Improving Logging to Reduce Permission Over-Granting Mistakes

Bingyu Shen, Tianyi Shan, Yuanyuan Zhou
University of California, San Diego

Abstract

Access control configurations are gatekeepers to block unwelcome access to sensitive data. Unfortunately, system administrators (sysadmins) sometimes over-grant permissions when resolving unintended access-deny issues reported by legitimate users, which may open up security vulnerabilities for attackers. One of the primary reasons is that modern software does not provide informative logging to guide sysadmins to understand the reported problems.

This paper makes one of the first attempts (to the best of our knowledge) to help developers improve log messages in order to help sysadmins correctly understand and fix access-deny issues without over-granting permissions. First, we conducted an observation study to understand the current practices of access-deny logging in the server software. Our study shows that many access-control program locations do not have any log messages; and a large percentage of existing log messages lack useful information to guide sysadmins to correctly understand and fix the issues. On top of our observations, we built SECLOG, which uses static analysis to automatically help developers find missing access-deny log locations and identify relevant information at the log location.

We evaluated SECLOG with ten widely deployed server applications. Overall, SECLOG identified 380 new log statements for access-deny cases, and also enhanced 550 existing access-deny log messages with diagnostic information. We have reported 114 log statements to the developers of these applications, and so far **70 have been accepted into their main branches**. We also conducted a user study with sysadmins (n=32) on six real-world access-deny issues. SECLOG can reduce the number of insecure fixes from 27 to 1, and also improve the diagnosis time by 64.2% on average.

1 Introduction

1.1 Motivation

Modern computer systems often rely on sysadmins to customize access control policies to achieve the security goals of an organization. Unfortunately, access control has been reported as "profoundly broken" due to the prevalence of misconfigurations [45, 56, 67]. Many recent security incidents have shown that even subtle errors in the access control configurations can result in severe data breaches and system compromises. For example, a misconfigured server of a billion-dollar consulting services company exposed customers' private information, certificates, 40,000 passwords, and other sensitive data to the public [17]. More recent real-world newsworthy security incidents caused by on access control misconfigurations [29, 30, 32, 33, 46, 59, 72].

Problem: client could not download a file
Log message: `get xxx.mp4` `550 Permission Denied.`
Attempts
1. Change file permission to 777 [harmful][useless] 2. Change file owner to the logined user. [userless] 3. Remove the `userlist_file` config entry to allow all users login. [harmful][useless]
Correct and safe fix
Access denied because the server's configuration will deny access to file matched with the pattern specified in `deny_file`. Change the pattern in this entry will allow download this file.

Figure 1: **Real world example of an access-deny issue for Vsftpd.** Sysadmins made several blind attempts due to lack of diagnostic information, some of which introduced security vulnerabilities. Our tool SECLOG can improve the log message by pinpoint the relevant configuration `deny_file` in the log message.

While access control misconfigurations can be introduced during initial settings, many are caused when sysadmins change configurations to resolve issues [61]. One of the most prominent examples is *access-deny issues*, i.e., legitimate users are denied access to data that they are supposed to have. As these users need to access the data to perform their jobs, sysadmins need to diagnose the issue and change access control settings to grant these users access.

However, server software often does not provide informative logging to guide sysadmins to understand the reported problems, which may result in incorrect fixes. Figure 1 shows a real-world access-deny issue in Vsftpd. The user was denied from accessing files on the FTP server and the server log only shows "Permission denied". The sysadmin attempted several fixes during the problem-solving process, some of which opened up access to more users and introduced security holes that can be easily exploited by malicious users. According to a recent study on sysadmin practices based on sysadmin online forums and mailing lists, 38.1% of the access-deny issues were resolved by insecure modifications that over-granted accesses, introducing security vulnerabilities [67].

Our main goal is not just about fixing the unintended access-deny issue for that particular user: we are more concerned about insecure "fixes" that over-grant the permissions. After all, there can be many ways to "fix" such issues for that particular user. For example, many incorrect fixes simply disable the entire protection domain and make the object accessible to everyone [67]. While such a "fix" resolves the issue for that particular user, it opens a big door for others, including

```
/* Cherokee-1.2.103: cherokee/config_reader.c*/
dir = cherokee_opendir (path->buf);
if (dir == NULL)
+    LOG_CRITICAL ("Could not open directory '%s', check the
server user and file permissions.", path->buf);
    return ret_error;
```

Silently ignoring the denied access in reading config file

(a) Cherokee

```
/* httpd-2.4.46: modules/aaa/mod_authz_core.c */
status ip_check_authorization(request *r, ...) {
  …; return AUTHZ_DENIED;/*check request's IP w/ config.*/
}
status method_check_authorization(request *r, ...) {
  …; return AUTHZ_DENIED;/*check request method w/ config.*/
}
status env_check_authorization(request *r, ...) {
  …; return AUTHZ_DENIED;/*check environment vars. w/ config.*/
}
int authorize_user(request* r, ...) {
 ...
 if (auth_result == AUTHZ_DENIED) {
  ap_log_rerror(APLOG_ERR, r, APLOGNO(01631)
   "%s: authorization failure for \"%s\": ",r->user, r->uri);
}
```

Invoke above functions based on the request's config. All w/ the same error msg.

(b) Apache httpd

Figure 2: **Two log message examples in two widely-used web servers, Cherokee and Apache httpd.** (a) Cherokee's directory opening is denied silently without any log messages. *This case was first identified by our work. The log message patch (generated based on our tool* SECLOG*) has been accepted.* (b) Apache denies access without logging specific root cause, different causes (IP, method, environment) have only a generic message, providing little information for diagnosis.

malicious users, to get into the system and access sensitive data. Even worse, it can remain undetected until exploited by malicious users, causing a major real-world security incident.

Therefore, to reduce the number of such security incidents, it is important to help sysadmins to resolve access-deny issues safely. While it is impossible to eliminate human mistakes (after all, to err is human), one natural question is whether system builders (i.e., server software developers) can improve our software to help sysadmins diagnose and fix such issues.

Fortunately, the answer is 'Yes'. When a sysadmin tries to fix an access-deny issue, the first step is to understand (1)*Who gets denied? (i.e., "subject")*; (2)*What operations get denied? (i.e., "operation)*; and (3) *What data is denied access? (i.e., "object")*. These questions may sound trivial but on investigation are complicated. The "subject" may not be the user who reported this issue, because today's systems involve many components such as application servers, databases, etc. Similarly, the "object" and "operation" may not be that straightforward either. For example, for users who cannot access a web page, their problem may be related to that page's file permission, or the clients' IPs.

A common practice to understand access-deny issues is to check the server logging. Unfortunately, logging practice in today's systems is ad-hoc and lacks critical details [67]. The incompleteness of access-deny related log messages is reflected in two ways. **First**, some access control checks deny permissions silently without any log at all to guide sysadmins. Figure 2a shows such an example in the Cherokee web server [2]. If there is no log message at all, sysadmins have to go on a wild goose chase. Sometimes sysadmins resort to the online internet forums, but the inadequate log messages often bring incorrect suggested fixes which can result in major security vulnerabilities [68]. **Second**, even when log messages exist, the messages are often too generic, and lack critical information related to subject, object and operation, all of which are important for sysadmins to understand the root cause and fix the settings safely. Figure 2b shows a real-world example of denied access with only generic information in log messages. Although there are three different possible reasons, i.e., 1) HTTP request method, 2) IP, 3) environment variables, that can result in access denial, it has only one log message: `AUTHZ_DENIED`. While this may make the source code easy to read, such generic error code provides very little value for sysadmins to troubleshoot and fix the access-deny issue.

As such, it is critical to improve logging quality to provide accurate information for sysadmins to fix access deny issues correctly. Unfortunately, to the best of our knowledge very little work has been done on this direction. The closest work was efforts by Yuan et al. on improving log messages for the purpose of software bug diagnosis [69, 71, 74]. While this work is inspiring, logging for developers is different from logging for sysadmins. To fix access-deny issues correctly, log messages for sysadmins need to have different critical information and thus different techniques need to be applied to identify such information. More details about are in §6.

1.2 Our contributions

This paper makes one of the first attempts (to the best of our knowledge) to help developers improve log messages for sysadmins to diagnose and fix access-deny issues correctly and safely. First, we studied five large open-source server programs including Apache httpd, PostgreSQL, Vsftpd, NFS-Ganesha, and Proftpd to understand the problems in access-deny logging practice. Our study confirms our motivation that: (i) many access control check locations do not have any log statements for access-deny, leaving no information for sysadmins; (ii) Even for cases with access-deny log statements, a large percentage of them lack relevant details such as subject, object, and action of the denied access to guide sysadmins understand and fix such issues (§2).

Second, we built a tool called SECLOG which employs static analysis to automatically detect access control check locations in various server applications. SECLOG identifies missing access-deny log locations and relevant information at log location. We evaluated SECLOG with ten widely deployed server programs. Overall, SECLOG inserts 380 access-deny log statements in these software, and also automatically enhances 550 existing log messages with relevant information. We have reported 114 inserted or enhanced log messages to the developers of these popular programs, so far **70 have been confirmed and accepted into their main branches**.

Furthermore, to evaluate the effectiveness of our inserted or enhanced log messages, we conducted a user study with sysadmins (n=32). Our user study results show that the log messages enhanced by SECLOG can cut access-deny issue diagnosis time by 64.2%. More importantly, with the original log messages, the sysadmins introduced 27 over-granting security issues in their fixes, whereas only one such issue was introduced with the enhanced log messages.

2 Understanding Access-deny logging in Real-world Applications

Before we dive into a solution, we first aim to understand the current status in access-deny logging practices and possible opportunities for inserting such log messages with useful information for sysadmins. We first conduct an empirical study with five widely-deployed server software systems – Apache httpd (webserver), PostgreSQL (database), Vsftpd and Proftpd (FTP server), and NFS-Ganesha (NFS), written in C/C++ languages. We focus on server software systems because their access control settings are usually critical, and over-granting permissions due to mistakes can lead to major security incidents, as we have depicted in Section 1.

2.1 Methodology

To collect the access-deny log messages in software systems, we first identify the access-deny program locations through two kinds of program signatures, error code and access control check functions. First, we collect error codes related to access denial in each application from their official documentation and manuals. For example, the error codes that would lead to a 403 HTTP status include `HTTP_FORBIDDEN`, `AUTHZ_DENID` in Apache httpd. From the location where the error code is assigned and the propagation of the error code, we record the log message that is specific to the denied access.

Second, we search by the code pattern that performs permission check to find the access-deny program locations. All the studied applications use access control check (ACC) functions to perform access check to system resources (e.g., file and port), or perform application-specific access check (authorization/authentication modules, DB privileges, ACL lists). Figure 2b shows the example authorization functions in Apache httpd. Then we identify access-deny log messages by finding the call sites of ACC functions and propagation of check results. We remove the duplicates if one point can both be identified by error code and ACC function.

Our study first focuses on access-deny log points that are enabled by default during production such as `Fatal`, `Error`, `Warn` verbosity levels. To be more comprehensive, we also look into log points at more verbose levels that are usually not enabled during production, such as `Debug` or `Trace` levels, to further understand and expose issues in current practices of access-deny logging. (Finding 1)

Second, we examine whether the log message at the access-deny log point contains relevant information needed to understand the reason for denial. More specifically, we classify the information into the subject, access action and denied object at each point. (Finding 2)

Third, to understand more about how those access log messages were added, and whether they were added as afterthoughts, we also looked at the change history of the logging at each access-deny program point in the public version control repository. Only commits make changes to access-deny log points are selected. (Finding 3)

We have two inspectors meet and discuss the standard for the collected information before studying each software. Each inspector independently investigates and records the information of interest, including the root cause configuration, the relevant variables in source code, and the request's characteristics such as subject, object, and action at the access-deny program location. Then the two inspectors compared the results and discussed them with each other in iterations. All results reached a consensus in the end.

Threats to Validity. Like all characterization studies, there is an inherent risk that our study may be specific to the applications studied and thereby may not be generalized to other software. While we cannot establish representativeness categorically, we have taken care to select diverse server software systems that are widely used in different areas. Note that our study only focuses on server software where access control is critically important; the findings may not apply to client or mobile applications. These studied programs also share commonalities that all are open-source and written in C/C++, which is common in popular server systems.

Another potential source of bias is that we may miss access-deny points that use ad-hoc access control checks that can not be identified by the error codes or ACC functions. In practice, we found that the well-maintained server programs check for access via ACC functions or differentiate the logging via error codes. Ad-hoc checks which were treated as general errors by the application developers, were also excluded by our study, such as format checks of authorization headers.

2.2 Findings

Finding 1: *Under the **default** verbosity mode, 14.1% to 64.7% of cases have no log messages at all when an access is denied (Table 1).* Such an overlook in software practices makes it quite challenging for sysadmins to troubleshoot an unintended access-deny issue and fix the access control setting correctly without over-granting permissions and introducing security vulnerabilities.

Compared to other logging, access-deny logging is more important as it guides sysadmins in the right direction to fix access control settings correctly. Unlike other misconfigurations that usually have visible erroneous symptoms during software execution, *access-control misconfigurations that over-granting permissions can go silent for months without being noticed until being exploited by malicious users, causing catastrophic security incidents*.

Application	At default level		Only at debug level	
Apache httpd	115	64.6%	10	5.6%
PostgreSQL	374	77.8%	0	0.0%
Vsftpd	61	85.9%	0	0.0%
NFS-ganesha	24	35.3%	38	55.9%
Proftpd	145	56.8%	25	9.8%

Table 1: The number of access-deny program points that have log messages at default verbosity level and *only* at debug verbosity level.

App.	Subject	Action	Object
Apache	Server user, process ID	HTTP methods; file perm. (`rwx`).	Webpages,files
Postgre.	DB user	DB privileges	Tables, schemas, etc.
Vsftpd	FTP user; process ID	FTP commands; file perm. (`rwx`).	Files, directories.
NFS.	Client IP, user name	NFS commands (e.g., `mnt`)	Files, dirs, NFS exports
Proftpd	FTP user; process ID	FTP commands; file perm. (`rwx`).	Files, dirs.

Table 2: The subject, action and object in each application.

Apache httpd, NFS-ganesha and Proftpd contain some log information only at debug-level logging. While this is better than no log messages at all, the information may not be very helpful because (1) sysadmins may not know that there exist some log messages at debugging or tracing verbosity level (since most sysadmins do not read source code [67]); (2) it would require sysadmins to restart the software with debugging level enabled and reproduce the denied request to get the relevant information from log messages. As such, for the following characteristics studies, we only focus on the default level log messages.

Finding 2: *Most existing access-deny log messages lack relevant information to guide sysadmins, with 37.5-100% of the log messages missing subject information, 0.0-64.7% missing action or access type information and 0.0-75.4% missing information about the accessed objects. However, the majority (70.8% - 100%) of relevant information related to the denied access is available within the same function of the corresponding access check operation (Table 3).*

We classify useful information for sysadmins into three categories, subject, action and object, based on the specific scenario of the access [44]. There are mainly two scenarios of access-control checks which involve different subjects, actions or objects. The first scenario checks access to *system resources* including files, network sockets, etc. In this scenario, the subject is the role of server process in the OS; the action is the operation to be processed on the resource (e.g., read or write the file, bind network port); the object is the system resource. The other scenario checks the access to application-specific resources (e.g., authorization/authentication modules, DB privileges, ACL lists). The subject is the role in the application to perform the operation (e.g., the DB user or the authenticated user in the web server); the action is the required access to the resource (e.g., `SELECT/REFERENCES` privileges in PostgreSQL); the object is the application-specific resource. More details about the subject, object and action in each application are shown in Table 2.

The information related to subject, action and object is fundamental for sysadmins to understand the access-deny issue and come up with correct solutions. Missing any of the information may cause additional difficulties for sysadmins. **First**, subject information is necessary to know who is being denied. However, among all software, only a few percentages of log messages include subject information. One might assume the subject information is the legitimate user who reported the access-deny issue. But for many software programs, it is more complicated. For example, in most database servers, DB user is usually the database account used by application server to perform the operation requested by the end user, which is different from the end user. For file accesses, subject information is usually the server process, but it would be more complicated if `setgid` or `setuid` are used. **Second**, the action (access type) information is critical for sysadmin to grant only necessary privileges. For example, PostgreSQL utilizes logging templates in may places, but the template only includes the denied object, such as `"permission denied for table %s"`. They do not have information related to the user role or required privileges while PostgreSQL has 12 distinct levels of privileges [8]. **Third**, the object information is useful to know what data access is denied. Sysadmins usually need to inspect the ACLs associated with the object to decide how to change the privileges.

App.	Total	Subject		Action		Object		Same func.	
Apache	115	102	88.7%	25	21.7%	18	15.7%	112	97.4%
Postgre.	374	275	73.5%	242	64.7%	107	28.6%	357	95.5%
Vsftpd	61	59	96.7%	20	32.7%	46	75.4%	61	100%
NFS.	24	9	37.5%	0	0.0%	6	25.0%	17	70.8%
Proftpd	145	145	100%	0	0.0%	0	0.0%	145	100%

Table 3: The number and percentage of log messages that do *not* contain subject, action or object information; "Same func" is the number of access-deny program points that have relevant information within the same function of the check operation.

Application	Add logs	Revise logs
Apache httpd	40	511
PostgreSQL	165	1728
NFS-ganesha	7	344
Proftpd	33	1209

Table 4: The number of patches that added or revised access-deny logging statements in source code. Vsftpd does not have a public version control repository and was excluded from this study.

Fortunately, the majority of the relevant information is available within the same function of the access check operation as shown in Table 3. This means that when developers write the access-deny log statements in the source code, they could have included the relevant information without much complexity.

Finding 3: *Access-deny logging practice is ad-hoc and many existing log messages are added as afterthoughts (Table 4).* We find that many efforts from the developers are needed to add and revise the access-deny logging statements. The patches are made to (1) add new log message in previously silently denied program location or (2) revise existing log message to include additional information (e.g., file name). The added information was crucial to help sysadmins resolve the denied access, but would require huge efforts from experts to find and understand the denied locations, which calls for the need for an automated tool to assist the developers.

3 Challenges and Design Choices

Motivated by our real-world applications observation, we aim to design and build a tool, called SECLOG, to help software developers to enhance access-deny log messages and also insert missing log messages to provide guidance for sysadmins to fix access-deny issues without over-granting permissions. To achieve this goal and improve the quality of access-control logging in general, there are three fundamental challenges.

1. **How to log:** How to maintain high-quality log statements in a continuous software development process.

2. **Where to log:** How to identify the access-control check (ACC) program locations (including missing ones) in large server software and where to place the log statements.

3. **What to log:** How to find the critical information to add into log statements that can guide sysadmins in solving the access-deny issues safely and correctly.

We discuss the challenges and design choices in detail, as well as **compare them with the alternative approaches.**

3.1 How to Log

Software code is constantly evolving to meet the dynamic needs, which also requires developers to spend efforts to maintain high-quality log statements. However, current access-deny logging practice is ad-hoc as it relies on developers to manually identify log locations and add useful logging information at each location (c.f. §2). One simpler approach is to provide the developers with a logging library to guide/enforce developers writing better log messages. While it is possible to manually make an improvement at each access-check program location, this process is tedious and error-prone because (1) there are hundreds of ACC program locations in server software (e.g., more than 300 access-check points in large software like PostgreSQL) and (2) for each location, developers need to examine the entire call chain to search for log statements or the lack of such statements, as well as to identify critical information related to the access denial.

We design SECLOG to automate the process with the help of static analysis for the following considerations. **First**, the source code contains rich information related to the denied access, which could be extracted with static analysis (Finding 2 in §2). **Second**, static analysis can go through all the access-control check locations to identify inadequate or missing log messages, which would be challenging for developers to manually go over. **Third**, the automated process could be integrated to the CI pipelines to help with code review, which enforces consistent logging practice for even new contributors. Though SECLOG still requires some annotations from developers, this as a one-time small effort would benefit the code development process in the long run.

3.2 Where to Log

3.2.1 Where to Find ACC Locations

It is challenging to identify ACC locations across different server applications because each application performs various access control checks on the requests. Manually identifying each location in a large server application requires expertise and huge efforts.

SECLOG addresses this problem by leveraging the common adoption of ACC functions to reduce the input from developers while achieving a high coverage of access check locations. To understand the percentage of access-check locations covered by ACC functions, we performed a measurement on the five applications in §2. We find that the majority (65.6%-100%) of access check program points can be identified by ACC functions. (More details in Appendix A.1.) Using ACC functions allows SECLOG to identify up to more than 300 access control check locations in large software like PostgreSQL with no more than 34 ACC functions (Table 17).

Although ACC functions still have to be annotated by developers, the effort is much lower than manually inspecting hundreds of access-check points. The developers can easily identify these ACC functions in modules related to system resources access (e.g., file or network), or application-specific checks in access control related source files. As shown in Evaluation §5.3, even novice developers could annotate the ACC functions with high coverage in a short amount of time.

3.2.2 Where to Place Log Statements

After SECLOG finds all ACC program locations, the second challenge is where the existing or new (to be added by SECLOG) access-control log statements should be placed. One approach would be placing the log statements *inside* the ACC function instead of after ACC function's call sites. However, the call site contains more relevant information for sysadmins to resolve the access-deny issues. According to our study, 68-100% relevant information related to the denied access is available within the same function of the call site of ACC functions (Table 3). Besides, the majority of access-deny logging statements are at ACC functions' call sites instead of inside a check function (More details in §A.1 Table 18). Placing the logging statements at call sites, SECLOG can improve existing logging statements without intrusive logic modifications to source code.

Another alternative approach is to *always* add after the check at the call site, regardless of whether the access result is denied or not. This approach would have two disadvantages. **First**, too much logging will introduce higher performance overhead and downgrade the server throughput [60]. **Second**, too much logging can overwhelm sysadmins to find the related log message when they need to fix issues like an unintended access-deny case reported by a legitimate user.

In sum, SECLOG decides to place the log statements after the call sites of ACC functions only when the access is de-

nied. To achieve this, we design a semantic pattern matching algorithm to identify the access-deny paths in Section 4.1.

3.3 What to Log

The third challenge would be what should be included in the access-deny log messages to help sysadmins fix the access-deny issues correctly. A naive solution would be to include all the parameters of the ACC function as relevant information since they are used to perform access control checks. However, it is imprecise and misses the opportunity to collect specific denied reasons. **First**, not all function parameters are involved in the decision of whether the access would be denied. Logging unrelated information may confuse the sysadmins. **Second**, the parameters may be a large object represented as a struct with many fields (e.g., the request object in Apache httpd contains more than 100 fields). Only the fields that cause the access denied are relevant and should be logged. **Third**, some accesses may be denied in an ACC function for different reasons. Figure 3 shows that the `cherokee_mkdir_p_perm` function could be denied for two reasons: (1) denied to *create* a new directory with the specified mode, or (2) denied to *open* an existing directory with certain permissions. To guide sysadmins for an unintended access-deny case, it is more useful to give the specific information in the log message.

1. Access control check function

```
cherokee_mkdir_p_perm(buffer_t* dir, mode, perm){
  re = cherokee_stat(dir->buf, &foo);
  if (re != 0) {/*if not exist, create the dir.*/
    ret = cherokee_mkdir_p(dir, mode));
    if (ret != ret_ok)
      return ret_error;   <- Deny path 1 return value
  }
  /* dir exist, check permissions */
  ret = cherokee_access(dir->buf, perm);
  if (ret != ret_ok)
    return ret_deny;      <- Deny path 2 return value
  return ret_ok;}
```

variables on path1
variables on path2

2. Access control check function call site

```
cherokee_handler_rrd(…){ //...
  ret = cherokee_mkdir_p_perm(img, 0775, W_OK);
  if (ret != ret_ok) {
- LOG_CRITICAL("Cannot create the '%s' directory",img);
+ LOG_CRITICAL("Cannot create the '%s' directory; or
+  the directory doesn't have write permissions", img);
  return ret_error;
}
```

Figure 3: **Collecting access-deny information inside the access control check function.** With the result check function specifies `retval != ret_ok`, SECLOG detects two different access-deny return points and constructs two slices with backwards slicing. The first sliced path extracts `mode` whereas the second one is related to `perm`.

To find precise information related to access control from source code, SECLOG identifies the information relevant to the access from two sources: (1) inside the ACC function and (2) at the ACC function's call site. To ensure the collected information is only related to the denied access, SECLOG performs backward slicing from the access-deny return value inside the ACC function and data dependency analysis to find the related variables. We discuss more about the analysis process in Section 4.2.

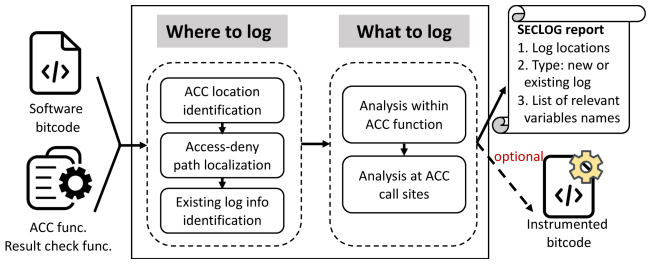


Figure 4: **Workflow of SECLOG**

1. Access control check function

```
// check permission on opening a file; return the check result as status struct
status_t check_open_permission(struct obj_handle* obj,
            struct flag_t* flags, bool create);
```

2. Access control check function call sites

```
status_t fsal_open2(struct obj_handle** obj, struct flag_t* flags,
        enum mode cmode, ...){
... // init
status = check_open_permission(*obj, flags, cmode>=FSAL_EXCLUSIVE);
if (! (status.major == NO_ERROR)) goto out;
```

3. Result check function

```
bool result_check_int_1(status_t status) {
  return status.major != NO_ERROR;
}
```

Return true when the access control check function's return value represents *access denied*

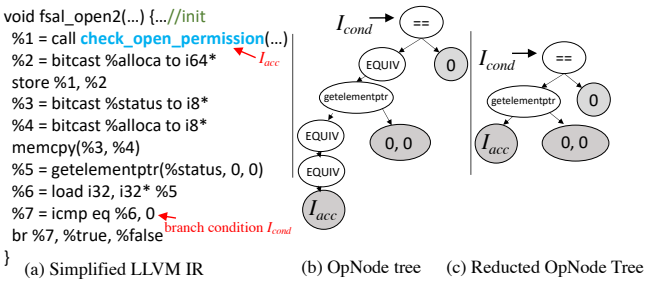
Figure 5: **An access control check function example from NFS-Ganesha.** The function `check_open_permission` returns a struct to represent error status. The result check function (provided by the developers) determines whether the return value of the access control check function indicates "access denied".

4 Design and Implementation

SECLOG targets large server software where over-granting permissions and security incidents can cause major incidents. Since most server software was written in C/C++, SECLOG is built on top of LLVM compiler frameworks that can handle C/C++ programs. SECLOG's static analysis algorithms can be easily extended to handle Java applications.

The overview workflow of SECLOG is shown in Figure 4. SECLOG operates on the LLVM bitcode of the application and processes the IR representation of the source code. Besides, SECLOG takes two types of annotations from the developers: (1) A list of developer-selected ACC functions in the target software, and (2) the corresponding result check function paired with the ACC function that is used to determine whether the access is denied based on the return value of ACC functions. For example, if the return value from an ACC function is not `NO_ERROR`, then it is access-denied. One example ACC function and corresponding result check function is shown in Figure 5.

SECLOG further performs analysis to identify (1) where to log (§4.1) and (2) what to log (§4.2). SECLOG produces a report including the list of access-control log locations, the type of log at each location (existing or new), and the list of relevant variable names at each location. Developers can utilize SECLOG's report to generate the final human-readable log messages. SECLOG can also be configured to instrument



(a) Simplified LLVM IR (b) OpNode tree (c) Reducted OpNode Tree

Figure 6: Result checking operations in LLVM IR for call site's code snippet `fsal_open2()` in Figure 5. The corresponding `OpNode` tree is constructed and reducted for matching in Algorithm 1.

the bitcode by inserting the pairs of variable names and values at each logging location with provided logging functions.

4.1 Identify Where to Log

SECLOG identifies the logging locations with the help of ACC functions as discussed in §3.2.1. To achieve this goal, SECLOG will first perform static analysis to identify the access-control check locations. Then SECLOG places the log statements only when the access is denied at the check location, by identifying the access-deny paths at the ACC function call site. We discuss the detailed analysis algorithms as follows.

4.1.1 Identify Access-control Check Locations

SECLOG identifies the log location by first extracting the call graphs from the bitcode. With the help of call graph, SECLOG can comprehensively find all the call sites of ACC functions where the access is checked. Most ACC functions in the same software follow a similar convention to indicate access denial in the same software. This makes it easier for developers to provide result check functions.

We find that function pointers are used as special ACC functions in certain C-based applications, such as Apache httpd. The application uses function pointers in a struct as a function template to call different ACC functions at any call site. However, the use of function pointer makes the log messages at the function call sites less specific, since the different access control check functions may be called at the same site, but fails for different reasons, as shown in Figure 2b. To find the call sites via the function pointers, we require the developers to annotate the specific field in the struct that represents the function pointers to ACC functions. Then SECLOG identifies all possible functions that can be assigned to this type of pointer. This helps us extend the call graph between the call site of function pointer and the real ACC functions to conduct the remaining analysis.

4.1.2 Identify where to Place Access-Deny Logs

As discussed in §3.2.2, SECLOG decides to place the log statements only when the access is denied. This requires SECLOG to identify the existing access-deny path in the source code. If there is an access-deny path (i.e., the code snippets at the ACC function's call site to check whether the ACC function's return value represents access-deny), the snippets would be similar to the operations included in the result check function. Figure 6(a) shows the corresponding LLVM IR of the example function's call site in Figure 5. The ACC function `check_open_permission` returns a struct of error status. To perform the check in LLVM IR code, the call site performs several operations to retrieve one field in the struct and check if that field represents the access is denied.

To capture the result checking operations and identify access-deny paths, as detailed in Algorithm 1, SECLOG performs a semantic pattern matching process by using the program slicing technique with data flow analysis starting from each access-deny program location. The algorithm will match the result check function with the denied branch.

Semantic pattern matching algorithm. The result check function takes the return value of the ACC function as an input, and returns true when the access is denied as shown in Figure 5. The algorithm perform the analysis to match the result check function with the denied branch in two steps.

First, for each call site, SECLOG identifies the call instruction I_{acc} (i.e., the instruction that invokes the ACC function) and the branch conditions impacted by I_{acc}. It performs intra-procedural static forward slicing that has data dependency on I_{acc}. S_{acc} is the set of all instructions in I_{acc}'s forward slice. SECLOG then computes the set of all related conditional branch instructions in S_{acc}: $\{I_{cond} | I_{cond} \in S_{acc} \cap \texttt{Type}(I_{cond})\texttt{=branch}\}$.

Second, SECLOG performs a semantic pattern matching between the result check function and the paths between I_{acc} and all branch instructions I_{cond} in S_{acc}. This step prunes out infeasible paths that do not match with the operations. Algorithm 1 `OpNodeTreeMatch()` lists the core steps.

- Construct the operation `OpNode` from instructions in S_{acc}. Each instruction in S_{acc} can be mapped to a `OpNode` that includes the instruction (i.e., operation), the operands, and the defined type of the LLVM IR in the SSA style.
- Construct `OpNode` tree backwards from I_{cond}. The root of the tree is the `OpNode` of I_{cond}, while the leaves of the tree can only be I_{acc} or constants types. The `OpNode` tree of the result check function R_{check} can be constructed similarly between the input parameter and R_{check}'s return value.
- Reduce `OpNode` tree. The operations in defined `EQUIV` type like `load, store, cast` can be removed, so that the two `OpNode` trees can be matched in the simplest form.
- Match `OpNode` tree. The final operation matching between the two trees uses the `MatchTree()` procedure in Algorithm 1. The matching algorithm returns None if no match, otherwise it outputs the true or false branch for the branch instruction as the access-deny branch. Note that SECLOG also performs simple logic transformations, to ensure that the branch condition like `rv == -1` can match with specified result check functions like `bool check(rv){return rv < 0;}`. This can be further improved by a formal SAT solvers [31].

Algorithm 1: OpNodeTreeMatch

```
struct {
    instr: the original instruction I;
    operands: operands of I;
    type: Defined type of I, (e.g. EQUIV type includes
      store,load,cast instructions)
} OpNode;
Function OpNodeTreeMatch(I_cond, I_acc, R_check, S_acc):
    Input: I_cond: branch condition instruction;
    I_acc: call instruction of access check function;
    R_check: OpNode tree root of the result check function;
    S_acc: the instructions tained by I_acc;
    ins_map ← {}  ▷ mapping from instruction to OpNode;
    for I_i ∈ S_acc do
        ▷ convert I_i to OpNode
        ins_map[I_i] ← OpNode(I_i, Operands(I_i), Type(I_i))
    root ← ins_map[I_cond]    ▷ root is the tree from I_cond
    root ← ReduceTree(root, ins_map)
    return MatchTree(root, R_check)
Function ReduceTree(root, ins_map):
    if root.type = EQUIV then
        root.operands[0] = ▷ EQUIV only has 1 operand
          ReduceTree(ins_map[root.operands[0]])
    else
        for operand ∈ root.operands do
            ReduceTree(operand, ins_map)
    return root
Function MatchTree(root, R_check):
    r_match ← []; retval ← 0
    if root.type = R_check.type then
        ▷ Recursively match operands of root and R_check
        r_match ← matchAllOperands(root, R_check)
        if None ∉ r_match then
            if Value(root) = ¬Value(R_check) then
                retval = 1 − retval
            return retval
    return None
```

4.1.3 Identify Existing Logs or Add New Ones

To avoid redundant logging, SECLOG finds whether the access-deny log statements already exist inside the ACC function. SECLOG performs static backward slicing from the access-deny return values to search for a logging statement.

SECLOG first searches for logging statements specific to the denied access at the ACC function call site. We count them as specific for the denied access only when the basic block where the log statement lies in the denied branch (identified from our previous step), but does not post-dominate the allowed branch.

An access-deny error may be logged in the upper caller function by propagating the error code. SECLOG records the return value and performs data flow analysis in the call chain to check if the return value is propagated back to its caller in the call chain. SECLOG recursively looks at upper caller function at up to three levels for efficiency considerations.

If no access-deny branches were found, SECLOG can optionally instrument the bitcode by inserting a *checker* by calling the result check function. To avoid affecting the original semantics, the checker was inserted immediately after the basic block where the ACC function was called. Inside the branch, only a log statement is added.

```
vsf_privop_pasv_listen(struct session* p_sess){
  static struct vsf_sysutil_sockaddr* s_p_sockaddr;
  minport = max(1024, tunable_pasv_min_port);
  maxport = min(65535, tunable_pasv_max_port);
  ...
  the_port = random(minport, maxport);
  vsf_set_port(s_p_sockaddr, the_port);
  retval = vsf_util_bind(p_sess->fd, s_p_sockaddr);
  if (vsf_is_error(retval)) {
    die("vsf_util_bind");
  } ...
```

Data dependency

Figure 7: **An example from Vsftpd showing the benefit of collecting relevant information at a caller of an ACC function.** By tracking data dependency at the call site of `vsf_util_bind`, SECLOG adds these port settings in the corresponding access-deny log statements to provide more information for sysadmins.

4.2 Identify What to Log

Based on the characteristics of ACC function, SECLOG identifies the information relevant to the access from two sources: inside the ACC function and at the ACC function's call site.

Inside the ACC function. For each access-deny return value in the ACC function, SECLOG performs backward slicing to find a slice from the access-deny return value backward to the beginning of the function. Then SECLOG extracts all live-in variables [16] (i.e., function parameters, global variables, constants), on which the access-deny return value has a data or control dependency. The variables collected along each deny slice are added separately to represent various reasons for denials (e.g., Figure 3). Those variables are also used in analysis at the call site. If the access check function is a library call (i.e., the source code can not be analyzed), the above analysis is skipped and all function parameters are treated as relevant.

At the ACC function's call sites. ACC function's call sites have specific context information that is useful to guide sysadmins. From the function parameters, SECLOG traces back to the global variables of the configuration settings, and adds them into the corresponding access-deny log statements. (e.g., Figure 7). More specifically, SECLOG collects live-in variables starting from the relevant variables identified inside the ACC function in the previous step. These variables are data-dependent on the relevant variables in ACC function. To collect such information, SECLOG performs a revised backward slicing which only performs the static backward slicing on the data flow graph.

5 Evaluation

We evaluate SECLOG to answer the following questions: First (§5.1), how effective is SECLOG in improving existing access-

Applications	Category	LOC	# ACC func.	# Res. func.	Anno. Time	Analysis Time (mins)
Apache httpd [1]	Web server	199K	18	4	-	45
PostgreSQL [7]	Database	886K	34	2	-	354
vsftpd [11]	FTP	16K	17	3	-	0.2
NFS-ganshea [5]	NFS	165K	9	3	-	14.0
Proftpd [9]	FTP	228K	32	2	-	10.1
Postfix [6]	Mailserver	124K	13	3	72	32.1
HAProxy [3]	Proxy server	167K	5	3	55	4.7
Cherokee [2]	Web server	62K	12	3	32	0.2
Redis [10]	Key-value	134K	5	3	35	0.6
mSQL [4]	Database	35K	3	1	20	0.3

Table 5: **Evaluated applications.** Applications in gray are covered in our study (§2). We added five other software to evaluate the generality of SECLOG. *ACC func*: the number of ACC function. *Res. func*: the number of types of the result check functions. *Anno. Time*: average time in minutes that were spent on annotating the ACC and result check functions. *Analysis time* is total running time for SECLOG to analyze the application. More discussions are in §5.3.

control log statements and identifying missing log locations? Second (§5.2), how effective are SECLOG-enhanced logging statements in helping the administrators in solving real-world access-deny issues? Third (§5.3), how much effort is it for the developers to use SECLOG? Lastly (§5.4), how efficient is SECLOG's analysis and what is the runtime overhead to the server software? The experiments were conducted on a machine with Intel Core i7-7700 CPU (3.6GHz, 8 cores), 16 GB RAM, 1 TB HDD with Ubuntu 16.04.

5.1 Improve Access-Deny Logging

To evaluate the effectiveness of SECLOG in improving existing access-control log statements and identifying missing logging locations, we applied SECLOG on ten server applications and verified the enhanced log messages and new ones.

Appl.	Existing logs		New logs	
	#	# New vars.	#	# Vars.
Apache.	93	5.62	44	7.05
Postgre.	203	2.61	121	4.16
vsftpd	39	1.67	9	2.44
NFS.	8	3.00	20	5.90
Proftpd	145	4.03	111	4.13
Postfix	8	8.75	36	14.28
HAProxy	3	3.33	10	6.40
Cherokee	11	2.18	23	3.35
redis	11	1.82	6	3.00
mSQL	19	2.00	-	-

Table 6: **Evaluation results for SECLOG improvement of existing and new log statements.** This table shows (1) The number of existing log messages enhanced by SECLOG and the average number of additional variables added per existing log statement. (2) The number of newly added log messages by SECLOG and the average number of relevant variables per new log statement.

► **Number of improved log messages.** Table 6 shows the number of existing log messages enhanced by SECLOG and *new* log statements added by SECLOG, for the ten evaluated software applications, and the diagnostic information (represented as numbers of variables) added into the logging statement. SECLOG automatically inserted 6 to 121 log statements into these applications at those access-deny program locations that have no log in the call path. For the existing log messages, SECLOG has added about 1.67-8.75 additional variables per log message. For newly added log messages, there are about 3.0 to 14.28 variables per log message on average.

Appl.	# exi. logs	#(%) covering all vars. in the log	#(%) missing any vars.
Apache	93	73 (79.5%)	20 (21.5%)
Postgre.	203	203 (100%)	0 (0.0%)
vsftpd	39	39 (100%)	0 (0.0%)
NFS.	8	5 (62.5%)	3 (37.5%)
Proftpd	145	144 (99.4%)	1 (0.6%)
Postfix	8	7 (87.5%)	1(12.5%)
HAProxy	3	2 (66.7%)	1(33.3%)
Cherokee	11	11 (100%)	0 (0.0%)
redis	11	11 (100%)	0 (0.0%)
mSQL	19	19 (100%)	0 (0.0%)

Table 7: **SECLOG's coverage of the variables in existing log statements.** The variables in the existing logs are manually picked by developers. We consider a log statement covered by SECLOG only when **all** variables in the log statement are present in SECLOG-identified variables; otherwise it is regarded as missed.

The number of SECLOG-enhanced log statements varies with the number of access-control checks in the software due to (1) software size and (2) the design of ACC code. The larger software like Apache, PostgreSQL and Proftpd contains a higher number of checks. However, in software like redis/NFS-ganshea, the number of ACCs is small despite the software being large in terms of LOC. This may be because the design of ACC code varies based on the nature of each software. The web servers and database servers have many access control privileges and authentication/authorization modes, therefore the access control checks are conducted in many program locations. In contrast, the file system server or the memory cache server (redis/NFS-ganshea) mainly rely on file system protection and have fewer access control checks.

► **Cases confirmed and accepted by developer.** We took a step further to report the SECLOG-enhanced log messages to the open-source projects. Note that some applications (e.g., vsftpd) do not have a formal forum for us to report issues, so we did not report them. We have submitted 114 enhanced log messages as patches to the developers of those applications that have formal forums. When we contact the developers, we prioritized submitting patches to those who responded to our initial patches. So far **70** of 114 have been accepted and merged in their *main branches*.

► **Coverage of variables in existing log messages.** To understand the accuracy of SECLOG-identified variables, we use the variables in existing log messages as the ground truth to compare with SECLOG's outputs. These variables are manually picked by developers. We consider one log message covered by SECLOG only when **all** the existing variables in the log message are in the SECLOG-identified variables; otherwise it is regarded as missed. As shown in Table 7, SECLOG achieves a high coverage in all applications.

We further looked into the log messages with variables *missed* by SECLOG. There are mainly two reasons. **First**, the variables logged by developers are not used in this access-control check. For example, in the 3 cases missed in NFS, developers choose to log the client's IP address when they perform the authentication (i.e., checking credentials). Simi-

Appl.	# exi. logs	Subject		Action		Object	
		before	after	before	after	before	after
Apache.	93	3 (3.2%)	88 (94.6%)	88 (94.6%)	93 (100%)	79 (84.9%)	93 (100%)
Postgre.	203	0 (0.0%)	203 (100%)	0 (0.0%)	152 (74.9%)	203 (100%)	203 (100%)
vsftpd	39	3 (7.7%)	26 (66.7%)	5 (18.2%)	39 (100%)	9 (23.1%)	33 (84.6%)
NFS.	8	0 (0.0%)	8 (100%)	1 (12.5%)	8 (100.0%)	0 (0.0%)	2 (25.0%)
Proftpd	145	0 (0.0%)	90 (62.1%)	145 (100%)	145 (100%)	145 (100%)	145 (100%)
Postfix	8	0 (0.0%)	7 (87.5%)	6 (75.0%)	8 (100%)	1 (12.5%)	4 (50.0%)
HAProxy	3	2 (66.7%)	3 (100%)	3 (100%)	3 (100%)	2 (66.7%)	2 (66.7%)
Cherokee	11	1 (9.1%)	9 (81.8%)	10 (90.9%)	10 (90.9%)	10 (90.9%)	10 (90.9%)
redis	11	0 (0.0%)	11 (100%)	9 (81.8%)	11 (100%)	5 (45.5%)	6 (54.5%)
mSQL	19	0 (0.0%)	19 (100%)	0 (0.0%)	11 (57.9%)	0 (0.0%)	11 (57.9%)

Table 8: **Evaluation results of SECLOG's improvement on the existing log statements.** For each application, we compare the number and percentage of log messages containing subject, action and object **before** and **after** SECLOG's improvement.

Appl.	# new logs	Subject	Action	Object
Apache	44	41 (93.2%)	44 (100%)	44 (100%)
Postgre.	121	121 (100%)	111 (91.7%)	121 (100%)
vsftpd	9	9 (100%)	9 (100%)	7 (77.8%)
NSF.	20	20 (100%)	20 (100%)	20 (100%)
Proftpd	111	102 (91.9%)	111 (100.0%)	111 (100.0%)
Postfix	36	36 (100%)	36 (100%)	31 (86.1%)
HAProxy	10	10 (100%)	10 (100%)	8 (80.0%)
Cherokee	23	18 (78.3%)	23 (100%)	20 (87.0%)
redis	6	6 (100%)	6 (100%)	6 (100%)
mSQL	-	-	-	-

Table 9: **The number and percentage of newly added log messages by SECLOG containing subject, action and object.**

larly, for 7 cases in Apache, developers log the requested URL when the code checks access for requested files. **Second**, the variables logged by developers *mismatched* SECLOG's outputs. For example, for 9 cases in Apache, developers logged the filenames which are missed by SECLOG. This is because the file operation only uses the file descriptor to perform the check. SECLOG could not find the filename previously associated with the descriptor in the upper call chain.

► **Improvements on subject/action/object information.** To understand the improvement of log messages, we further classify the existing variables and newly added variables into subject, action and object based on the characteristics of each application. The results are shown in Table 8 and Table 9 for existing and newly added log statements respectively. For the existing log messages in Table 8, SECLOG can systematically improve the log message with variables related to subject, action and object. For subject, SECLOG can find process id for file accesses, and user info for the SQL queries in different applications; For action, SECLOG can find the access mode or privileges related to the access; For object, SECLOG can find the variables related to files, DB or table names.

However, SECLOG still can not identify all the information related to subject, action or object from the source code because the check may not involve all the attributes. For example, in 25.1% of the cases where SECLOG can not identify the action in PostgreSQL, the code simply checks whether the user is the owner of the object (e.g., table). Similarly, SECLOG can not identify the object in the authentication checks where the check only involves attributes related to the subject.

► **Helpfulness of SECLOG-identified variables.** We conducted a survey to quantitatively evaluate the helpfulness of the information in SECLOG-identified variables in diagnosing access-deny issues. We choose three commonly used applications, Apache, PostgreSQL and vsftpd, and in each application, we randomly selected 10 access-deny program points improved by SECLOG. For each program point, we carefully designed an access-denied scenario that would be denied access at the program point. Then we show the participants with the meaning of the SECLOG-identified variable and the actual value at the access-deny program point. To compare with the helpfulness of SECLOG-identified variables, we randomly selected two additional variables at the access-deny program point (i.e., within the call site of ACC function based on the LLVM IR code). One example question is shown [15]. For each scenario, participants are asked to rate the helpfulness of each variable on a 5-point Likert scale, with 1 as "not helpful" and 5 as "extremely helpful". To avoid overwhelming the respondents with too many questions, only four scenarios are randomly drawn for each respondent. Participants were not compensated for our survey.

Appl.	SECLOG	Random
Apache	3.63	1.43
Postgre.	4.25	1.39
vsftpd	3.88	1.22

Table 10: **Average helpfulness rating of variables in a 5-point Likert scale**, with 1 as not helpful and 5 as extremely helpful.

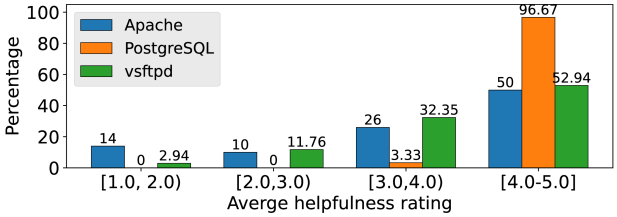


Figure 8: **The distribution of SECLOG-identified information helpfulness ratings.** The rating is on a 5-point Likert scale, with 1 as not helpful and 5 as extremely helpful.

We distributed our survey through the software's mailing list and slack channel, subreddits of sysadmin forums. In total, we received 108 valid responses (36 for Apache, 42 for PostgreSQL and 30 for vsftpd). No personal identifiable information is collected in the survey.

Results. The average helpfulness ratings of SECLOG-identified variables and randomly selected variables are shown in Table 10. In all three applications, the helpfulness ratings of SECLOG-identified are between 3 (moderately helpful) and 5 (extremely helpful), which are significantly higher than randomly selected variables (p<0.005, Mann-Whitney U test).

The distribution of helpfulness ratings in SECLOG-identified variables is shown in Figure 8. For all applications, more than 50% of variables have ratings higher than 4 (very helpful) and more than 75% are higher than 3 (moderately helpful). In PostgreSQL, SECLOG can help identify the accessed table name (object), required privilege (object) and the user who is executing the query (subject) in all the locations, which are all regarded as very helpful.

We further look at the variables with ratings lower than 3 (moderately helpful) in Apache and vsftpd. This is be-

Problem	Description
vsftpd-1	Users could not download the txt file because of misspelled regex patterns in configuration `deny_file`
vsftpd-2	User could not login because the user customized config. file is not owned by root.
vsftpd-3	User could not retrieve directory listing in the passive mode because the allocated port is being used by other application.
Postgres-1	The SQL query is denied because the user lacks `USAGE` privilege on the schema and `INSERT` privilege on the table.
Postgres-2	The SQL query is denied because the user lacks `EXECUTE` privilege to run the functions in the schema.
Postgres-3	The SQL query is denied because the user lacks `REFERENCES` privilege when creating a table with foreign key reference.

Table 11: **Six real-world access-deny issues from vsftpd and PostgreSQL evaluated in our user study.** They were selected from real cases from the online sysadmin forums.

cause for library calls that SECLOG cannot analyze, SECLOG treats all parameters as related to avoid losing useful information, which may include additional less useful information. In Apache, 7 out of 12 variables are the `apr_file_t` object which stores the file handle, which are the parameters of libcalls. Similarly in vsftpd, 3 out 5 variables are introduced as libcall parameters. Future improvements may further prune such variables by adding filters for specific variable types.

5.2 User Study on SECLOG

To evaluate the effectiveness of SECLOG in helping sysadmins fix issues accurately without introducing permission over-granting security mistakes, we conducted a controlled user study. Our study was approved with an IRB exempt status. All our study involving human subjects (i.e., questionnaire survey and user study) were approved by the university's Institutional Review Board (IRB) with an IRB exempt status. We took several measures to protect the participants' rights. First, the researchers who conduct the study were trained with ethics for user research courses before the study. Second, the participants were informed of the purpose, rights and risks before the study and the study is totally voluntary. The participants were informed that they can opt out anytime and they still get compensated for their time in the study. Third, no personal identifiable information is collected or stored during the study.

We crawled the real-world problems related to the software from sysadmin forums (e.g., stackexchange, stackoverflow) and the software's administration mailing list, and further filtered the access control related cases with keywords filtering. We examined the cases and found the common ones as our user study cases. Finally, we used six real-world problems from two server software (vsftpd and PostgreSQL) as shown in Table 11. For example, problem 3 in vsftpd is related to the common port range settings in FTP the server that was posted in many threads [12, 13]. We also use the real-world issue in Figure 1 as problem 2 in vsftpd.

We recruited our participants from each server's user mailing list and Slack workspace, and sysadmins Reddit. In total,

# (%) of insecure solutions	Problem 1		Problem 2		Problem 3	
	Original	SECLOG	Original	SECLOG	Original	SECLOG
vsftpd	1 (6.25%)	0 (0.0%)	6 (37.5%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
PostgreSQL	8 (50.0%)	0 (0.0%)	4 (25.0%)	1 (6.25%)	8 (50%)	0 (0.0%)

Table 12: The number and percentage of insecure fixes in each problem that over-granted permissions and introduced security issues, in the group with original logs and SECLOG-enhanced logs. More details of the insecure fixes are in Table 13.

we recruited 32 work professionals. The participants were compensated for their time with a $30 giftcard.

Methodology. The study is conducted remotely using Zoom for communication due to COVID-19. The participants ssh to a remote server to resolve the designed problems. Each user is asked to resolve six problems in Table 11. We randomly allocate the participants to the group with SECLOG-enhanced log messages (group A) and the group with original software (group B). The **only** difference between the groups is the information in the log messages.

Before the study for each server application, we give the participants 30 minutes to read a detailed guide on how to manage the servers with the specific sections in the official manual related to our study. Then we ask the participant to solve a *warm-up* case for each server program to help them get familiar with the server environment, such as the location of configuration files and commands to modify the settings. The warm-up case is not included in our results.

After the warm-up question, we give them three problems from each server application. The order of the problems is randomized. Each problem is given 30 minutes. If they cannot finish during the given time, we count it as 30 minutes, which penalizes the enhanced group since many participants with the original software could not resolve the problem within the maximum time. We also record how they resolve the problem by recording their commands and modifications, and then evaluate whether the fixes over-grant the permissions and introduce security issues.

After all problems are finished, participants can optionally participate in a short interview. We show the participants both the original and enhanced log messages. We ask the participants whether and how they would change their solutions and reasons for changes; And the participants will rate the usefulness of both log messages on a 5-point Likert scale. One survey question example is shown in [15].

► **Insecure fixes.** We record all the changes made by the participant during the study. We regard the solution as *secure* if the solution only grants necessary privileges to solve the access-deny issue in each problem; If the solution grants excessive privilege to the denied user or other users, it is regard as insecure. For example in PostgreSQL problems, the solution is insecure if the participant grants more privileges than required to execute the original SQL. In problem Postgres-3, only `REFERENCES` privilege is required; however, half of the participants in group with original log messages fix the problem by granting `ALL` privileges on the table to the denied user.

Problem	Solution description	Consequence	Number of occurrences
vsftpd-1	1. Delete other configuration options related to `userlist`	Previous settings limit that only users on `userlist` can login; after the change every user can login.	1
vsftpd-2	1. Delete configuration option `user_config_dir`	All the configurations customized by each user under `user_config_dir` do not take effect.	4
	2. Change permission of config file `/etc/user_config_dir/ftpuser1` to 777	All users in the system can read/write/execute the file.	1
	3. Delete the configuration option `userlist` and other options.	Previous settings limit that only users on `userlist` can login; after the change every user can login.	1
PostgreSQL-1	1. Grant `ALL` privileges on the table to user and grant all privileges on the schema to user.	The user has more privileges than required to execute the query, including `DELETE` or `UPDATE` privileges.	3
	2. Grant `ALL` privileges on the schema to user.	The user has excessive privileges on the schema, e.g., with the `UPDATE` privilege, the user can modify the schema.	3
	3. Grant `ALL` privileges on the table to user.	The user has excessive privileges on the table, e.g., with the `DELETE` privilege, the user can delete the table.	1
	4. Grant `INSERT` on all tables in the schema to user.	The user has `INSERT` privileges on all the tables, so the user can modify other tables.	1
PostgreSQL-2	1. Grant `ALL` privileges on all functions in the schema to user.	The user can execute all the defined functions in the schema.	2
	2. Grant `EXECUTE` on all functions in the schema to user.	The user can execute all the defined functions in the schema.	1
	3. Grant `ALL` privileges on all tables in the schema to user and grant `ALL` privileges on all functions in the schema to user.	The user has all the privileges to the tables and functions in the schema.	1
PostgreSQL-3	1. Grant `ALL` privileges on the table to user.	The user has more privileges than required `REFERENCES` on the table.	8

Table 13: **Description for the insecure fixes for the problems in user study.**

# of unfinished participants	Problem 1		Problem 2		Problem 3	
	Original	SecLog	Original	SecLog	Original	SecLog
vsftpd	1 (6.25%)	0 (0.0%)	9 (56.25%)	0 (0.0%)	4 (25.0%)	0 (0.0%)
PostgreSQL	0 (0.0%)	0 (0.0%)	2 (12.5%)	0 (0.0%)	1 (6.25%)	0 (0.0%)

Table 14: The number and percentage of participants in each problem who cannot resolve the problem in the 30-minute time period in the group with original logs and SecLog-enhanced logs.

This may be because `REFERENCES` is a less common privilege, and the original log messages only tell sysadmins a generic warning of "Permission denied" but did not explicitly tell them what privilege it lacks. As such, some participants just gave random tries like granting `SELECT`/`CREATE` privileges. When it did not work, they just chose to grant `ALL` to resolve the problem.

Table 12 shows the number of insecure fixes (e.g. fixes that over-grant permissions and introduce security issues) introduced by the participants during the study. We find that the group with original log messages (group B) introduces 27 insecure fixes in total. In contrast, the group with SecLog-enhanced log messages (group A) only has one insecure fix (in problem Postgres-2). The result shows that diagnostic information added by SecLog) are effective to reduce the number of insecure fixes by sysadmins.

► **Diagnosis time.** Besides the number of incorrect fixes, we find that with the help of SecLog enhanced log messages, the participants are more likely to have a much faster diagnosis and fix time. As shown in Table 14, all participants in group A completed the task. In contrast, around one-fifth (18/96) of the tasks in group B were not completed within the time limit. For example, 9 out of 16 participants in group B could not finish task vsftpd-2. The original log message only gives a generic warning of "The config file is not owned by root", but does not pinpoint where the config file is, which makes

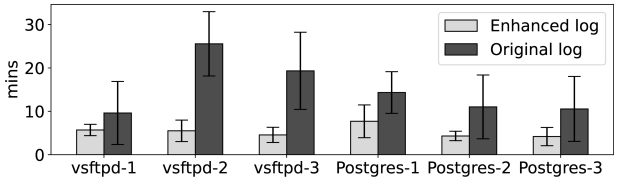


Figure 9: **The comparison of average completion time excluding insecure fix cases.** The error bar represents the 95% confidence interval.

Helpfulness Rating	Original			SecLog-Enhanced		
	Avg.	Median	Distribution	Avg.	Median	Distribution
vsftpd-1	1.80	2		4.36	4	
vsftpd-2	2.28	2		4.64	5	
vsftpd-3	1.32	1		4.80	5	
Postgres-1	2.40	2		4.52	5	
Postgres-2	2.44	2		4.56	5	
Postgres-3	2.16	2		4.60	5	

Table 15: Helpfulness ratings of original and enhanced log messages in the problems. All questions are on a likert scale where 1 is "not helpful" and 5 is "extremely helpful".

the participants mistakenly look into the default config file. Excluding all the insecure fixes, Figure 9 shows that group A is 2.79x faster in completing the tasks than group B. The differences between group A and group B are all significant (p<0.05, Mann-Whitney U test) excluding problem vsftpd-1.

► **Post-study helpfulness survey of whole log messages.** The participants can optionally participate in a short interview to review the original and SecLog-enhanced log messages and rate the helpfulness. One survey question example is shown in [15]. In total, 11 participants in group A and 11 in group B participated in the interview. For participants who chose to grant `ALL` privileges in the PostgreSQL problems,

Workload	# Access	Deny %	Slowdown %	Log size increase %
Course	42K	0.6%	0.0%	0.3%
Group	250K	13.0%	4.7%	23.8%

Table 16: **Performance overhead of Apache httpd with SecLog-enhanced logs.** The data in gray are characteristics of the workload. "Deny" is the percentage of accesses with access-denied status. "Slowdown" is the total execution time to replay the workload; "Log size" is the size of error log.

they noticed that excessive privileges were granted and would choose to give only necessary privilege if they had seen the enhanced message.

25 participants finished the helpfulness ratings of log messages in the six problem on a 5-point Likert scale. The results are shown in Table 15. We find that the enhanced log messages are rated as more helpful than the original messages. In particular, the original log message in problem vsftpd-3 was only rated as 1.32 (close to "not helpful"). The log message only shows "vsf_sysutil_bind". As one participant commented that *"the old log messages were only written so that the original developers of the software would understand."* In comparison, the enhanced log messages give more detailed information about the errors, with a rating of 4.80.

Limitation. The user study has limitations as it is conducted in a controlled environment and users may have different levels of experience and expertise. We tried several things to reduce the bias: (1) We recruited 32 users with relevant server management experiences; (2) We randomly and evenly distributed them to two groups; (3) We provided each with a guide and related manual sections and gave them 30 mins to read these guides; (4) We gave them warm-up questions to get them more familiar with the software and environment.

5.3 Efforts in Adopting SecLog

► **Number of annotations.** We annotated the ACC functions and result check functions in ten open-source server software including web servers, databases, FTP, proxy servers, etc. Table 5 lists the evaluated software. The number of ACC functions in each software is small (3-34), which varies with the application type and code base size. Each ACC function pairs with one result check function, but many ACC functions that have the same format to represent the return value can share the same result check function. In fact, each evaluated software only has 1-4 types of result check functions.

► **Manual efforts in annotation.** Two authors measure the time that they each spent on annotating the ACC functions and the result check functions for five new applications that are not studied in Section 2. As shown in Table 5, the annotation time is usually less than 1 hour since the number of ACC functions is small. We did not measure the time spent on the five applications covered in Section 2 because we studied them extensively to understand their practices.

To further measure the manual efforts in using SecLog, we recruited two graduate students (no co-authors of this paper) to annotate the ACC functions and result check functions. The detailed methodology and results are presented in [15]. From the study, we find that even for developers have less experience and knowledge of the software, they are able to find the ACC functions and annotate the result check functions with high coverage (>90%) in a short amount of time (avg. 25 minutes). The manual efforts required for software maintainers could be even smaller.

► **Efforts in using SecLog's outputs to write log messages.** When the authors submit patches based on SecLog's outputs, the main efforts are spent on understanding the meanings of the variables in the context of each application and correlating them to write log messages; the other efforts are mainly development efforts including finding appropriate logging functions and testing.

To further evaluate the efforts, we conducted a study with three developers to write log messages with SecLog's outputs. The detailed methodology and results are presented in [15]. From this study, we find that the developers try to understand the meaning of SecLog-identified variables based on the code logic and software documentation. The average time to write one log message is 4.33 minutes. The written log messages are syntactically different from the final log messages accepted by the developers, but they are semantically equivalent, i.e., covering the same information (More details in [15]). The efforts spent by maintainers who are familiar with the code logic could be even less.

5.4 Performance and Overhead

We first measure the performance of SecLog's static analysis. The result (Table 5) shows that the analysis time is almost proportional to the number of ACC functions' call sites in the application. We optimize the analysis time by allowing SecLog to re-use the analysis result of functions via summaries. SecLog is single-threaded but it can be further parallelized by analyzing function summaries concurrently.

Next, we measure the performance overhead of Apache httpd's SecLog-enhanced version. All enhanced/inserted log messages have `ERROR` log level which is enabled by default. We use two real-world workloads from a university CS department's websites. `Course` hosts courses pages and `Group` hosts one research group's website. The result is shown in Table 16. (1) For the throughput, we find that on the `Group` workload, the throughput was downgraded by 4.7%. This may because the percentage of the access-denied requests is higher. (2) For the error log size, it increased by 26% on the `Group` workload. We also calculated the average size of access-deny log messages caused. The performance overhead may be larger with a higher percentage of access denial. For potential optimizations, developers may need to improve the common path, i.e., reduce the logging overhead on common types of access-deny errors.

6 Related Work

Improving log messages for bug diagnosis. LogEnhancer [71] and its follow-up works [70, 74] improve logging to collect more diagnostic information for software developers to troubleshoot software bugs. As briefly discussed in §1, we differ from their work in several major ways: (1) They target to collect *intermediate variable* values to help *developers* narrow down the search paths for root cause analysis. Our work focuses on adding/enhancing access-deny log messages for *sysadmins* to understand and fix the reported access-deny issue correctly without over-granting permissions or introducing security issues. As sysadmins do not refer to source code or even have access to source code [67], logging intermediate variables is meaningless to sysadmins. (2) Different goals also lead to different technical challenges and solutions. Their work started from a failure or an error to backtrack important intermediate *control variables* values to help developers reduce the search space, whereas SECLOG starts from access check function call sites, and conduct both backward and forward data and control flow analysis to identify access-deny path, relevant information such as subject, object and action which *determine the outcome of an access check function*. In a way, while we also consider control-dependency, SECLOG looks more into data flow from the result of an ACC function, whereas their work focuses more on control flow analysis to narrow down the possible execution path for developers.

Access-deny issues characteristic study. Xu et al. [67] studied the access-deny issues from a Human-Computer Interaction (HCI) perspective by looking at the practices of how sysadmins resolving access-deny issues from online forums. However, *no solution was proposed and evaluated in their study*. Our work not only provides additional insights to their findings (e.g., Finding 3 on commit history) but also provides a solution to enhance access-deny log messages and insert new log statements.

Access control management. Many access control models have been proposed to cope with different application context, including basic access control matrix [44], discretionary access control [52] for file systems, role based access control for databases [36], attribute based access control for Apache web server [42], and numerous variants [38, 63]. The complex models as well as the different syntax and schema in various software make it hard for sysadmins to manage the policies [18]. Two lines of works have been proposed to help sysadmins in the management process.

The first line of work facilitates proactive access control management with tools and frameworks, such as customizing access control frameworks based on usage context [27, 40, 64], simplifying the implementation of policies [25, 62] and automatically generating secure policies based on security goals [20, 22, 51]. Our work is orthogonal to these works by providing a general approach to improving the access-control logging which aims to help sysadmins gather useful information when an access-deny issue happens.

Another line of work focuses on passively detecting the misconfiguration mistakes by finding inconsistencies in the access control policies. Various approaches have been proposed including data mining [19, 28, 66, 73], testing [48, 49], and verification [37, 43]. These works aim to detect the inconsistencies in the configurations by comparing the configuration against security property from specifications, which are usually provided manually by the sysadmins, or by comparing with other correct systems. In contrast, SECLOG aims to help sysadmins before they make configuration changes — the information enhanced by SECLOG would provide more insights in the troubleshooting process.

Least-privilege policy generation. A line of work has been proposed to use the audit log data to generate least-privilege policies in different environments [24, 50, 54, 55, 57]. Polgen [57] first proposed to generate SELinux policies based on the interaction patterns between different security contexts. Molly et al. propose to mine roles and identify redundant or unused roles from access usages logs with a learning approach [24, 50]. Sanders et al. conducted a series of work to generate least-privilege policies from audit logs with a counting-based model based on time window [53], and further propose Privilege Error Minimization Problem to minimize the over-privilege or under-privilege in RBAC [54] and ABAC models [55]. These works detect the over-privileged policy only *after* the over-privileged access is recorded in the access log. In contrast, we are helping sysadmins to understand and fix unintended access-deny issues correctly in order to *prevent* insecure fixes and over-privileged user accesses.

Empirical studies on security practices of sysadmins. There are a few empirical and HCI studies on some reasons or phenomenons of sysadmin mistakes. Fahl et al. [35] observed that a large percentage of sysadmins who operate on HTTPS websites used non-validating certificates deliberately, because of little tooling support and few affordable certificate options. Dietrich et al. [34] found that besides personal and environmental factors, the systems' poor support can cause misunderstanding and misconfigurations. Li et al. [47] found that sysadmins have faced challenges in updating server programs such as handling update-caused issues and deploying updates without disruptions. Continella et al. [26] identified misconfigurations that could cause unsecured Amazon s3 buckets and suggested stricter default policies and warnings for sysadmins to mitigate this issue. These works motivate for better support and assistance for sysadmins in managing server software. Our work is exactly along this direction.

7 Discussions and Limitations

Generating human-readable log messages. SECLOG can not generate human-readable log messages automatically. Instead, SECLOG can insert variable names and values in the log messages. To make the log messages understandable by sysadmins, the semantics of log messages still need to be pro-

cessed by the developers. Developers can also design utility functions to automatically process the semantics of certain types of variables to readable strings. Moreover, SECLOG can show all the identified locations and variables to help developers to know where to enhance or add access-deny log statements. SECLOG can be used as a plugin in programming IDE and suggest developers with the potential log locations and information. Some recent works [41] may be combined with SECLOG to generate human-readable log messages.

Coverage of ACC locations and relevant information. SECLOG may miss ACC locations if developers do not provide a complete list of ACC functions, or the software performs ad-hoc checks without ACC functions. As discussed in §5.3, the new developers can find ACC functions with high coverage in a short amount of time; for software maintainers, the effort could be less. The developers may also refactor some code to adopt general ACC functions instead of ad-hoc checks.

SECLOG may also miss relevant information that is far in the call chain. However, this should not be common since Finding 2 shows – most (70.8%-100%) of the relevant information is usually available in the same function. Besides, tracking further in the call chain may include too much irrelevant information that can confuse sysadmins.

Security and privacy concerns. Server logs contain valuable information for developers and administrators in the troubleshooting process, which may be leveraged by attackers to gain system details. Therefore, various laws and standards have been proposed to enforce the compliance requirement in log storage, analysis and disposal [21, 39]. SECLOG assumes that the server logs should be kept secure from attackers.

As a static analysis tool, SECLOG cannot differentiate what information is considered sensitive and what should not be visible to sysadmins. Existing works on automatic private information filtering techniques [23, 58, 65, 75] can be combined with SECLOG to filter variables that may potentially leak private user information. In addition, when the SECLOG-identified information is processed by developers, developers can decide whether the information is sensitive and should be included in the log message.

Providing Secure fixes. Even though SECLOG can provide more information for sysadmins to understand unintended access-deny issues correctly, SECLOG cannot solve the problems for sysadmins. This means that SECLOG cannot eliminate the chance of sysadmins making mistakes. As we have shown in our user study, while SECLOG can significantly reduce the number of insecure fixes from 27 to 1, there is still one insecure fix even with SECLOG enhanced log messages.

Logging Templates. Logging templates are adopted in many applications to report error messages for specific type of errors. One potential approach to improve logging quality is to enforce the logging practice via logging template. With the logging templates, developers only need to fill in the variables required by the template to generate the final log message. However, if the template lacks detailed information, the logs will systematically lack information in all places. Therefore, it is promising to improve the logging templates to help developers write better log messages. SECLOG can be used to detect the missing information in the logging templates and developers may use the detected missing information to improve the logging templates, too.

8 Conclusion

This paper focuses on helping developers to improve access-deny log messages for sysadmins to fix access-deny issues correctly and safely. We first conducted a study on five large server programs, to understand the current status of access-deny logging practices. Based on our findings, we designed and implemented a tool called, SECLOG, that can help developers find missing log locations and identify relevant information at the log location. Our evaluation on ten server software and user study show that SECLOG is effective in helping developers to improve the quality of access-deny log messages and reducing the insecure fixes made by sysadmins. SECLOG's source code is publicly available at [14].

References

[1] Apache httpd web server. https://httpd.apache.org/.

[2] Cherokee web server. http://cherokee-project.com/.

[3] HAProxy. https://www.haproxy.com/.

[4] Mini SQL (mSQL). https://hughestech.com.au/products/msql/.

[5] NFS Ganesha File Server. https://fedoraproject.org/wiki/Changes/NFSGanesha.

[6] Postfix. http://www.postfix.org/.

[7] PostgreSQL. https://www.postgresql.org/.

[8] PostgreSQL official docs. https://www.postgresql.org/docs/12.

[9] ProFTPD. http://www.proftpd.org/.

[10] Redis. https://redis.io/.

[11] Vsftpd. https://security.appspot.com/vsftpd.html.

[12] vsftpd passive mode refused. https://serverfault.com/questions/344540/vsftpd-error-500-oops-vsf-sysutil-bind.

[13] vsftpd problem: 500 OOPS: vsf_sysutil_bind. https://www.linuxquestions.org/questions/linux-server-73/vsftpd-problem-500-oops-vsf_sysutil_bind-675699/.

[14] SecLog's source code repo (anonymized for double-blind review). https://anonymous.4open.science/r/AceInstrument-F4BA/, 2022.

[15] Supplementary Materials. https://ucsdopera.github.io/seclog/supplementary.pdf, 2022.

[16] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[17] Bradley Barth. Accentuate the negative: Accenture exposes data related to its enterprise cloud platform. https://www.scmagazine.com/home/security-news/data-breach/accentuate-the-negative-accenture-exposes-data-related-to-its-enterprise-cloud-platform/, Oct. 2017.

[18] Lujo Bauer, Lorrie Faith Cranor, Robert W Reeder, Michael K Reiter, and Kami Vaniea. Real life challenges in access-control management. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 899–908, 2009.

[19] Lujo Bauer, Scott Garriss, and Michael K Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):1–28, 2011.

[20] Matthias Beckerle and Leonardo A Martucci. Formal definitions for usable access control rule sets from goals to metrics. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, pages 1–11, 2013.

[21] Joel Brenner. Iso 27001 risk management and compliance. *Risk management*, 54(1):24–29, 2007.

[22] Seraphin Calo, Dinesh Verma, Supriyo Chakraborty, Elisa Bertino, Emil Lupu, and Gregory Cirincione. Self-generation of access control policies. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, pages 39–47, 2018.

[23] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. *ACM SIGOPS Operating Systems Review*, 42(2):319–328, 2008.

[24] Suresh Chari, Ian Molloy, Youngja Park, and Wilfried Teiken. Ensuring continuous compliance through reconciling policy with usage. In *Proceedings of the 18th ACM symposium on Access control models and technologies*, pages 49–60, 2013.

[25] Yi Fei Chen. Usability of the access control system for openldap. Master's thesis, University of Waterloo, 2019.

[26] Andrea Continella, Mario Polino, Marcello Pogliani, and Stefano Zanero. There's a hole in that bucket! a large-scale analysis of misconfigured s3 buckets. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 702–711, 2018.

[27] Antonio Corrad, Rebecca Montanari, and Daniela Tibaldi. Context-based access control management in ubiquitous environments. In *Third IEEE International Symposium on Network Computing and Applications, 2004.(NCA 2004). Proceedings.*, pages 253–260. IEEE, 2004.

[28] Tathagata Das, Ranjita Bhagwan, and Prasad Naldurg. Baaz: A system for detecting access control misconfigurations. In *USENIX Security Symposium*, pages 161–176, 2010.

[29] Jessica Davis. 63,500 patient records breached by New York provider's misconfigured database. https://www.healthcareitnews.com/news/63500-patient-records-breached-new-york-providers-misconfigured-database, Apr. 2018.

[30] Jessica Davis. Long Island provider exposes data of 42,000 patients in misconfigured database. https://www.healthcareitnews.com/news/long-island-provider-exposes-data-42000-patients-misconfigured-database, Mar. 2018.

[31] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[32] Safety Detectives. Australian sports fan portal leaks 132GB of private data. https://www.safetydetectives.com/blog/bigfooty-leak-report/, 2020.

[33] BOB DIACHENKO. Document Management Company Left Credit Reports Online. https://securitydiscovery.com/document-management-company-leaks-data-online/, 2019.

[34] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. Investigating system operators' perspective on security misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1272–1289, 2018.

[35] Sascha Fahl, Yasemin Acar, Henning Perl, and Matthew Smith. Why eve and mallory (also) love webmasters: a study on the root causes of ssl misconfigurations. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 507–512, 2014.

[36] David Ferraiolo, D Richard Kuhn, and Ramaswamy Chandramouli. *Role-based access control*. Artech house, 2003.

[37] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, pages 196–205, 2005.

[38] Eric Freudenthal, Tracy Pesin, Lawrence Port, Edward Keenan, and Vijay Karamcheti. drbac: distributed role-based access control for dynamic coalition environments. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 411–420. IEEE, 2002.

[39] J Frields. National industrial security program. operating manual supplement. Technical report, DEPARTMENT OF DEFENSE WASHINGTON DC, 1995.

[40] Mansura Habiba, Md Rafiqul Islam, and ABM Shawkat Ali. Access control management for cloud. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 485–492. IEEE, 2013.

[41] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R Lyu. Characterizing the natural language descriptions in software logging statements. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 178–189, 2018.

[42] Vincent C Hu, D Richard Kuhn, David F Ferraiolo, and Jeffrey Voas. Attribute-based access control. *Computer*, 48(2):85–88, 2015.

[43] Karthick Jayaraman, Vijay Ganesh, Mahesh Tripunitara, Martin Rinard, and Steve Chapin. Automatic error finding in access-control policies. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 163–174, 2011.

[44] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[45] Butler W Lampson. Computer security in the real world. *Computer*, 37(6):37–46, 2004.

[46] Richard Lawler. Capital One data breach affected 100 million in the US. https://www.engadget.com/2019/07/29/capital-one-data-breach/, Jul. 2019.

[47] Frank Li, Lisa Rogers, Arunesh Mathur, Nathan Malkin, and Marshini Chetty. Keepers of the machines: Examining how system administrators manage software updates for multiple machines. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[48] Evan Martin and Tao Xie. Automated test generation for access control policies via change-impact analysis. In *Third International Workshop on Software Engineering for Secure Systems (SESS'07: ICSE Workshops 2007)*, pages 5–5. IEEE, 2007.

[49] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 667–676, 2007.

[50] Ian Molloy, Youngja Park, and Suresh Chari. Generative models for access control policies: applications to role mining over logs with attribution. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 45–56, 2012.

[51] Qun Ni, Jorge Lobo, Seraphin Calo, Pankaj Rohatgi, and Elisa Bertino. Automating role-based provisioning by learning from examples. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 75–84, 2009.

[52] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted computer system evaluation criteria. In *National Computer Security Center*. Citeseer, 1985.

[53] Matthew Sanders and Chuan Yue. Automated least privileges in cloud-based web services. In *Proceedings of the fifth ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, pages 1–6, 2017.

[54] Matthew W Sanders and Chuan Yue. Minimizing privilege assignment errors in cloud services. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 2–12, 2018.

[55] Matthew W Sanders and Chuan Yue. Mining least privilege attribute based access control policies. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 404–416, 2019.

[56] Sara Sinclair and Sean W Smith. What's wrong with access control in the real world? *IEEE Security & Privacy*, 8(4):74–77, 2010.

[57] Brian T Sniffen, David R Harris, and John D Ramsdell. Guided policy generation for application authors. In *SELinux Symposium*, 2006.

[58] Kunal Taneja, Mark Grechanik, Rayid Ghani, and Tao Xie. Testing software in age of data privacy: A balancing act. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 201–211, 2011.

[59] Biometric Update. Biometrics company allegedly leaves unhashed fingerprint data of thousands exposed to internet. https://www.biometricupdate.com/202003/biometrics-company-leaves-unhashed-fingerprint-data-of-thousands-exposed-to-internet, 2020.

[60] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[61] Verizon. 2020 Data Breach Investigations Report. https://enterprise.verizon.com/resources/reports/2020-data-breach-investigations-report.pdf, 2020.

[62] Artem Voronkov. *Usability of Firewall Configuration: Making the Life of System Administrators Easier*. PhD thesis, Karlstads universitet, 2020.

[63] Guojun Wang, Qin Liu, and Jie Wu. Hierarchical attribute-based encryption for fine-grained access control in cloud storage services. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 735–737, 2010.

[64] Hua Wang, Yanchun Zhang, and Jinli Cao. Access control management for ubiquitous computing. *Future Generation Computer Systems*, 24(8):870–878, 2008.

[65] Rui Wang, XiaoFeng Wang, and Zhuowei Li. Panalyst: Privacy-aware remote error analysis on commodity software. In *USENIX Security Symposium*, pages 291–306, 2008.

[66] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. Towards continuous access control validation and forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 113–129, 2019.

[67] Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. How do system administrators resolve access-denied issues in the real world? In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 348–361. ACM, 2017.

[68] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244–259, 2013.

[69] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 249–265, 2014.

[70] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 293–306, 2012.

[71] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):1–28, 2012.

[72] ZDNet. Database leaks data on most of Ecuador's citizens, including 6.7 million children. https://www.zdnet.com/article/database-leaks-data-on-most-of-ecuadors-citizens-including-6-7-million-children/, 2019.

[73] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 687–700, 2014.

[74] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 565–581, 2017.

[75] Rui Zhou, Mohammad Hamdaqa, Haipeng Cai, and Abdelwahab Hamou-Lhadj. Mobilogleak: A preliminary study on data leakage caused by poor logging practices. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 577–581. IEEE, 2020.

A Appendix

A.1 Additional Observations

We performed additional studies on the five real-world server applications used in Section 2. These observations are helpful to design of SECLOG but may not provide additional insights to understand the access-deny logging for the general audience. We append the observations here for reference.

Application	# ACC function	# Call sites
Apache HTTPD	18	141 (79.2%)
PostgreSQL	34	319 (66.3%)
Vsftpd	17	52 (73.2%)
NFS-ganesha	15	40 (65.6%)
Proftpd	32	256 (100%)

Table 17: The number of access control check (ACC) function and the number of access-check program points that are identified by the ACC functions.

Observation 1: *A small (15-34) number of ACC functions are used to perform access checks in many (40-319) program locations for different kinds of purposes (Table 17).*

The studied applications commonly use ACC functions to perform checking in modules related to network, files, or authentication and authorization. These ACC functions are called in the majority (65.6%-100%) of access check program points. Only in a small number of cases, the application may perform ad-hoc checks for certain operations without directly calling an ACC function. For example, Vsftpd directly checks whether the anonymous user is allowed against server configurations in the initialization phase. ACC functions are called in many different locations of the source code. On average, one ACC function has 6.1 call sites. For example, `pg_class_aclcheck` in PostgreSQL which is used to check whether the user has certain privileges to access a table, has 42 different call sites in its source code. Without tooling support, large number of access check locations make it challenging for developers to ensure good logging at every single point.

Application	Has log at caller	Has log in ACC func.	w/o logs
Apache HTTPD	100 (58.1%)	15 (8.7%)	72 (41.8%)
PostgreSQL	211 (66.1%)	100 (31.3%)	108 (33.9%)
Vsftpd	35 (67.3%)	8 (15.4%)	10 (19.2%)
NFS-ganesha	14 (35.0%)	1 (2.5%)	26 (65.0%)
Proftpd	146 (57.0%)	8 (3.1%)	110 (43.0%)

Table 18: The number of access-check program points identified by the ACC function where there are log statements at the function's call site, inside the function, or no log statements.

Observation 2: *Comparing all logging locations in the source code, logging at access control check function's call site is more common than other locations in source code (Table 18).*

We zoom in to understand where the log messages are placed in today's practice. Developers usually place log messages at the ACC function's call sites instead of inside the ACC function itself. The main advantage of placing messages at a call site is that it can log more relevant information. Only in a few cases, the developers choose to add log messages only inside the access control check function. Note that in PostgreSQL, all the ACC functions have log messages inside the function, but there are also logged messages at the function's call site. If SECLOG follows the common practice to place the logging statements at the call site of ACC functions, SECLOG can reuse most of the existing access-deny logging statements and avoid some performance overhead.