENTATION

We implemented Decap as a Python script that uses a slightly modified version of the open-source Confine [10] and Sysfilter [8] tools to extract the system calls required by a given application. Our modifications are mostly related to interfacing and integrating the two tools into Decap's workflow. As Confine and Sysfilter do not perform any system call argument analysis, we implemented our own limited data flow analysis for extracting the values passed as arguments to the 12 system calls responsible for the CAP_SYS_ADMIN capability. After identifying the required system calls (and their arguments, when needed), Decap uses our pre-generated system call to capability mapping to build and enforce a restrictive capability profile to the target application. The input to Decap is the main application executable and all its library dependencies, and its output is the deprivileged version of the executable.

6.1 System Call Identification

Decap relies on both Confine [10] and Sysfilter [8] to identify the system calls required by a given application. Confine is tailored to containers, and thus Decap integrates only its static binary analysis component, and does not use its dynamic analysis phase, which is used to identify the programs launched in the container.

Similarly to Confine, Decap uses objdump to recursively find all dynamic libraries loaded by a given executable, and then extracts the libc functions imported by the main executable and its libraries. This step generates a list of all libc functions used by the application as a whole. Decap then uses the libc callgraph provided by Confine to map these libc functions to their respective system calls. Note that Confine extracts this callgraph by analyzing the source code of glibc. This is a one-time operation that should be performed on the exact libc version used on the system where the deprivileged applications will run, and is the only analysis that is performed at the source code level—all other libraries and the main executable are analyzed in their binary form.

System calls can also be invoked directly. To handle these cases, Decap uses objdump to analyze the main executable and its libraries and find any direct system call invocation sites. Note that disassembly accuracy is not a concern for this step, as direct system call invocations can be accurately identified by pinpointing the syscall assembly instruction, or calls to the libc syscall() function. In both cases, the value passed as the first argument at the call site holds the system call number. According to the x86-64 calling convention, the first argument is passed through the RDI register for the syscall() function, and through the RAX register for the syscall instruction. In both cases, extracting the actual value passed as the first argument (i.e., the system call number) can be identified in a straightforward way using backwards data flow analysis, because the corresponding register is typically initialized within the few instructions preceding the call site.

Unlike Confine, Sysfilter generates a callgraph for all the libraries an application depends upon (including glibc) by solely analyzing their binaries. Sysfilter then uses these callgraphs to extract the system calls required by the application, and as a result, has more potential for removing unnecessary system calls in comparison with Confine. This is because applications that link with these libraries do not always fully use all functions available in them, and in most cases, just a portion of the libraries' functionalities are used [1, 40]. As a result, Sysfilter can reduce some of the overapproximation of Confine. However, we find that Sysfilter itself suffers from a different kind overapproximations, mainly because it builds the glibc callgraph at the binary level.

As discussed in Section 5.1, Decap mitigates the effects of both tools' overapproximation by considering the intersection of the system call sets extracted by each to infer the capabilities required by the program. Given that both tools produce sound results, the intersection of the two sets of system calls is also sound. Still, to ensure that Decap does not break application functionality, we manually validated the soundness of the intersected system call list by selecting a set of applications from our dataset (including, `chfn`, `chsh`, `passwd`, `gpasswd`, `newgrp`, `su`, `newgidmap`, and `newuidmap`) and manually analyzing their source code to ensure that the system calls removed by Decap are not actually required.

To illustrate the importance of combining both tools, we briefly discuss `chfn` as an example. For this utility, the sets of system calls responsible for `CAP_SYS_ADMIN` identified by Confine and Sysfilter include [`ioctl`, `madvise`, `clone`, `prctl`, `mount`] and [`ioctl`, `madvise`, `clone`, `prctl`, `shmctl`, `msgctl`, `setns`], respectively. The `mount`, `shmctl`, `msgctl`, and `setns` system calls are not in the common set, and thus can be removed. To ensure that removing these system calls is sound, we checked the reachability of each of these system calls. The `mount` system call is invoked by the `selinux_init_load_policy` function in `libselinux.so`, which is loaded by `chfn`. However, this function is not reachable from the source code, and `mount` is not invoked (neither directly nor through its wrapper function) from any other part of the code. Confine considers this system call as needed because it does not apply library specialization, and thus includes all system calls required by any linked library. The `shmctl`, `msgctl`, and `setns` system calls are also not reachable from the source code, and are included by Sysfilter due to its less precise libc callgraph. Based on Decap's argument analysis on the system calls of the intersection, it turns out that `chfn` does not require `CAP_SYS_ADMIN`. If the intersection was not taken, or even if just one of the system calls left out by the intersection was considered as needed, Decap would not have been able to reach this favorable outcome, and `CAP_SYS_ADMIN` would have to be retained.

6.2 System Call Argument Analysis

Decap uses `objdump` to perform argument-level analysis for the 12 system calls that conditionally require `CAP_SYS_ADMIN`. First, Decap performs a one-time analysis of the glibc binary to identify system call invocations, and any potential use of sensitive arguments, by complex functions (e.g., `fork()`, `printf()`). Decap then relies on the results of this analysis to decide whether `CAP_SYS_ADMIN` is required by a given application whenever one of these complex libc functions is imported. To perform this analysis, Decap uses `objdump` to disassemble the glibc binary, and then it identifies *all* the internal direct invocation sites for those 12 system calls. By performing a single-level backwards inter-procedural data flow analysis for the value passed as the critical argument (e.g., the third argument for `clone`) at those invocation sites, Decap attempts to extract the concrete values passed to them, and compares them with the sensitive values identified in our mapping (Section 4.2). Then by using the glibc callgraph provided by Confine [10], Decap tracks from which complex function each invocation site is reachable, and determines whether `CAP_SYS_ADMIN` should be included for that specific complex function. Due to the value is not hardcoded in glibc, Decap was not able to derive concrete values for two of the system calls when invoked from two different complex functions. In these cases, if the respective complex function is imported by the target application or its libraries, we assume that `CAP_SYS_ADMIN` is required.

System calls can also be called through their wrapper functions or invoked directly by the application or its libraries through the libc `syscall()` interface or the `syscall` assembly instruction. Decap uses `objdump` to disassemble the target program and its libraries and extracts the values passed as critical arguments for those 12 system calls by performing single-level backwards inter-procedural data flow analysis at their call sites. After extracting concrete values for these arguments, Decap compares them with the pre-generated list of sensitive argument values and includes the `CAP_SYS_ADMIN` capability if a match is found.

7 EXPERIMENTAL EVALUATION

We experimentally evaluated our prototype implementation of Decap with a set of 201 setuid programs. Each setuid program originally has access to *all* the capabilities supported by a given Linux kernel version. Our evaluation thus mainly focuses on assessing Decap's effectiveness in deprivileging programs by measuring the number of capabilities removed for each program. Given the importance of the `CAP_SYS_ADMIN` capability, we also measured the effectiveness of Decap's argument-level analysis in removing this critical capability. Finally, we evaluated the security benefits of Decap by analyzing past vulnerabilities reported for some of the programs in our data set, and showing how their deprivileged versions mitigate the outcomes of exploitation by limiting the harmful operations an attacker may perform. All our experiments are based on Ubuntu 18.04 with kernel v5.4, which provides 38 capabilities.

7.1 Dataset Collection

The Ubuntu 18.04 distribution provides 75,229 packages in its default repositories. To perform an extensive analysis and evaluate an as large number of setuid programs as possible, we downloaded and attempted to install all these packages in search for setuid programs (each separately in a different container, to prevent any failures due to conflicts). Among all available packages, we were able to successfully install 72,576 of them (96.5%), from which we extracted 201 setuid binary executables. Our automated installation process in some cases failed due to missing dependencies and other conflicts that were not possible to resolve automatically. The extracted programs include popular utilities such as `passwd` and `mount`, and also less known programs such as `at` and `amgtar`.

7.2 System Call to Capability Mapping

We begin with a more in-depth analysis of the system call to capability mapping that we generated using the approach discussed in Section 4. For Linux kernel v5.4, our mapping includes 126 system calls (out of 334 available), which depend on at least one of the 38 available capabilities to perform some form of privileged operation, i.e., capabilities are related to only 38% of the system calls. Figure 4 shows the number of system calls that depend on at least one capability, as a percentage of all capabilities. Just 20 system calls are responsible for half of the available capabilities.

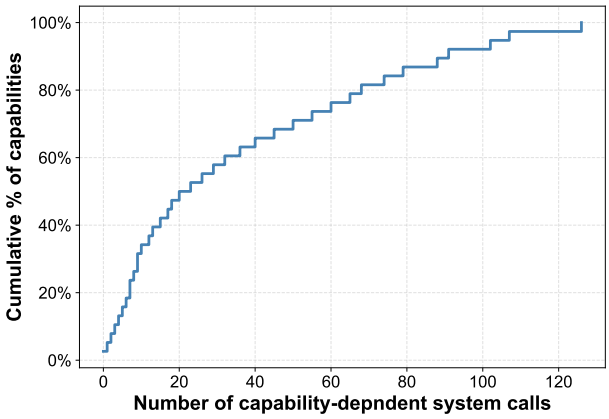

Figure 4: CDF of the 126 capability-dependent system calls out of the 334 provided by Linux kernel v5.4, as a percentage of the 38 capabilities supported by the same kernel. Half of the available capabilities are required by only 20 or fewer system calls.

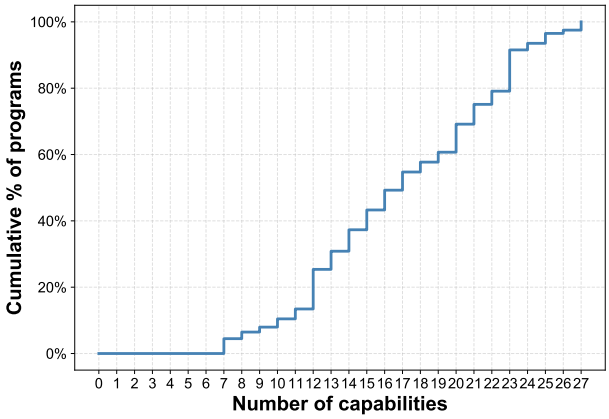

Figure 5: CDF of the number of capabilities required by a given program after deprivileging it with Decap, as a percentage of the 201 setuid programs in our dataset.

As shown in Table 2, the critical `CAP_SYS_ADMIN` capability is required 28 system calls, the highest number among all capabilities. On the other hand, there are 14 capabilities that are required only by a single system call, (e.g., `CAP_SYS_PACCT` is required solely by the `acct` system call). The `ioctl` system call requires the highest number of capabilities (seven) to operate correctly, which is expected given the wide range of operations it can perform. Overall, except `CAP_SYS_ADMIN` and perhaps `CAP_FOWNER`, there is an almost one-to-one conceptual mapping between capabilities and system calls or group of related system calls associated with the same broad type of privileged functionality.

7.3 Deprivileging Programs

We used the 201 setuid programs we collected to evaluate the effectiveness of Decap in removing unnecessary capabilities. Decap

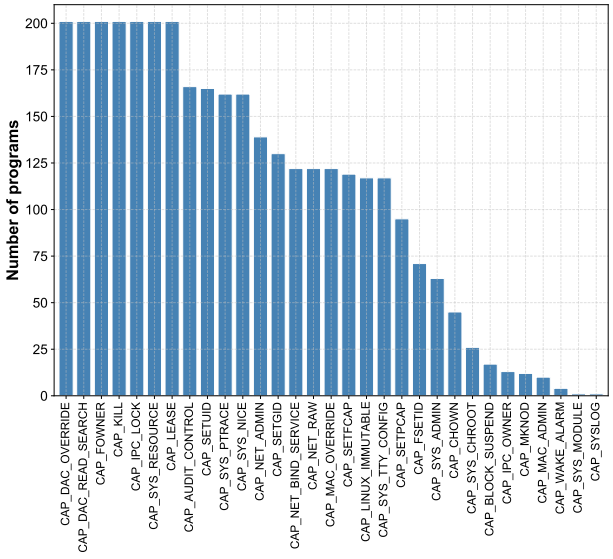

Figure 6: Number of programs requiring a given capability.

first identifies the required system calls of a given program, and then uses our pre-generated mapping to derive the set of required capabilities. As shown in Figure 5, half of the programs require fewer than 16 capabilities to operate correctly. Although capabilities are not completely mutually exclusive, their overlap is small, and thus this result represents a significant deprivileging degree compared to launching these programs as privileged (which would grant them *all* the 38 available capabilities). Across all setuid programs in our dataset, the number of required capabilities per program varies between seven and 27. More specifically, in the best case, Decap reduces the capabilities required by `hashquery` by 82%, and in the worst case it still removes 29% of all capabilities from `polkit-agent-helper-1`.

We further analyzed the specific capabilities required by each program. Our results indicate that some of the capabilities are more frequently required, while others are not used at all. Figure 6 shows the number of setuid programs which require a given capability. We observe that seven capabilities (leftmost bars) are always required by all programs. On the other hand, seven capabilities (not shown in the figure) are not needed by any of the programs in our dataset.

Although some of the seven capabilities that are always included are indeed needed for carrying out certain privileged operations, for some programs they are included as a result of the overapproximation in Decap's analysis. For example, the `open` system call requires `CAP_DAC_OVERRIDE` *only* when a process attempts to open a file for which it does not have access to. Our analysis currently does not resolve the arguments of the `open` system call, and thus conservatively assumes that `open` always requires this capability. Extracting and analyzing the passed values to the `pathname` argument across all `open` call sites, which would determine whether `CAP_DAC_OVERRIDE` is actually needed, is quite challenging from a static analysis perspective. First, `pathname` is a pointer, the target of which may not be statically identifiable through pointer analysis, and second, even if the actual target value can be identified across

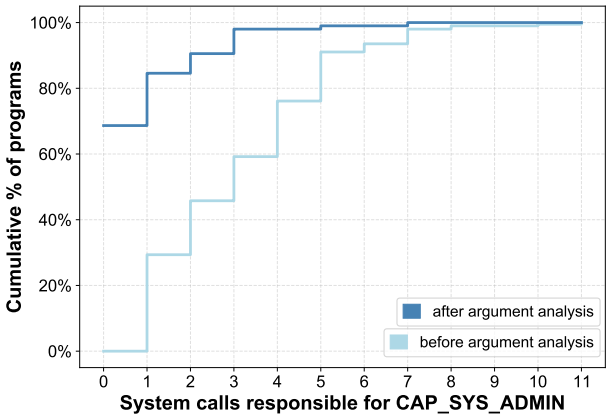

Figure 7: CDF of the number of system calls responsible for `CAP_SYS_ADMIN` before and after argument analysis, as a percentage of the 201 setuid programs in our dataset. Without argument analysis, `CAP_SYS_ADMIN` would have been required by all programs.

all invocation sites, further type analysis must be performed to identify what a given path refers to (e.g., file, directory, symbolic link, terminal device, block device, automount point). We will explore the implementation of more sophisticated system call argument analysis as part of our future work.

As shown in the same figure, Decap successfully removes the security-critical `CAP_SYS_ADMIN` capability from 69% of the setuid programs (it is retained only for 63 out of 201). Considering that this capability is basically equivalent to superuser permissions, this result demonstrates the effectiveness of Decap in meaningfully deprivileging existing programs. We discuss in more detail how Decap's argument-level analysis contributes to this privilege reduction in the next section.

7.4 System Call Argument Analysis

In many cases, the need for `CAP_SYS_ADMIN` depends on the value of certain system call arguments. As discussed in Section 5.2, we have identified 12 system calls that conditionally require `CAP_SYS_ADMIN` (in addition to 16 more that always require it), for which Decap extracts the values of their respective arguments. Given the importance of this capability, we explored further the characteristics of Decap's argument-level analysis, not only to show it effectiveness, but to also gain additional insight on the minority of programs for which `CAP_SYS_ADMIN` cannot be removed.

Figure 7 shows the number of system calls that lead to the inclusion of `CAP_SYS_ADMIN`, before and after applying Decap's argument analysis. Without argument analysis, `CAP_SYS_ADMIN` would not have been removed from *any* of the programs across our dataset. After applying Decap's argument-level analysis, however, `CAP_SYS_ADMIN` can be removed from 69% of the programs. This significant increase shows the importance of analyzing the arguments of those 12 system calls, and reflects the "catch all" nature of `CAP_SYS_ADMIN`.

Table 4: Top-20 most popular programs [50] for which `CAP_SYS_ADMIN` is removed, along with the number of capabilities they require, and the number of CVEs that can lead to privilege escalation (given that our search was not exhaustive, entries without any CVEs do not mean that vulnerabilities were never disclosed for those programs).

Program	Capabilities	CVEs
1) `passwd`	23	53
2) `ping`	16	151
3) `traceroute6.iputils`	14	-
4) `arping`	18	-
5) `at`	16	-
6) `apt-update`	12	-
7) `pmount`	20	2
8) `VirtualBox`	15	221
9) `v4l-conf`	20	1
10) `procmail`	17	3
11) `mailq`	10	1
12) `pppoe`	18	2
13) `uml_net`	16	1
14) `beep`	10	1
15) `blinklight-fixperm`	8	-
16) `sensible-mda`	14	-
17) `schroot`	21	-
18) `lighttpd`	23	1
19) `uncompress.so`	14	-
20) `enlightenment_sys`	20	-

Table 4 shows top-20 most installed programs [50] for which Decap successfully removes `CAP_SYS_ADMIN`, along with the number of capabilities required. These include widely used utilities and server applications, for which the removal of `CAP_SYS_ADMIN` significantly reduces their attack surface. It is important to note that some applications (e.g., Nginx) do not use the `setuid` bit, but depend on being run as root due to their requirements (e.g., binding to a privileged port). Although they can be configured to launch their "worker" processes (which handle the client requests) as a non-privileged processes, an attacker could still attempt to exploit the main process which remains executing as root [32, 44] after successfully gaining access to a worker process.

Figure 8 lists all 63 programs for which `CAP_SYS_ADMIN` cannot be removed, along with the number of system calls responsible for its inclusion for each program. After argument analysis, the majority of the programs require `CAP_SYS_ADMIN` due to only one or two system calls. Among them are programs such as `mount`, `umount`, and `start-suid`, which invoke system calls that unconditionally require `CAP_SYS_ADMIN` (e.g., `mount`).

In a few cases, our single-level inter-procedural backwards analysis cannot extract the values for a critical system call argument, and recursive inter-procedural analysis is required. For example, `ldap_child` invokes the `keyctl` system call, and its second argument specifies whether `CAP_SYS_ADMIN` is needed or not. However, Decap was not able to extract a concrete value for this argument by performing its single-level analysis. We leave extending our analysis to handle these cases as part of our future work.

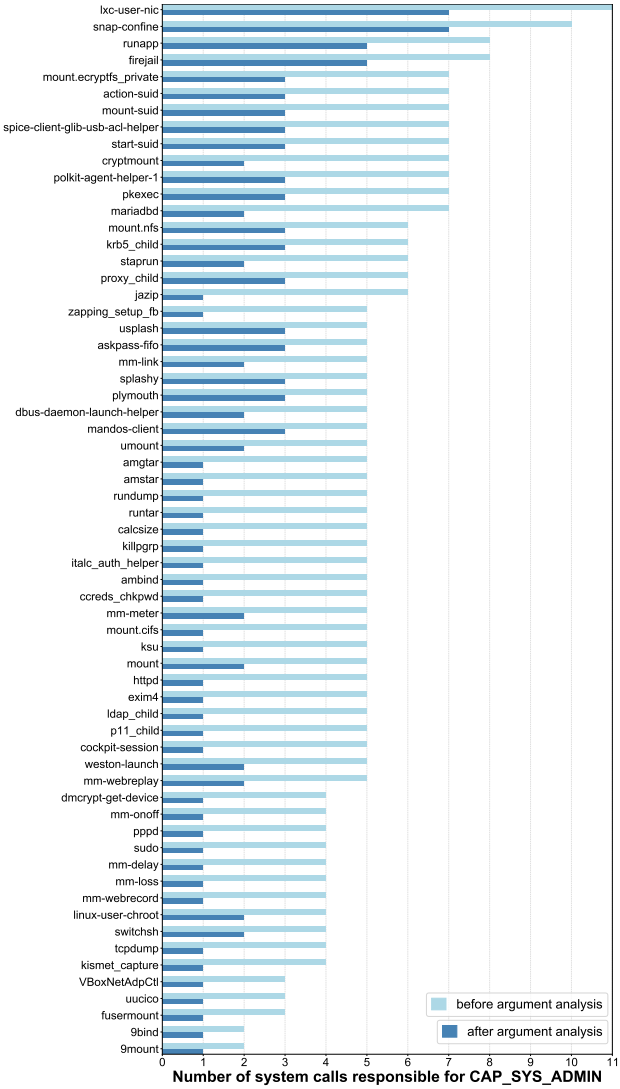

Figure 8: Number of system calls responsible for CAP_SYS_ADMIN before and after argument analysis, for the 63 (out of 201) setuid programs for which Decap does not remove `CAP_SYS_ADMIN`.

7.5 Security Evaluation

To evaluate the security benefit of deprivileging setuid programs, we analyzed previously disclosed vulnerabilities affecting the programs in our dataset. Although Decap does not neutralize these vulnerabilities, it significantly reduces the risks associated with their successful exploitation. Exploiting a vulnerability in a deprivileged program will not provide code execution capability as root (unless `CAP_SYS_ADMIN` has been granted), but instead will constrain the harmful operations an attacker can perform to only those allowed by the program's capabilities.

To collect previously disclosed vulnerabilities, we performed a semi-automated text parsing analysis on more than 230K vulnerability entries from MITRE's CVE list [7]. Through this analysis, we identified 5,018 CVEs affecting at least one of the programs in our dataset. We further refined this set of vulnerabilities by only considering those that can be used to perform privilege escalation (i.e., we excluded CVEs related to denial of service and other low-severity attacks). This further reduced the total number of in-scope vulnerabilities to 869.

We assume that Decap has a meaningful effect against these vulnerabilities only if it can remove `CAP_SYS_ADMIN` for a given program. This is the case for 612 CVEs affecting 39 setuid programs in our dataset. Some of these 39 programs are included in Table 4, along with the number of mitigated vulnerabilities we identified for them. Due to the limitations of our vulnerability gathering approach, we were not able to find CVEs for many of the programs in our dataset. However, this does not necessarily mean that those programs do not have any previously disclosed vulnerabilities, as our search was non-exhaustive.

We discuss a few representative examples to show how Decap can reduce the outcomes of exploitation. CVE-2022-0563 [33] is a privilege escalation vulnerability that affects the `chfn` and `chsh` setuid programs. A local unprivileged user can exploit this vulnerability to gain superuser privileges. Decap removes 15 capabilities from `chfn` and `chsh`, including `CAP_SYS_ADMIN`. Attackers who successfully exploit this vulnerability and gain arbitrary code execution in a deprivileged version of these programs, will be able to perform a much more limited set of privileged operations compared to their setuid versions. CVE-2006-3378 [31] is a vulnerability in the ubiquitous `passwd` program that allows local attackers to gain root privileges. Decap removes `CAP_SYS_ADMIN` for this program as well, constraining the attacker to a more limited set of privileged operations. The `lprm` utility is used to remove jobs from the printer spooling queue, and was affected by a vulnerability [6] that allows local users to gain root privileges. Decap deprivileges this program and revokes its access to 21 capabilities, including `CAP_SYS_ADMIN`.

8 LIMITATIONS AND FUTURE WORK

We generated our system call to capability mapping by thoroughly investigating different sources and performing extensive experiments. Although we did not observe any failures after applying Decap to the 201 programs in our data set, our current mapping may still be incomplete. We plan to maintain a publicly available version of the mapping and welcome community contributions as part of our future effort to continuously improving and refining it.

A complete mapping could be obtained from the kernel's callgraph, but such a mapping would suffer from overapproximation due to the inherent imprecision of callgraph extraction. More importantly, although this mapping would be complete, it would not necessarily be correct. Such a mapping would represent all permission checks as currently implemented in the kernel, and given that checks are often added in an ad-hoc manner, this would result in inconsistencies. Zhang et al. [54] have shown that there are missing, inconsistent, and redundant permission checks in the Linux kernel. Therefore, a complete and sound mapping will be possible only after these permission check issues are resolved in the Linux kernel. We leave the investigation of automatically deriving the mapping using the kernel's callgraph as part of our future work.

Given the importance of CAP_SYS_ADMIN, we have currently applied our argument-level analysis only to system calls that may lead to its inclusion. The same analysis could be applied to other capabilities and system call arguments to increase the precision of our mapping, and consequently the number of removed capabilities. For example, the mapping of the sendto system call to CAP_AUDIT_WRITE is only valid for certain types of file descriptors, and most applications that use sendto do not actually require this capability. Another example is bind, which only requires CAP_NET_BIND_SERVICE when binding a socket to a privileged port. Currently, our mapping does not consider the addr argument and always includes this capability when bind is used. More importantly, besides CAP_SYS_ADMIN, a few other capabilities can be considered "root-equivalent" [49] and should be removed whenever possible. We plan to continue analyzing such security-critical capabilities and refining our mapping with more argument dependencies for more system calls as part of our future work. As discussed in Section 7.3, more sophisticated static code analysis will be required for strings and other complex types of arguments.

Although Decap's argument-level analysis reduces the over-approximation related to CAP_SYS_ADMIN, our single-level inter-procedural analysis cannot always resolve all sensitive argument values across all call sites, as discussed in Section 7.4. We plan to extend our argument extraction process to rely on complete inter-procedural analysis to improve the coverage of sensitive argument value identification.

Decap operates at the binary level, except Confine's dependncy on the source code of libc. This is an acceptable requirement, as it is a one-time operation, and the source code of a given libc version is easy to obtain (in contrast to application source code).

Running both Confine and Sysfilter results in significant duplicate work. Ideally, an optimized implementation would combine Confine's more precise libc callgraph extraction with Sysfilter's library specialization approach, and run only those two components. We leave this integration optimization as part of our future work.

9 RELATED WORK

9.1 Attack Surface Reduction

Attack surface reduction through software debloating and specialization removes code and features from applications with the aim to neutralize undiscovered vulnerabilities, and complicate the exploitation of any remaining vulnerabilities. Several works perform software debloating by identifying unnecessary code and features through static analysis [1, 8, 10, 11, 25, 29, 30, 40], dynamic analysis [9, 15, 38], or a combination of both [10, 39]. Although the goal of these works is mainly to remove unnecessary parts of an application, some also filter unnecessary system calls [8, 10, 11], which in essence deprivileges the application by preventing it from invoking potentially dangerous system calls. Decap is complementary to system call filtering approaches (it actually relies on Confine [10] and Sysfilter [8]), as it further restricts the privileged operations that the remaining system calls may perform.

9.2 Privilege Reduction

Most previous works on privilege reduction focus on the implementation of new techniques in the kernel or applications to either remove the need for setuid programs, or reduce a program's privileges during its execution. Given that Linux access controls have been applied to the kernel in an ad-hoc manner, PeX [54] performs static analysis on the kernel to identify missing, inconsistent, or redundant permission checks.

9.2.1 Privilege Separation. Decomposing programs into privileged and non-privileged components has been an extensively explored approach [2, 23, 34, 43, 46, 52]. Similarly to Decap, the goal of these works is to restrict process privileges so that an attacker does not gain superuser access after successfully exploiting a vulnerability in the program. Shinagawa and Kono [47] and Provos et al. [37] propose to (manually) refactor setuid programs into two cooperating privileged and non-privileged processes. Privtrans [4] is a framework for automatically applying privilege separation. It requires annotations by the programmer and uses static data flow analysis to propagate the privileged operations across the entire program. Capsicum [53] implements fine-grained privilege tokens in the Linux kernel, which provides programs with a means to decompose their privilege requirements into different sections. Capweave [14] instruments programs to use the Capsicum primitives for enforcing a declarative policy that specifies when and which tokens a program should hold during its execution.

9.2.2 System Call Policies. An alternative to using setuid programs is to define and enforce system call policies that specify whether access to a resource should be permitted. Systrace [36] deprivileges setuid programs and relies on modifications in the kernel to permit privileged operations based on predefined system call policies. The kernel and a user-space daemon use these policies to determine whether a process should be permitted to perform a privileged operation. Protego [17] claims that the main reason for requiring setuid programs is related to the privileged operations performed by eight system calls, for which an equivalent policy can be enforced in the kernel. However, to apply these policies, multiple modifications need to be applied to the kernel, which limits the applicability of the approach. PoLPer [18] also uses policies to restrict system call invocations. However, the policies are built by executing the program and profiling the system calls based on their normal usage. Through this profile, PoLPer can identify anomalous behavior in a process' system call invocations. Rajagopalan et al. [41] propose a technique for enforcing policies by adding message authentication codes to the respective system calls.

9.2.3 Removing Unnecessary Capabilities. AutoPriv [16] provides a compiler-level analysis that helps programmers use capabilities more efficiently. Given a program that has been refactored to raise and lower privileges before and after system calls that use privileges, it analyzes whether a privilege is required at each basic block, and removes it when no longer required. Although this work removes capabilities after their usage, it is challenging to refactor a program and specify which capabilities are required by each system call. PrivAnalyzer [5] leverages AutoPriv to find the required capabilities at each program point, and provides a new compiler pass (ChronoPriv) that records the number of instructions executed with a given privilege. It also includes ROSA, a bounded model checker to determine the damage at each program point after an attacker exploits the program and abuses its privileges. MiniCon [19]

represents an eBPF-based capability enforcement system using Seccomp filters [20]. It uses dynamic analysis to identify a program's required capabilities by iteratively running the program, extracting a required capability (due to a permission check failure), and adding it to the permitted set, until the program runs successfully. As a purely dynamic analysis approach, it suffers from code coverage issues, as any capability required solely by code paths not exercised during training will be missed.

Most of these previous works require some form of policy generation (usually provided by the user) or programmer annotations, which limits their applicability. Decap, on the other hand, performs static analysis and extracts a superset of the required capabilities through a streamlined process, without any manual intervention. Manual analysis is only required for building the initial system call to capability mapping, which is one-time effort.

10 CONCLUSION

Despite the clear benefits of Linux capabilities for reducing the privileges of setuid and other over-privileged programs, their use remains scarce. By providing a clear mapping between system calls related to privileged operations and the corresponding capabilities they require, our work aims to facilitate the use of capabilities when developing new applications, and reduce the risk of existing setuid programs. Decap automatically deprivileges programs using static analysis to extract the system calls they may use (and for some of them, the values of certain arguments passed at their invocation sites), and then to enforce a minimal set of capabilities that are necessary for their correct operation. The results of our experimental evaluation demonstrate that despite the limited nature of our static analysis and the resulting over-approximation, Decap is quite effective in meaningfully deprivileging existing setuid programs. As part of our future work, we plan to improve Decap's accuracy by implementing complete inter-procedural argument analysis for system calls that conditionally depend on `CAP_SYS_ADMIN`, and introduce runtime instrumentation to extract sensitive argument values for additional system calls and capabilities.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. We also thank Eugene Stark for his valuable comments on an earlier draft of this paper. This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, and the National Science Foundation (NSF) through awards CNS-1749895 and CNS-2104148.

A APPENDIX

We analyzed the privilege characteristics (use of capabilities, use of setuid) of Linux programs included in different Ubuntu distribution versions since 2012. Among more than 59K software packages in the latest (at the time of writing) Ubuntu 21.10 release, we found just 29 capability-aware programs, listed in Table 5.

REFERENCES

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*. 70–83.

[2] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged app sandboxing for stock Android. In *Proceedings of the 24th USENIX Security Symposium*. 691–706.

[3] Daniel P. Bovet and Marco Cesati. 2002. *Understanding the Linux Kernel.* O'Reilly.

[4] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium*.

[5] John Criswell, Jie Zhou, Spyridoula Gravani, and Xiaoyu Hu. 2019. PrivAnalyzer: Measuring the Efficacy of Linux Privilege Use. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 593–604.

[6] cvedetails.com. 2003. Vulnerability Details CVE-2003-0144. https://www.cvedetails.com/cve/CVE-2003-0144/

[7] cve.mitre.org. 2022. CVE List. https://cve.mitre.org/

[8] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. Sysfilter: Automated System Call Filtering for Commodity Software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*.

[9] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-flow Trimming. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*.

[10] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*.

[11] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *Proceedings of the 29th USENIX Security Symposium*.

[12] Brendan Gregg. 2016. *Linux bcc Tracing Security Capabilities.* https://www.brendangregg.com/blog/2016-10-01/linux-bcc-security-capabilities.html

[13] Heino Sass Hallik. 2019. Linux privilege Escalation using the SUID Bit. https://materials.rangeforce.com/tutorial/2019/11/07/Linux-PrivEsc-SUID-Bit/

[14] William R. Harris, Somesh Jha, Thomas Reps, Jonathan Anderson, and Robert N.M. Watson. 2013. Declarative, Temporal, and Practical Programming with Capabilities. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[15] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*.

[16] Xiaoyu Hu, Jie Zhou, Spyridoula Gravani, and John Criswell. 2018. Transforming Code to Drop Dead Privileges. In *Proceedings of the IEEE Cybersecurity Development (SecDev)*. 450–52.

[17] Bhushan Jain, Chia-Che Tsai, Jitin John, and Donald E. Porter. 2014. Practical Techniques to Obviate Setuid-to-Root Binaries. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*.

[18] Yuseok Jeon, Junghwan Rhee, Chung Hwan Kim, Zhichun Li, Mathias Payer, Byoungyoung Lee, and Zhenyu Wu. 2019. PoLPer: Process-Aware Restriction of Over-Privileged Setuid Calls in Legacy Applications. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY)*.

[19] Haney Kang, Jinwoo Kim, and Seungwon Shin. 2021. MiniCon: Automatic Enforcement of a Minimal Capability Set for Security-Enhanced Containers. In *Proceedings of the IEEE International IoT, Electronics and Mechatronics Conference (IEMTRONICS)*.

[20] kernel.org. 2012. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html

[21] Michael Kerrisk. 2010. *The Linux Programming Interface.* No Starch Press.

[22] Michael Kerrisk. 2012. *CAP_SYS_ADMIN: the new root.* https://lwn.net/Articles/486306/

[23] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications.. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 273–284.

[24] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[25] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security (EuroSec)*.

[26] Vickie Li. 2020. *Becoming Root Through An SUID Executable.* https://vickieli.medium.com/becoming-root-through-an-suid-executable-47473173a6ec

[27] man7.org. 1999. Capabilities(7) - Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/capabilities.7.html.

[28] Alois Micard. 2020. Privilege escalation using setuid. https://blog.creekorful.org/2020/09/setuid-privilege-escalation/

[29] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*.

[30] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive Function Specialization against Code Reuse Attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*.

Table 5: Out of 59,021 packages in Ubuntu 21.10, there were only 29 capability-aware programs, while earlier versions have even fewer. (✗ : setuid, ✓ : capability-aware, ○ : neither setuid nor capability-aware, – : program not available)

Program	Ubuntu 12.04 (April 2012)	Ubuntu 14.04 (April 2014)	Ubuntu 16.04 (April 2016)	Ubuntu 18.04 (April 2018)	Ubuntu 20.04 (April 2020)	Ubuntu 21.10 (October 2021)
ping	✗	✗	✗	✗	✓	✓
fping	✗	✗	✓	✓	✓	✓
arping	✗	✓	✓	✗	✓	✓
oping	✗	✗	✗	✗	✗	✓
noping	✗	✗	✗	✗	✗	✓
traceroute6.iputils	✗	✗	✓	✗	✓	✓
gnome-keyring-daemon	✓	✓	✓	✓	✓	✓
slock	✗	○	✓	✓	✓	✓
check_dhcp	○	○	✓	✓	✓	✓
check_icmp	○	○	✓	✓	✓	✓
spine	○	○	○	○	✓	✓
hercifc	○	○	○	○	○	✓
pinger	–	✗	✗	✗	✓	✓
newpid	–	○	✓	✓	✓	✓
pollen	–	✓	✓	✓	✓	✓
gst-ptp-helper	–	–	✓	✓	✓	✓
apps.plugin	–	–	–	✓	✓	✓
newrole	–	–	–	✓	✓	✓
ss-local	–	–	–	✓	✓	✓
ss-redir	–	–	–	✓	✓	✓
ss-server	–	–	–	✓	✓	✓
ss-tunnel	–	–	–	✓	✓	✓
obfs-server	–	–	–	✓	✓	✓
stenotype	–	–	–	✓	✓	✓
dublin-traceroute	–	–	–	✓	✓	✓
seunshare	–	–	–	✓	✓	✓
json2file-go	–	–	–	–	–	✓
ukui-system-monitor	–	–	–	–	–	✓
prometheus-homeplug-exporter	–	–	–	–	–	✓

[31] nvd.nist.gov. 2006. Vulnerability Details CVE-2006-3378. https://nvd.nist.gov/vuln/detail/CVE-2006-3378

[32] nvd.nist.gov. 2019. Vulnerability Details CVE-2019-0211. https://nvd.nist.gov/vuln/detail/CVE-2019-0211

[33] nvd.nist.gov. 2022. CVE-2022-0563 - chfn and chsh. https://nvd.nist.gov/vuln/detail/CVE-2022-0563

[34] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 71–72.

[35] IO Visor Project. 2015. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc

[36] Niels Provos. 2003. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*.

[37] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*.

[38] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium*.

[39] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 461–476.

[40] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium*. 869–886.

[41] M. Rajagopalan, M.A. Hiltunen, T. Jim, and R.D. Schlichting. 2006. System Call Monitoring Using Authenticated System Calls. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 216–229.

[42] Dennis M. Ritchie. 1979. Protection of Data File Contents. US Patent 4,135,240.

[43] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, Andre DeHon, and Nathan Dautenhahn. 2021. MSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 296–311.

[44] Bob Rudis. 2019. Apache Httpd Server Privilege Escalation (CVE-2019-0211): What You Need to Know. https://www.rapid7.com/blog/post/2019/04/03/apache-http-server-privilege-escalation-cve-2019-0211-what-you-need-to-know/

[45] J.H. Saltzer and M.D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[46] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. 2016. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[47] Takahiro Shinagawa and Kenji Kono. 2004. Implementing a secure setuid program. *Parallel and Distributed Computing and Networks* (2004).

[48] Gurkirat Singh. 2021. Exploiting SUID Binaries to Get Root User Shell. https://tbhaxor.com/exploiting-suid-binaries-to-get-root-user-shell/

[49] Brad Spengler. 2011. False Boundaries and Arbitrary Code Execution. https://grsecurity.net/false_boundaries_and_arbitrary_code_execution.

[50] The Ubuntu Web Team. 2022. Ubuntu Popularity Contest. https://popcon.ubuntu.com/

[51] Michael Torres. 2018. Linux Privilege Escalation - SetUID. https://micrictor.github.io/Exploiting-Setuid-Programs/

[52] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. 2022. PrivGuard: Privacy Regulation Compliance Made Easier. In *Proceedings of the 31st USENIX Security Symposium*.

[53] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. 2010. Capsicum: Practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*.

[54] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. 2019. PeX: A Permission Check Analysis Framework for Linux Kernel. In *Proceedings of the 28th USENIX Security Symposium*. 1205–1220.