PoLPer: Process-Aware Restriction of Over-Privileged Setuid Calls in Legacy Applications

Yuseok Jeon
Purdue
Junghwan Rhee
NEC Laboratories America
Chung Hwan Kim
NEC Laboratories America
Zhichun Li
NEC Laboratories America
Mathias Payer
EPFL and Purdue
Byoungyoung Lee
Seoul National and Purdue
Zhenyu Wu
NEC Laboratories America

ABSTRACT

setuid system calls enable critical functions such as user authentications and modular privileged components. Such operations must only be executed after careful validation. However, current systems do not perform rigorous checks, allowing exploitation of privileges through memory corruption vulnerabilities in privileged programs. As a solution, understanding which setuid system calls can be invoked in what context of a process allows precise enforcement of least privileges. We propose a novel comprehensive method to systematically extract and enforce least privilege of setuid system calls to prevent misuse. Our approach learns the required process contexts of setuid system calls along multiple dimensions: process hierarchy, call stack, and parameter in a process-aware way. Every setuid system call is then restricted to the per-process context by our kernel-level context enforcer. Previous approaches without process-awareness are too coarse-grained to control setuid system calls, resulting in over-privilege. Our method reduces available privileges even for identical code depending on whether it is run by a parent or a child process. We present our prototype called PoLPer which systematically discovers only required setuid system calls and effectively prevents real-world exploits targeting vulnerabilities of the setuid family of system calls in popular desktop and server software at near zero overhead.

CCS CONCEPTS

• Security and privacy → Systems security: Software and application security;

KEYWORDS

Setuid system calls, Least Privilege Principle, Process hierarchy

ACM Reference Format:

1 INTRODUCTION

The setuid family of system calls\(^1\) is a well-established mechanism in major operating systems to manage privileges in applications [11]. The setuid system calls enable critical security functions such as user authentication and modular privileged components. The code invoking setuid calls, namely privilege sensitive code, provides a conceptual security gateway for privileged operations. However, if privilege sensitive code is misused or fails to perform rigorous checks, this code can lead to a disastrous system breach by allowing unintended privileges (e.g., a root shell spawned by an attacker). As such, privilege sensitive code essentially forms the foundational principle of least privilege [48].

In an ideal deployment with the principle of least privilege (PoLP), a certain entity is granted with a privilege only when needed, and de-privileged otherwise. Specifically, the system must be compartmentalized so that a privileged operation can only be executed after careful checking of the context. On one hand, compartmentalization [21, 23] reduces the amount of code that may execute setuid calls. And, on the other hand, it enables a strict interface on how privileged operations can be reached. For instance, Qmail [5] uses separate modules under separate user IDs where each ID has only limited access to a subset of resources. The highest privilege (e.g., root) is contained in a very small restricted module to prevent its misuse. As another example, secure computing mode (seccomp) [49] is a security facility in the Linux kernel allowing a process to make a one-way transition in a secure state; for instance, a system call is restricted on certain parameters or entirely after a state transition.

In addition, the principle of least privilege requires different modules to interact through clearly defined channels. That is, a module may only request services from other modules using a well-defined API. If the principle of least privilege is enforced correctly, vulnerable modules are unprivileged (e.g., rendering engines in a browser or audio/video decoding modules in media players). Thus, adversaries must not only hijack the control flow or manipulate the data flow but also launch confused deputy attacks [22] to circumvent least privilege. In particular, attackers must launch confused deputy attacks to confuse the trusted module through the exposed API, such that their hijacked unprivileged context can be escalated to a higher privilege context which is suitable to perform malicious actions.

Although the principle of least privilege raises the security bar significantly, it is challenging to enforce on legacy applications that

\(^1\)The setuid family includes system calls that set user ID (UID) and group ID (GID) in Unix-like operating systems, such as setuid, seteuid, setgid, and setegid. We also use setuid calls to refer to these system calls herein.
We present an automated approach to extract process-aware setuid contexts: Multiple processes and threads share the same code for execution. However, processes may have different requirements of setuid calls depending on their logic. It is crucial to divide program’s execution contexts of a whole program level into a finer-grained process level to prevent over-privilege of setuid calls. We solve this problem by learning and run-time monitoring with a process hierarchy context.

**Automated extraction of process-aware setuid contexts:** We present an automated approach to extract process-aware contexts of setuid calls from a program using static analysis and dynamic training. Data context and call contexts of setuid calls are indexed with a process hierarchy context to individualize each process behavior.

**Efficient and practical hardening of setuid calls using restriction on process context:** We propose a practical approach to harden data context and call context of setuid calls individualized per process. It tightens previously over-provisioned privileges due to the failure of distinction on processes and effectively prevents security exploits and bugs with minimal overhead. In the benchmarks of multiple commonly used client and server software, the performance overhead of our system is under 0.54%.

---

1. PoLPer represents Principle of Least Privilege Enforcer.
2 THREAT MODEL

We assume a strong adversary who can compromise any program in user space, including privileged programs with setuid calls, using a non-administrative user account. For example, the adversary has a login to a user account using a stolen password and hijacks the control flow of a privileged program by exploiting software vulnerabilities. The adversary can also manipulate the privileged program’s code and data in either disk or process memory.

However, based on the wide availability of code injection prevention [37, 53] and file-based [31] integrity checkers, naive manipulation in file or code injection would be easily detectable. Thus we assume that the binary file and code integrity of the privileged program, user-level libraries, system libraries, the operating system, and PoLPer can be verified. The operating system and the run-time enforcer of PoLPer inside the kernel space are part of the trusted computing base, and thus cannot be altered by the adversary.

In our usage model, a user is either a software developer who can provide a default policy for the program or a system administrator who can customize the policy based on an environmental context on the deployed system. This model is reasonable as seen in major security tools like AppArmor or SELinux that take a similar approach to deploy policies for various software. Therefore, we assume that PoLPer has access to the program binary (source code is optional) along with its workload regarding setuid calls in normal usages before its deployment and any adversarial attempt. Also if its source code is optionally available, PoLPer further improves the precision of policies. Because an administrator handles the program before its installation, this prior access is reasonable. This paper focuses on software written in C/C++ and compiled to native binary code.

3 DESIGN OF POLPER

3.1 Architecture

In this section, we present the architecture of PoLPer. Our goal is to enforce only required privileges for the setuid calls of application code, which are essentially the setuid calls identified in the program code and specified by developers. There are multiple aspects in how this family of system calls is used. First, there can be more than one process involved in program execution. In many server programs, the executed part of code and corresponding roles of processes are different even though they share the exact same code image. Differentiating such roles is essential to achieve least privileges (i.e., we prohibit a setuid call and a potential exploitation to a worker process which does not require a setuid call). We achieve this aspect with our novel process hierarchy context. Second, setuid family calls have different behaviors and risks depending on the input parameters (e.g., a nobody account vs. the root account) and process. Therefore, the values of parameters passed to privilege operations must be carefully inspected and restricted based on the required values profiled for each process. We extract this aspect, called process data context, from the program code with static analysis and dynamic training. Lastly, setuid calls should be inspected in a fine-grained way regarding which privilege sensitive code can invoke them in what specific call contexts in each process. Therefore, we learn and enforce the detailed call stack patterns of setuid system calls given each process and this aspect is called process call context.

Figure 2 presents the architecture of PoLPer regarding how it learns and enforces data and call contexts indexed by each process’ context. In the first process context analysis stage, given a program, its code and run-time execution is analyzed using static analysis and dynamic training to extract the process contexts of setuid calls. In the second stage, the process context enforcement, the extracted contexts are loaded into the OS kernel. Whenever the program calls any setuid system calls, its process hierarchy, data, and call contexts are cross-checked if the call complies with the extracted contexts. Any violating call is detected and prevented as a misuse of setuid system calls.

In this paper, we focus on setuid family calls as an instance of privilege operations because of their prevalence in major legacy software and corresponding attacks. We note that our system can be easily extended to apply our techniques to other operations such as capabilities as future work.

3.2 Extraction of Process Hierarchy Context

Process Behavior Role: Programs that leverage multiple processes and/or threads often leverage different units of execution to decompose functionalities. Each class of processes/threads executes a specific behavior, restricted to a subset of the code although the entire code image is shared across all processes and threads. For example, popular server programs such as sshd and apache use child processes as workers while the parent process manages a pool of workers and distributes the workload. To describe such different characteristics of a set of processes/threads inside the same program, we use the term process behavior role.

Such different behavior roles imply that the capability to run setuid calls are over-provisioned due to the inability to apply...
individualized policies for processes. Unrestricted capabilities to run setuid calls pose the risk that such code can be exploited through a vulnerability.

**Infering Process Behavior Role with a Process Hierarchy Context:** We infer a process’ behavior role using the hierarchical position of it relative to other processes. As process creation system calls (e.g., fork/clone) are invoked, the processes and threads inside a program form parent and child relationships, resulting in a process tree where the root is the first process of the program (Figure 3). We use the hierarchical distance between the process and the root process in the tree to infer its behavior role. We call this distance the process hierarchy context.

Our key intuition is that a process usually exhibits a different role depending on where it is located within a process hierarchy tree. Thus, we differentiate the process’ behavior role using the relative distance from the root (i.e., depth), which becomes our metric to systematically infer and capture a process behavior role.

In Figure 3 the root of a program is the oldest parent process whose parent’s program image is different from its program image. This may not be determined at the program creation time because, in many OSes (e.g., Unix variants like Linux), the first process of a program is a child clone of a program invoker. Later, its program image is replaced using another system call such as exec. Thus our algorithm determines this distance or depth at a program image replacement call (e.g., exec) and caches its result for run-time usage. The depth increases when a process forks/clones a child process/thread.

PoLPer uses process states in an operating system kernel. A process node represents a process state (e.g., a task_struct in Linux). When a setuid call is invoked, PoLPer determines the current process node (i.e., the current process in OS). Given a process node, our algorithm follows the reference for its parent until it reaches the root process of the program, whose parent has a different program image and has the depth 0. Then its depth is calculated as the distance of a traversal to the root process. If there is more than one role in the same depth, this scheme determines them as one combined role.

Given our evaluation, we found that this scheme is effective to distinguish a process behavior role at minimal run-time cost. However, in complex software, it is possible in some rare cases that multiple processes with the same depth can play different roles. Our current policy over-approximates this situation as combined process behavior. PoLPer can improve the precision by using additional program contexts, such as the call stacks of the origin process during fork and the child.

### 3.3 Extraction of Process Data Context

setuid system calls use parameters to represent various information involved to change a privilege in each process context. Given the logic and the context where a program makes a transition of privilege from one to another, the parameters will have corresponding particular values.

The patterns of parameters are learned with the process hierarchy context to differentiate unique parameter behavior of each process role. We use a hybrid approach combining dynamic and static analyses for the advantages of both approaches.

**Learning Process Data Context using Dynamic Training:** Our dynamic and static analyses have different contributions to capturing data context. The dynamic analysis captures crucial information including process hierarchy context and concrete values from outside of a program such as the values from files, network, or OS. The static analysis can further contribute to the patterns by discovering possible data flows systematically.

The same environment and input for setuid system calls are used for dynamic and static analyses. Our dynamic analysis starts with the program binary under the account assumed in the execution environment. When one of setuid family system call is invoked, it traps to our analysis module. Our module records concrete parameter values along with the process hierarchy context and call context at the setuid call.

**Enhancing Data Context using Static Analysis:** If source code is available, static analysis systematically discovers over-approximated potential data contexts. Combined with dynamic results, it further improves learning which will be presented in Section 5.2. Our analysis discovers data contexts with the following details. After the analysis, the result is associated with the profile from dynamic training.

- **Constant Values:** We found many instances of constant values in the setuid calls of evaluated programs where the parameter values are exactly determined in the code. For instance, some code always invokes a setuid call to set the root privilege. Then our analyses find the constant value corresponding to the root. These invariant cases are effective for tightly restricting the parameter.

- **Symbolic Values:** We also found that some parameters can be determined systematically by certain API calls. For instance, getuid system API obtains the current user ID from OS. A frequent use case invokes this API first, followed by a setuid call based on the returned value. Essentially there is a strict data flow from the source using a symbolic value, the current user ID, to the setuid call. Thus when this program is running under a certain account, we can have a specific value for the parameter given the context of the account. We first determine this case from static analysis, and then it is checked by accessing the user ID in the OS state (e.g., task_struct).

- **External Values:** There are other cases where these parameters are derived from various other constants or variables, an input, or OS. We use dynamic training to determine concrete values given in our execution environment. Technically it would be feasible to determine the source of such variables using advanced program analysis techniques (e.g., system level information tracking) as a future improvement.

We use the UniSan framework [36] as our basis of context-sensitive data flow analysis and extended various functionalities such as the coverage of global scope. The following steps show a high-level description on how our Algorithm 1 works.

- **Line 1:** The input for this analysis is a set of setuid system calls and program code. We select 8 setuid system calls in Linux in this work.

- **Line 2-5:** Static analysis first determines the list of call sites for a setuid call p, which is one of the supported set of setuid system calls shown as P. For each call site s of p, data flow analysis is performed.
Algorithm 1 Static data context extraction.

```plaintext
C : Code
P : Set of setuid family system calls
F : Profile of data contexts

function EXTRACTDC(C, P)
    FindReachableDataFlow(V, r, success, matches = False)
    if success == True then
        out = {}
    for p ∈ P do
        if m ∈ matches do
            [type, val or F] = m
            switch type do
                case Const
                    out = out ∪ (Static, val)
                case Function
                    out = out ∪ (Dynamic, return(F))
        res = [[]]
        for r in R do
            if r is a constant then
                res[r] = (Static, call(r))
            else
                res[r] = (Dynamic, return(F))
        end for
        end function
    end function
end function
```

- Line 9-11: For each privilege sensitive code `s` calling a setuid function `p`, a list of its parameters in the invocation is obtained.
- Line 13-15: If a parameter is a constant value, the analysis stops and records the value.
- Line 16-17: If a parameter is a return value from a function `F` (e.g., `getuid()`), this case is handled as a symbolic variable.
- Line 18-20: If a parameter `r` is a variable, an inter-procedural data flow analysis is performed to find the source of the value. To find the sources, it first obtains a set of all variables and constants, denoted as `V`. And then, it checks whether any item `v` reaches `r`.
- Line 21-33: When one or more constant or variable is used for the parameter `r`, possible values are stored in the profile. If the source that `r`’s value is from is a constant (Line 26-28), the corresponding value is stored. If it is returned from a function, its value is determined in dynamic training (Line 29-31).
- Line 34-35: If no variable or constant can reach a parameter `r`, it could be due to limitation of static analysis. Then its concrete value is determined in dynamic training.

### 3.4 Extraction of Process Call Context
A call stack at a setuid system call can identify a specific code location and call context how the setuid call is made. By being recorded together with the process hierarchy context this information can represent specific patterns of setuid calls in each process role.

We leverage dynamic analysis for high accuracy call context (i.e., call stack). Static analysis may be able to determine the call context as well. However, if the call stack involves dynamically linked library code, which may have internal function calls within the library, dynamic analysis provides higher accuracy.

### 3.5 Run-time Enforcement of Process Contexts
This section presents how to detect execution anomalies based on the process contexts of setuid family system calls.

When a program executes, any invocation of a setuid call is captured by the system call interposition layer. The process hierarchy context and process contexts are collected and compared with the profile that was previously extracted from the program to determine whether they are part of known behavior.

Any unknown behavior is denied returning a failure code and logging of this violation incident (to be exported to a system administrator). Upon a denied attempt, this detailed log showing program internal states will help the developer and an administrator to understand how (process hierarchy context, call context) and with what parameters (data context) this application was about to be exploited but protected.

### 4 IMPLEMENTATION
In this section, we present the technical details in the implementation of PoLPer. There are two major components: process context analyzer and enforcer.

**Process Context Analyzer:** Our static data flow analysis is implemented by extending UniSan [36], which is based on the LLVM framework [35]. UniSan is designed for eliminating information leak vulnerabilities in OS kernels caused by uninitialized data reads. We use its functionalities to track data flow for identifying the data context in this paper. Our major extensions are three-fold: (1) checking reachability from an object allocation site to privilege operation calls, (2) collecting possible store values (e.g., `foo = getuid()`) during tracking data flow, and (3) tracking global objects (UniSan only track heap and stack objects) since the parameters may have values derived from global objects. Dynamic training is performed by using process context enforcer to be explained next with a permissive learning mode which records process contexts on setuid system calls.

**Process Context Enforcer:** This component residing in the OS kernel intercepts an invocation of a setuid family system call and inspects the run-time states. We implemented this component using Kprobes [29], a kernel-based probing mechanism which can dynamically hook any kernel routine. Note that Kprobes avoids any instrumentation or modification inside software binary or in the user space. Thus it provides a near-native speed of execution of the program code without intrusively instrumenting the application.

PoLPer inserts Kprobes hooks on the entry points of setuid family system calls. When any setuid system call is invoked, PoLPer takes a control through Kprobes and extracts process hierarchy, data, and call contexts. When this module is used for dynamic analysis in a learning mode, the extracted contexts are temporarily stored in kernel heap, then it is stored in a file as a profile for the program. When it is used for the enforcement, the extracted run-time contexts are verified whether they comply with the profile.

When PoLPer inspects the process hierarchy, data parameters, and the call stack on setuid calls, we applied multiple optimization code to keep the run-time overhead minimal.

### 5 EVALUATION
In this section, we evaluate PoLPer in multiple aspects: (1) detection of real-world exploits, (2) multi-context extraction, (3) performance evaluation, and (4) case studies.

**Experimental Setting:** All evaluations were performed on Ubuntu Linux 14.04.5 LTS with a quad-core 3.40GHz CPU (Intel i7-6700), 1TB HDD, and 16GB RAM. Note that we chose the old OS version
to evaluate real exploit case studies in situ. Our design and implementation have no restriction to support other Linux operating system distributions.

Evaluation Target Programs: We have tested PoLPer with the following real-world desktop and server programs: Ping, Sudo, AccountsService, Upstart, Telnet, Shadow, SSH, Wireshark, Apache, and Nginx. These are common selected applications which use setuid system calls with a varying size and complexity from small utility programs (e.g., Ping) to larger server programs like Apache. As described in our threat model, this work focuses on the software written in C/C++, and compiled to a native binary.

5.1 Detection of Real-world Security Exploits

We evaluate the effectiveness of PoLPer using real-world exploits as illustrated in Table 1. The first column shows exploit patterns. The following columns present the details of attack exploits regarding the name of the software that is being attacked, the exploit name, and the privilege operations exploited. A comparison between PoLPer and other approaches follows. CFI (control flow integrity), NCI (non-control data integrity), DFI (data flow integrity) respectively refer to the approaches that detect control flow manipulation [3], non-control data manipulation [12], and the manipulation of both [10]. Note although DEP [37, 53] has been widely deployed to prevent stack smashing attacks, it has not been successful due to the ROP attacks, which can usually disable DEP and transfer control to the shell code using mprotect in Linux and VirtualProtect in Windows [1, 2].

PoLPer uses Process Data Contexts (P-DC) and Process Call Contexts (P-CC) determined using process hierarchy contexts at run-time to detect each exploit. For all cases, PoLPer detects them by using process contexts, while only one of CFI or NCI is able to detect each attack. Data flow integrity (DFI) approach can detect both control data and non-control data manipulation but with a significant overhead [10]. PoLPer can detect these exploits with negligible overhead (see Section 5.3). In all cases, the contexts of privilege sensitive operations are extracted and enforced while identifying the process hierarchy context at run-time. PoLPer can complement DEP, CFI, NCI when they fail depending on the exploit type and it offers efficient prevention comparable to DFI with much lower overhead to be presented.

The top two cases use data-oriented programming to manipulate data without changing the control flow of a program where control flow integrity (CFI) cannot detect them. The following cases involve a change in its call stack where a setuid system call is called. CFI, DFI, and PoLPer (P-CC) are able to detect the attacks while NCI could not detect them.

Ground Truth Validation: As a ground truth validation, we manually examined the source code to locate relevant setuid family calls of 1–25 function call instances in only 10 minutes−1 hour due to a small well-defined scope of privilege functions. In our environment set up and the workload given, we did not face false positives or false negatives, and manual code examination also found our analyses properly cover the code relevant to our experiments.

5.2 Extraction of Process Contexts

Next, we evaluate the details of PoLPer regarding the extraction of multiple contexts.

Extraction of Process Hierarchy Context: Table 2 presents the process hierarchy context extracted from dynamic training. We cover 8 setuid family system calls in Linux shown in column headers for the programs shown in the row headers. seteuid and setegid columns are omitted since they are replaced with setresuid and setresgid calls. Column $D_0, D_1, D_2$ respectively show the process hierarchy information of depth 0, 1, and 2 (i.e., the root, a child of the root, a grandchild of the root). Inside each column, there are two sub-columns, $P$ and $I$. $P$ shows the number of process instances and $I$ shows the average number of setuid call invocation per a process.

This table illustrates process-aware privilege operations under various process depths (depth 0–2). This new context of process hierarchy enables more concise and stricter contexts for each individual depth by avoiding the merge of the contexts for all depths at run-time each process is restricted only using the contexts of its depth.

Reduction of Rules: Each context of a process is described as an enforcement policy rule. Process hierarchy context enables the reduction of run-time complexity by checking only the rules associated to each process depth. Table 3 shows this runtime cost reduction. The reduction ratio of enforcement rules (shown as Rule cut) is calculated as $\frac{\text{Process aware rules}}{\text{Non-process aware rules}} = \frac{\sum D_i |n_i|}{N|\cup_i D_i|}$, where $D_i$ is a set of distinct rules for a depth $i$, $n_i$ is a number of processes for a depth $i$, and $N$ is the sum of $n_i$. In general the programs with multiple processes have high percentage of rule reduction. Overall there is 48.92% reduction of the rules on average.

Extraction of Static Data Context: Table 4 shows data contexts systematically extracted from source code. For each setuid system call, our static analysis determines where the parameters come from. The sources of parameters are shown as four notations. If the parameter comes from a function call (e.g., seteuid(getuid(0))), it is shown as $F$. The parameter coming from a constant (e.g., seteuid(0)) is counted as $C$. When the parameter is determined from a data flow analysis which eventually derives it from a constant or a function, it is shown as $V$ (e.g., int uid = 0; seteuid(uid)). If the static analysis cannot determine the source of a parameter, it is indicated as $N$. Since several programs have complex code/data structures and initialize values using the values from a file (e.g., /etc/passwd), this $N$ case is supported by dynamic data context analysis. Note this is an implementation issue of static analysis which can be improved by advancing the scalability of data flow analysis or with advanced system-wide analyses.

Extraction of Dynamic Process Data and Call Context: The run-time data context and call context in our dynamic analysis are shown in Table 5. Column $C$ represents the diversity of call contexts for each privilege operation and Column $D$ shows the average number of distinct parameters per each context. For instance, if $C$ is 1, there is only one particular way to call a privilege operation inside the program. If only one value is used for its parameter, $D$ would be 1. In the same reason as Table 2, seteuid and setegid columns are omitted due to their replacement by other operations.
**Table 1:** Evaluation of security exploit analysis. This table presents the effectiveness of PoLPer for real-world exploits. P-DC and P-CC respectively represent data contexts and call contexts by process hierarchy context.

**Table 2:** Extracted process hierarchy context. This table presents setuid syscalls and the process hierarchy (column head) for each application (row head). P is a number of process instances and I is an average number of setuid calls per process called. D₀ is the root, Dᵢ is a nᵗʰ child of D₀. It illustrates PoLPer’s novel capability to learn and detect *process-aware* contexts.

**Table 3:** Reduction of rules due to process-aware restriction using process hierarchy contexts.

**5.3 Performance Impact**

We evaluated the performance impact of PoLPer in two aspects: the run-time overhead caused by the inspection of an individual setuid call, and end-to-end performance of applications.

**Micro-benchmark:** This benchmark measures the cost of the inspection of a setuid family system call which includes (1) an interposition of a system call, (2) getting its process hierarchy depth, its parameter(s), and call stack information, and (3) retrieving and comparing the run-time values with the PoLPer’s profile.

Note that all data types used by setuid family system calls have well-defined sizes (e.g., integers) and do not use complex data types such as strings, arrays, or pointers. Therefore, data contexts are efficiently checked without any uncertainty of overhead.

Figure 4 depicts the overhead of PoLPer’s inspection of a setuid system call. This graph shows the run-time overhead of a setuid call verification with a varying size of call contexts and data contexts. The X-axis represents the number of call context and the Y-axis indicates verification time in microseconds. The complexity of inspection due to different data context size is shown by different graphs. As the size of data context (e.g., parameters) increases from 1 to 10 (shown as d1 to d10), inspection time of a system call increases. Also, a higher number of call contexts causes higher overhead.

**Ground Truth Validation:** We performed a ground truth validation using manual analysis of the source and confirmed that the process hierarchy, call, and data contexts from our static and dynamic analyses correspond to the application behavior in our test bed’s configurations and workloads.
If there is one call context with one data context for a setuid call, its verification takes 18 µs. As the number of call and the size of data context are increased, the overhead increases as well. These cases are simulations of extreme situations with a lot of call contexts and data contexts for setuid system calls showing how PolPer will work in challenging cases. Real-world programs typically have only a small number of call contexts and data contexts as described in Section 5.2. Therefore, real-world overhead is typically insignificant as presented next.

End-to-end benchmarks: We performed another set of experiments to measure the overhead of PolPer in the end-to-end performance of application software. A list of benchmarked software was selected based on the popularity of server and client software which have privilege sensitive code invoking setuid system calls. The selection includes from small utility programs such as ping to large server programs (e.g., Nginx and Apache). Based on our study of these desktop and server software, the overhead is typically not strongly dependent on the workload because generally the setuid calls are not used per each workload but per an initialization of software. Table 6 shows the data. The column \( |I| \) shows the number of setuid calls.

Overall the performance impact of setuid inspection is very marginal. First, we tested ping where each try sends 10 packets repeated 10 times and overhead was 0.02%. Ping has one setuid call verified. Nginx and Apache servers are tested by using Apache Bench (normally used for measuring of HTTP web servers). To measure overhead, we repeated 100K HTTP GET requests 10 times with concreancy of up to 100 requests to the localhost. There was no overhead other than measurement errors since Nginx and Apache do not have setuid calls verified during the benchmark.
workload. Login/logout behavior was tested with telnet. This workload triggered 5 setuid call verifications. We repeated this process 10 times to get an average performance number (0.29%). Another popular software for login/logout is SSH, which is tested with SCP as another selection because they share authentication logic. SCP was measured by downloading a 1KB file 10 times from a local directory. During this process, there were 28 verifications of setuid calls which caused overhead of 0.54%.

5.4 Case Study: A Real-world Data-oriented Attack

We leverage a vulnerability in sudo to show how PoLPer detects and prevents exploitation. Hu et al. [24] proposed an approach to constructing data-oriented attacks automatically and showed an attack on sudo using its format string vulnerability (CVE-2012-0809). This attack changes ud.uid to the root ID value using the format string vulnerability in the sudo_debug function as shown in Figure 5.

PoLPer detects this attack using a data context shown in Table 7. ud.uid should be initialized by getuid(), which has an expected value of 1000. This exploit is detected because it sets zero to ud.uid.

```c
struct user_details {
    uid_t uid;
    ...
} ud_details;

//in get_user_info()
ud.uid = getuid();
//in sudo_debug()
vprintf(...);
//in sudo_askspass()
setuid(ud.uid);
```

Figure 5: Sudo code example.

5.5 Case Study: Process-aware Detection of a Data-oriented Attack

In Figure 6, we present an example of data modification attack that highlights the unique capability of process-aware detection that can distinguish the abnormality of the attack. In this example, setuid call sets a user ID from a variable uid determined inside an if-else statement. Based on code analyses both non-root-id (from getuid) and root_id should be permitted because they are used respectively by a child and a parent process.

```c
int change_privilege(int uid) {
    ...
    return setuid(uid);
}

void vulnerable_function() {
    int uid;
    char buffer[10];
    int pid = fork();

    if (pid == 0) { //child process area
        uid = getuid();
        ...
        strcpy(buffer, argv[1]); // manipulate uid (buffer overflow)
        change_privilege(uid);
        non_root_works();
    } else { //parent process area
        uid = root_id;
        change_privilege(uid);
        root_works();
    }
    ...
}
```

Figure 6: Process-aware exploit example.

<table>
<thead>
<tr>
<th>Non-process-aware</th>
<th>Process-aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>(parent/child)</td>
<td>(parent)</td>
</tr>
<tr>
<td>setuid</td>
<td>0</td>
</tr>
<tr>
<td>Parameter</td>
<td>Profile</td>
</tr>
<tr>
<td>(Profile) 1000 (from getuid()), -1, -1</td>
<td>root_id</td>
</tr>
<tr>
<td>(Exploit) 0 (root shell), -1, -1</td>
<td>non-root-id</td>
</tr>
<tr>
<td>Exploit</td>
<td>Profile</td>
</tr>
<tr>
<td>root_id</td>
<td>root_id</td>
</tr>
</tbody>
</table>

Table 7: Process-aware detection of the sudo exploit.

<table>
<thead>
<tr>
<th>Call Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td># Inode</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>21 158023</td>
</tr>
<tr>
<td>20 10171</td>
</tr>
<tr>
<td>3 10171</td>
</tr>
<tr>
<td>1 10171</td>
</tr>
</tbody>
</table>

Table 8: Comparison of non-process-aware and process-aware enforcement.

When an adversary manipulates the uid using a buffer overflow in the strcpy function to obtain the root_id and a corresponding root shell in a child process, a non-process-aware approach cannot block this attack because both root and non-root-id are legitimate as shown in Table 8. In contrast, PoLPer will prevent this exploit attempt because it distinguishes the setuid parameters in parent and child processes at run-time and enforces that only non-root-id is allowed in a child process. This case highlights a unique strength of process-aware detection.
6 DISCUSSION

Extended Support for Other Privilege Operations: This work focuses on the protection of setuid family system calls in Linux as these system calls are commonly leveraged for privilege escalation attacks. Privileged processes can bypass kernel permission checks and it is hard to control their privileges in a fine-grained way. Therefore, from Linux kernel version 2.2, privileges were divided into several categories according to capabilities [8].

Although capabilities were designed to remedy the problems of setuid family system calls, many popular legacy programs do not adopt them. Therefore, the vulnerability and problems of setuid calls still remain.

Conceptually, capabilities are also privileges and PoLPer can be extended to cover these sensitive operations as a future work — this extension is pure engineering effort and orthogonal to the conceptual work presented in this paper.

Analysis Coverage: In general, the focus of our code analysis is small relative to the full size of the application since setuid system calls are a small subset of all available code and can, therefore, be evaluated by enumerating all call sites of setuid calls. As a ground truth validation for our evaluations, we manually examined the source code of the applications evaluated and we found setuid family calls of 1–25 function call instances only in 10 minutes–1 hour due to a small well-defined scope of privilege functions. Manual code examination confirmed that our combined static and dynamic analyses achieve complete coverage relevant to our environment and the workload giving no false positives in experiments.

Based on our experiments, this is reasonable as in many programs setuid calls are used for the initialization of services which are common across workloads. For example, many client programs (e.g., ping, sudo, su) and server programs (e.g., Apache, Nginx) follow this pattern. Another set of programs (e.g., ssh, scp) exhibits complete coverage by having commonly used functions in every transaction which cannot be missed by either of our analyses.

However, if the software is written in a way which uses frequent setuid calls with diverse patterns in complex software structures, it may cause high complexity in the static and dynamic analyses to achieve complete contexts. As a complementary analysis method, fuzzing framework such as AFL [58] could help to further improve the coverage of static and dynamic analyses if needed for such complicated programs.

Mimicry Attacks and Manipulation of Call Contexts: For a mimicry attack where a system call sequence may fall within the original program’s pattern, our approach will validate data parameters, and call stacks with process hierarchy contexts on setuid calls beyond a system call sequence. A combination of these contexts will significantly raise the bar to make a meaningful attack or maintaining its control without a detection. As another possible attack, a fake stack on the memory can be easily determined by PoLPer because it has complete knowledge on the process’ stack memory. Misleading the view of stack walking via the manipulation of the ebp register can be easily caught because our approach uses stack layout from .eh_frame which verifies the register values and stack consistency in the unwinding steps similar to [20]. As a worst-case scenario, even though the adversary managed to bypass control flow integrity check, our approach applies multiple context checks on the data, process as well. Thus it would be considerably challenging to evade all of them. PoLPer can be further improved to be resilient to advanced stack attacks by combining with several known techniques such as a shadow stack [15] and Control-flow Enforcement Technology (CET) [26].

Address Space Layout Randomization: The kernel component of PoLPer uses information regarding memory layout including individual library addresses and therefore supports ASLR as we record not absolute addresses but files and offsets.

Generalization of Model: As we described in Section 2, our usage model of PoLPer is that system administrator customizes the policy based on an environmental context on the deployed system. This is a similar model used in major security tools like AppArmor or SELinux to deploy policies. On the deployment of a new software, this model is updated along with the installation via a new training. We are expecting that updating rules for new applications would not be different from already tested applications. This is because PoLPer is based on the general program behavior of setuid system calls such as process hierarchy, call stack, and parameters, which are orthogonal to the types of applications. Also, PoLPer did not have any particular assumptions about the target applications. When a new user is added into the system (e.g., adduser), which is another case to change an environmental context, PoLPer requires to update the model as well to handle a new user. Since a user ID is easily recognizable in the rule sets, it is straightforward to extend data flow check to use an ID template, therefore, including or excluding users. Our future work on this ID generalization will further improve the convenience and usability of PoLPer.

7 RELATED WORK

In this section, we discuss the approaches related to PoLPer and how PoLPer is differentiated from them. Table 9 provides a comparison between PoLPer and other works that are most closely related.

The principle of least privilege [48] means enforcing minimal privileges that allow the user/module to perform an intended role. The principle of least privilege is mainly achieved by separating the system into isolated compartments or using other techniques such as setuid system calls [11], Linux capabilities [8]. Qmail [5] has a software architecture that follows this principle (separating modules run into separate user IDs). Approaches such as JIGSAW [56], WatchIT [50], program compartmentalization [21, 23], SMV [23], SOAAP [21], Minion [30] follow this principle to reduce unnecessary privileges. PoLPer also follows this principle with a focus on the setuid system calls. The necessary privilege is extracted from the source code and dynamic training and enforced at run-time.

Several approaches [11, 17, 28] have investigated the status of setuid system calls and identified their semantic inconsistency. This problem occurred with human errors because they were insufficiently documented and poorly designed. Authors proposed more stable high-level APIs instead of low-level setuid system calls [11, 17] or migration of setuid policies from user-space programs to the kernel [28] as remedies.

Another line of work [19, 32, 38] models system call behavior during run-time. These approaches only rely on run-time behavior, unlike PoLPer. Additionally, they are based on process insensitive context. Therefore their detection policies do not differentiate
These approaches are based on mandatory access control systems (e.g., privilege escalation) based on the manipulation of control and non-control data manipulation). With the rise of automatic synthesis of non-control data attacks [12, 24, 25, 27], data-flow must be protected as well. Current fine-grained solutions are either not yet practical because of coverage issues (e.g., KENALI [52] is only designed for OS kernels) and overhead issues (e.g., CFI [3]’s overhead is 104% since it handles all control and non-control data manipulation).

For the detection and prevention of privilege operation attacks (e.g., privilege escalation) based on the manipulation of control and non-control data, practical data flow integrity checking solutions are needed. PoLPer provides coarse-grained data context integrity (both control and non-control) on the scope of setuid calls with negligible overhead.

### 8 CONCLUSION

PoLPer systematically extracts only the required contexts of setuid calls from programs to discover the distinct demand of privilege operation of each process. PoLPer transparently enforces these process-aware characteristics using a comprehensive combination of process contexts so that unnecessary contexts of setuid calls are tightly restricted in legacy software without any change. Our evaluation presents that PoLPer can prevent real-world exploits based on state-of-the-art attack techniques manipulating data context or control context of setuid system calls effectively and efficiently with near zero overhead in the end-to-end performance in various desktop and server programs.

### ACKNOWLEDGMENTS

The first author worked on this project during an internship at NEC Laboratories America, Princeton. We would like to thank the anonymous reviewers for their detailed and constructive comments. This work was supported by NSF awards #1801601, #1513783, and ONR award N00014-17-1-2513. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.