Let the Right One In: Discovering and Mitigating Permission Gaps

Beng Heng Ng and Atul Prakash

University of Michigan, Ann Arbor MI 48105, USA,
{bengheng, aprakash}@eecs.umich.edu

Abstract. Permissions that are granted but unused, or *permission gaps*, represent needless risks that systems are subjected to and should be removed expeditiously. More insidiously, when left unchecked over time, granted permissions may not always be revoked when accesses to the objects are no longer required. In practice, identifying permission gaps can be hard since for many system services, a second reference point other than granted permissions is usually unavailable. We argue that in most cases permission gaps must therefore be estimated. We propose DeGap, a framework that uses standard system logs as a reference point and a common logic for estimating permission gaps in various system services. We present the results from DeGap when it was used to identify gaps in SSH server configurations, file permissions, and user groups. DeGap identified potentially overly relaxed SSH server configurations, incorrect permissions on security-relevant files, and dormant user groups. Of course, just discovering permission gaps may not be enough; administrators need to know how to fix them by modifying configuration files. DeGap can also identify specific changes to service configurations that help to reduce permission gaps.

1 Introduction

Permissions for system objects (e.g., files and network sockets) serve as a first line of defense against misuse by unauthorized users. Ideally, only permissions needed by legitimate users to access objects are granted. Unfortunately, this can be difficult to achieve in reality. Granted permissions that are neither needed nor used, and called *permission gaps*, often arise due to administrator oversights or new software, which is less scrutinized. The gaps can persist undetected and be exploited [1–3].

One method for identifying potential permission gaps is statically analyzing a system. In several works on Android apps [4–6], the authors compared permissions requested by an Android app *from* the user with permissions that could be used by the app based on static code analysis. Unfortunately, even legitimately granted permissions could be exploited by malware by, say, code injection. Thus permissions, which are granted but not actually used, should be removed. Besides, using static analysis to identify gaps is not broadly applicable. In [6], a firewall with too liberal permissions is used to motivate the need for closing permission gaps. But, static analysis of services listening on the ports and comparing them with firewall permissions is unlikely to identify the gaps because,

usually, only the firewall is responsible for blocking the remote requests. There is only one reference point, i.e., the firewall policy, to compare with for identifying potential gaps.

Furthermore, any permission gap analysis is likely to be computing an *estimate* of a gap. Security requirements change over time and may not even be precisely known. For example, an administrator may miss out revoking the access right of a user who has left the job. Even in the static analysis example, there could be a rare app that wants to acquire extra permissions from users, anticipating a future capability in the app.

The above leads to the question: what is the best that we can do in an automated manner, given that permission gaps exist in systems but static analysis is infeasible and any analysis must necessarily be an estimate? We propose a framework, DeGap, that analyzes potential permission gaps using a common logic for different services. DeGap can determine a lower bound on the set of permissions for an object that is consistent with its usage during a selected period. These permissions are then compared with the assigned permissions to expose potential permission gaps. DeGap then proposes a set of suggested configurations for achieving that lower bound. While we demonstrate the feasibility of our framework in the setting of system objects, it should be readily applicable to other scenarios, such as improving privacy settings in social networks. Overall, we make the following core contributions:

- We describe DeGap's core components while highlighting the design nuances for components that need customization for different services. We elaborate the common logic used for analyzing permission gaps.
- We implemented DeGap and show that it is able to detect permission gaps for three scenarios: `SSHD`, file permissions, and user groups in `/etc/group`. For `SSHD`, DeGap found two users had not had their access revoked even though it was no longer required. Also, DeGap found that legitimate users only used public key authentication to log on to a server, but password-based authentication was unnecessarily allowed. The server had been the target of password brute-force attacks. Thus, revoking the password-based authentication method was a low-impact means of tightening the authentication policy and enhancing security. For `auditd`, DeGap discovered a private key for one service that was mistakenly set to be world-readable. It also found two additional files on the servers that could be exploited to execute a privilege escalation attack. Finally, DeGap identified various user groups that were unused during monitoring and were candidates for tightening.
- DeGap was able to automatically suggest changes that can be made to configurations for reducing permission gaps for all three scenarios.

The paper is structured as follows. Section 2 provides the related work. Section 3 discusses the limitations. Section 4 presents definitions and basic techniques that are used as building blocks for for discovering gaps and identifying fixes. Section 5 describes the system architecture. Section 6 presents evaluation results. Finally, Section 7 presents conclusions and possible future work.

2 Related Work

The concept of permission gaps has been discussed previously in works on permission gaps for Android applications by Felt et al. [4], Au et al. [5], and Bartel et al. [6]. The phrase "over-provisioning of permissions" has also been used to describe a similar notion [7]. The key distinction between DeGap and these works is that in these works, the application is considered to be the subject, G is the set of authorizations requested *from* the user during application installation, and R is an upper bound on the set of permissions usable by the application, based on static code analysis (which tends to be exhaustive). DeGap examines a different scenario where R is incomputable by static analysis. DeGap could still be useful in the context of Android applications. An application could make use of more restricted permissions than it actually does. In that case, log analysis with DeGap may identify opportunities for tightening permissions within the code.

DeGap is not an access control mechanism; it complements existing access control mechanisms by providing a means to verify the correctness of access control policies. However, a brief review of existing access control mechanisms may be useful. Access control models, such as Role-Based Access Control (RBAC), Discretionary Access Control (DAC), and Mandatory Access Control (MAC), have been proposed. Access control is typically enforced with a reference monitor that mediates every access by a user to an object in the system [8, 9]. In practice, perhaps SELinux [10] and AppArmor [11] are the most widely accepted implementations of MAC. Both systems provide monotonic security, which is also a goal of DeGap. However, these mechanisms adopt a different philosophy from DeGap; SELinux and AppArmor begin by restricting accesses to objects and relaxing the limits when needed, while DeGap aims to identify accesses that are no longer required and then restricting these accesses.

Other efforts to improve a system's permissions resulted in integrated tools that perform functions beyond permission checking, such as vulnerability analysis. These tools, which include COPS [12], Tiger Analytical Research Assistant [13], and Bastille Linux [14], return a list of files and directories with world-readable or writable permissions. This is also achievable using the `find` command on Linux-based systems [15]. We argue that the challenge is not in finding such a list, since it is likely to be huge. Instead, the fundamental problem that we are solving with DeGap is in determining if permissions are indeed unused.

3 Limitations

DeGap uses logs, which record past object accesses over the analysis period. Clearly information about prior accesses provides no guarantee regarding future access patterns. But in the absence of the ability to accurately predict future uses and despite the possibility of false positives, DeGap provides a means for identifying potential permission gaps so long as access patterns remain unchanged.

An adversary may have accessed an object prior to or during log analysis. DeGap is not able to distinguish legitimate accesses from illegitimate ones.

Illegitimate accesses can be filtered out if they are known prior to analysis, for example using reports from an intrusion detection system (IDS). DeGap provides a database and a query engine that allows the exclusion of known illegitimate accesses. For unknown illegitimate accesses, an attacker's activity will be treated as normal; in that case, DeGap is still useful in tightening the system to prevent other attacks that did not occur during the monitored period.

Since accesses are logged after they have occurred, DeGap is by design incapable of preventing illegitimate accesses as they happen. Without reinventing the wheel, we leave this responsibility to existing tools such as IDS. The same limitation applies to other security alerting systems [16], such as Tripwire, a system that detects changes to file system objects [17].

4 Tightening Permission Gaps

We now define the notion of permission gap and its usage in DeGap. Using the same terminologies as Lampson [18], *subjects* access *objects* using a set of *rights*. We represent a *permission* as a tuple (s, o, r) that denotes subject s as having the right r to access object o. In turn, system objects are collectively guarded by a set of *granted permissions* $G = \{g_1, g_2, \ldots, g_n\}$, where each g_i is a permission tuple (s_i, o_i, r_i). A subset $TrueGap$ of granted permissions G are not required for all legitimate accesses to succeed. This subset constitutes the *true permission gap* for the system. These extraneous permissions potentially increase the attack surface (elaborated in Appendix A) of the system, and the goal is to help users discover these gaps and recommend ways of fixing them.

The problem is that computing $TrueGap$ is generally not possible for an automated tool in the sense that the precise set of legitimate accesses that should be allowed normally cannot be automatically inferred if the only reference point available is the set of granted permissions. A second reference point is needed to compute an estimate of the gap.

We attempt to compute an estimate of the $TrueGap$ by using past accesses, usually recorded in system logs, as a second reference point that can be compared with G. Let R be the set of permissions that were appropriately authorized, as per log files. We define the notion of *permission gap*, P, to be $G \setminus R$. '$\setminus$' denotes set subtraction. There is no permission gap if and only if $P = \emptyset$.

In a static system (where G and $TrueGap$ do not change), the permission gap P from the above definition will be be an upper bound on $TrueGap$. How tight this bound is will generally depend on both the quality of the log files and the period of time that they cover. The approach we take is that since $TrueGap$ must be estimated, a reasonable choice, as good as any available for most services, is to estimate the gaps based on past usage. At least, that way, users have some well-defined reference point when deciding whether to tighten permissions. Using this definition also permits users to ask the question, "*Given a permission that is proposed to be revoked, were there past actions that would have been denied had the permission never been granted in the first place?*"

Answering the preceding question is important before revoking permissions as it may be indicative of denial-of-service problems caused by the revocations.

Using a firewall analogy, if blacklisting a sub-domain of IP addresses is proposed as solution to prevent attacks from a subset of nodes in that domain, it is important to consider whether there have been legitimate accesses from that sub-domain in recent past. If there were, then blacklisting the entire sub-domain may be unacceptable.

Our approach of using best-effort estimation of gaps is consistent with the approach in other areas of applied security. IDS and virus detection systems do not always guarantee correctness, but they are still useful to administrators as an aid in securing systems.

We used a model of subjects, objects, and rights to express permissions. Without loss of generality, the approach could be extended to support more detailed models of permissions, e.g., where subjects, objects, and rights have attributes and granted permission is viewed as a combination of approved attributes of subjects, objects, and rights. The key requirement is that there be a way to compute $G \setminus R$, given G and R.

4.1 Gap Analysis and Traceability

Our goal is not only to determine whether a permission gap exists, but exactly the setting in a configuration file or the permission on an object that contributes to the gap, i.e., the question of mapping $G \setminus R$ back to specific configuration settings; ideally, the administrator should be told the specific setting that contributes to the gap without guessing. This is particularly important because gaps are likely to be reported at a lower abstraction level by the tool than what an administrator is accustomed to.

The challenge in identifying the settings in a configuration file that are candidates for tightening is that a reverse mapping from P to a configuration setting may not always be available or straightforward to provide. For example, if logs indicate that remote root login to `SSHD` is not required, identifying the specific place to make the change (e.g., `PermitRootLogin` field or the `AllowUsers` field) will require a fair amount of domain knowledge. One potentially has to write two parsers, one to go from a configuration file to G and another from gaps to specific settings in a configuration file.

DeGap supports two approaches to the problem for discovering changes in configuration settings that lead to reducing the permission gap. In the first approach, the existence of a reverse map from gaps to configuration settings is assumed to be available for a service and can be provided to DeGap. We used this approach first for analyzing gaps with file permissions using logs from `auditd`.

In the second approach, the existence of a reverse map from gaps to configuration settings is assumed to be unavailable. We only require the availability of a one-way transform from configuration settings to G. We used this approach analyzing gaps in configuration settings in sshd_config using `SSHD` authentication logs and later also applied it to `auditd` logs. The second approach requires less work in applying DeGap to a new service since a reverse map from gaps to configuration settings does not have to be defined. However, the first approach can sometimes be more efficient. We will primarily discuss the second approach

in the paper since we have found it to work sufficiently well in practice that the extra work of creating a reverse map is probably not worthwhile.

In the second approach, to identify changes in settings that could reduce the gap, the basic idea is to generate potential deltas to a configuration file that tighten permissions and map the modified configuration files to a set of granted permissions G'. We then determine if G' helps close the permission gap with respect to a set of required permissions R.

A permission gap P, where P is $G \setminus R$, can be tightened by restricting G to G' where $G' \subset G$. This involves eliminating permissions. To eliminate the gap P completely, one must choose $G' = G \setminus P$. Realistically, this is not always possible due to limits on the granularity of the granted permissions that can be influenced by the configuration settings. For example, for Unix file permissions, changing the `other` mode bits will impact all non-owner and non-group users.

Removing permissions by changing configuration settings leads to three possible outcomes: under-tightening, over-tightening, or both. Over-tightening affects usability; some rights in R would have been denied. Under-tightening exposes the system to accesses that are not in R. For a static system (where G has not changed over the logging period), we claim the following propositions:

Proposition 1. (Over-Tightening Rule) *G' is over-tightened with respect to required permissions R if and only if $R \setminus G' \neq \emptyset$.*

Proposition 2. (Under-Tightening Rule) *G' is under-tightened with respect to required permissions R if and only if $G' \setminus R \neq \emptyset$.*

Proof. First, let G' be an over-tightened permissions configuration. This is equivalent to saying that there exists some permission r that has been requested but not granted; that is, $r \in R \setminus G'$ and hence $R \setminus G' \neq \emptyset$.

Similarly, let G' be an under-tightened permissions configuration. This is equivalent to saying that there exists some permission g that is granted but never requested, so that $g \in G' \setminus R$ and hence $G' \setminus R \neq \emptyset$. □

It is possible for a configuration change to result in G' that is both over-tightened and under-tightened with respect to R. This could theoretically occur if the configuration change results in revocation of multiple permissions that are insufficient to close the gap, but some of the revoked permissions are in the set R. For Unix files, removing group permissions could result in such a situation.

The over-tightening rule is a simplification of the reality. Since G is really a snapshot of granted permissions, it is possible that $R \setminus G$ is not empty – permissions could have been tightened during or after the logging period, because of a change in security requirements and removal of some objects, but before a snapshot of G was taken. If so, $(R \setminus G') \subset (R \setminus G)$ should hold. If true, then the configuration change contributes to reducing the permission gap between G and R; otherwise not. We accommodated this scenario by *normalizing* R (replacing it with $R \setminus G$) thereby removing requests in the log that pertain to objects that no longer exist or permissions that administrators have revoked. Once R is normalized, the over-tightening and under-tightening rules continue to apply.

To test if a configuration setting contributes to the permission gap, we can simply simulate a change to the setting to obtain G', and compute $R \setminus G'$. If

$R \setminus G' = \emptyset$, the setting contributes to the permission gap. On the other hand, if new tuples show up in $R \setminus G'$, that means that changing the setting caused some actions in R to be denied. The setting does not contribute to the permission gap. Additionally, we check that $G' \setminus R = \emptyset$. Otherwise, some of the previously granted subjects will be denied access with the changed setting.

In general, if there are n possible atomic deltas to an existing configuration setting, it will require $O(n)$ checks to identify all the deltas that can lead to tightening the gap without over-tightening. The above idea also applies to finding potentially stale members in a group. As an example, for `SSHD`, the `AllowUsers` field specifies a list of authorized users. To detect users in the list that could be contributing to a permission gap, one simply needs to simulate removing each user one by one and, for each G', determine if $R \setminus G' = \emptyset$; if yes, the user's authorization was not used for login and the user is a candidate for removal from `AllowUsers`. For `auditd` logs, we can apply a similar technique to periodically analyze membership lists in `/etc/group` for key groups for potential pruning.

5 System Architecture

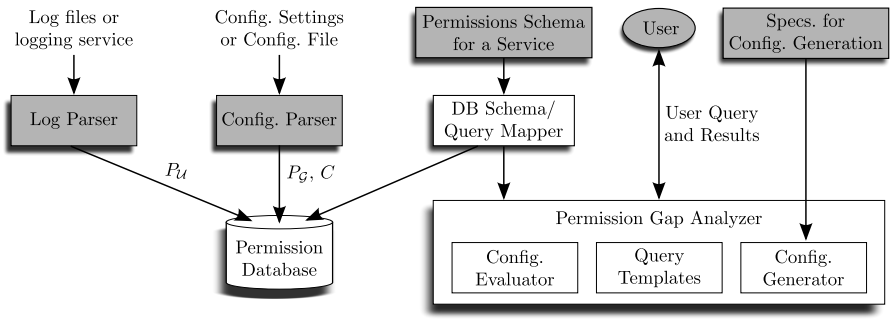

Fig. 1. Conceptual model for DeGap. Arrows indicate dataflow. Shaded components are specific to the service being analyzed.

Figure 1 shows the architecture of DeGap. Shaded boxes in the figure are specific to each service being analyzed. Bulk of the gap analysis system and the database is automatically generated from schemata that describe the attributes of subjects, objects, rights, and format of the service configuration files.

A Log Parser extracts permissions used from either log files or directly from a service logging facility and system activities and puts it in a database. For services, writing logs to a file is usually more efficient than writing to the database on the fly [19]. Most services can log to files out of the box. Thus, we adopt a hybrid approach where we use log files to record object accesses but post-process the logs to store required permissions, R, to a database so as to permit a general-purpose query engine for analyzing permission gaps.

The Config. Parser is a service-specific component that generates the set of granted permissions G. If configuration files are used to define granted permissions (e.g., `SSHD`), it reads in configuration files and generates G. For others, like Unix file permissions, it queries the system for the information. Config. Parser also extracts a sequence of configuration settings, C, that are used by Config. Generator for discovering settings towards reducing permission gaps.

The DB Schema/Query Mapper is similar in spirit to tools like Django [20] in that it uses an input schema to generate initial tables for the database. We also use it to generate SQL queries from *query templates*, which provide simple ways of querying the database using highly expressive select-project-join (SPJ) queries [21] on entities and attributes in the configuration schema. Full SQL is also available. Users who wish to extend DeGap or make custom queries can either use templates or full SQL. The Config. Generator and Config. Evaluator internally make use of templates as well for direct queries on the database.

Config. Evaluator takes as input candidate configuration file (or settings) that a user would like to evaluate against R, the requests in the log files. It reports back whether the candidate configuration leads to a system that has a narrower gap than G without over-tightening.

Config. Generator uses specifications for configuration parameters (described in Section 5.1 and in Figure 2) as input and generates tighter alternatives for configuration settings. It then evaluates those modified settings using the Config. Evaluator to determine whether they tighten the permissions without over-tightening them. The subset of alternatives that are acceptable are presented to the user. Config. Generator can also apply a greedy algorithm (presented in Section 5.1) to generate a sequence of configuration setting changes that is *maximal*; tightening any remaining setting further leads to an over-tightened system.

5.1 Permission Gap Analyzer

For DeGap to be useful, users should be able to evaluate the effect on permissions when a configuration setting is changed. Towards this end, the Permission Gap Analyzer (PGA) provides the following capabilities:

- *Evaluate a proposed configuration setting change:* The gap analyzer uses Over-Tightening Rule and Under-Tightening Rule (see Section 4) to help evaluate a proposed configuration setting with respect to R. Over-tightening of permissions could lead to denial-of-service for legitimate users.
- *Generate a list of candidate configuration changes:* The gap analyzer uses the specifications for configuration generation to automatically iterate through possible one-step changes to configuration settings and identify the settings that can be tightened to reduce the gap.
- *Generate a sequence of configuration changes:* The gap analyzer can identify a full *sequence* of changes to a configuration file that provides a maximal solution to gap reduction. The solution is maximal and not necessarily optimal in that any further tightening will lead to over-tightening. But, there could be other sequence of changes that are longer in length or result in lower gap with respect to some objects. In general, finding an optimal sequence is likely

```
field = {type = <type>, values = <value 1> | <value 2> | ... | <value n>,
         default = <default>}

PermitRootLogin = {type = oneof,
             values = yes | without-password | forced-commands-only | no,
             default = yes}
AllowUsers = {type = setof(subject), default = *}
```

Fig. 2. Configuration specification format and examples for `PermitRootLogin` and `AllowUsers`.

to require exponential time as given n possible individual tightening steps, there can be 2^n combinations of tightening steps. We therefore focus on finding a maximal sequence rather than an optimal solution. Given that we are working with estimates of permission gaps, we believe this is a reasonable and practical strategy.

In the algorithms that follow, we use G to refer to granted permissions, R to refer to required permissions (these are derived from logs), and G' to refer to a candidate for granted permissions. C and C' refer to the combinations of configuration settings that result in G and G' respectively.

We discuss the above features of permission gap analyzer below. Gap analyzer also permits users to make direct queries on the database via query templates, as discussed in Section 5.2. They are a convenience feature to support specific queries on the permissions database and to allow extensibility.

5.1.1 Config. Evaluator The Config. Evaluator E takes in C', G, and R, then computes the permissions G' that would be granted by configuration C'. It compares G' with G and R, and returns three possible results: `tighter`, `tightest`, and `bad`. C' is `bad` if G' is not a subset of G, or at least one permission in R is denied by G'. Otherwise, G' is tighter than G. To check if G' is the tightest possible set of permissions, in addition to the two conditions, we check that G' is not being over-tightened and not being under-tightened, i.e., $G' \setminus R \equiv \emptyset$ and $R \setminus G' \equiv \emptyset$. The algorithm for E is shown in Figure 7 in Appendix B.

5.1.2 Automatically Generating Alternative Configurations To assist PGA in generating alternative configurations, a user specifies the format of a configuration file, along with choices for each field, as shown in Figure 2. The current version of the system views configuration files as a sequence of fields, where each field can either have (i) a single value from a domain or a set of values, or (ii) a sequence of values from a domain. This can obviously be generalized to allow nested fields, but is adequate as a proof-of-concept.

Each specification has three parts: `type`, `values`, and `default`. Type is specified as either `oneof` or `setof`. We found these two types sufficient for analyzing permissions for `SSHD` and `auditd`, but more types can be easily added. If `type` is `oneof`, the `values` part is mandatory. It specifies the possible values for the configuration in increasing order of tightness and is delimited by '—'. In Figure 2, `PermitRootLogin` is specified as having `type oneof` and can take four `values`,

`yes`, `without-password`, `forced-commands-only`, and `no` in order of increasing strictness. It is specified to have a default value of `yes`, thus, if `PermitRootLogin` is not present in the `SSHD` configuration file, the value `yes` will be used.

If the `type` is `setof`, the configuration takes a set of values, such as `AllowUsers` for `SSHD`. These configurations typically specify a range of values for two categories, subjects, or objects. Specifying a range of values for access rights is rare but possible. If a configuration affects only a subset of a certain category, computing permission gaps for permissions excluding those affected by the category is redundant. Towards optimization, we allow the user to augment the `type` with either `subject`, `object`, or `right`. Using this annotation, PGA pre-filters the permissions for the specified category for computing operations such as $G \setminus G'$. We will elaborate on `type setof` shortly.

The Config. Generator is used for tightening a single configuration setting from a given state. It provides a Python method *restrict_setting*(c) that generates a more restricted value for a field c. Repeated calls to the function return subsequent choices for that field (when choices are exhausted, the function returns None, which is equivalent to False in conditionals in Python). This function is used to automatically find the list of all possible configurations that only tighten a setting as well as a maximal solution of a sequence of configuration tightening steps using a greedy algorithm. The function *restrict_setting*(c) works as follows for the two types of fields that are currently supported:

Type `oneof` fields: Using the Config. Generator Specification, the function *restrict_setting*(c) returns the next (tighter) value for a field c in a configuration. If a configuration `type` is `oneof`, the generator returns the value that follows the current one. For example, if the current value for `PermitRootLogin` is `without-password`, then the function returns `forced-commands-only` on the first call, `no` on the second call, and finally Python's `None`.

Type `setof` fields: If the configuration `type` is `setof`, every time the function *restrict_setting*(c) is called to restrict a set, it removes one element in the set that has not been previously removed. For example, if `AllowUsers` were "user1, user2, user3", it would generate the following sequences: "user2, user3", "user1, user3", "user1, user2" and Python's None. Note that it only removes one value from the current list and does not generate all subsets. So, the procedure is linear in the size of the set.

In practice, sets can have special values such as `*`, which denote all possible values from a domain that are difficult to enumerate and apply the above strategy. For example, in `sshd_config`, if `AllowUsers` is missing, the default value for that can be considered to be `*`. The question then is how we represent G and generate alternatives for tighter configurations when the values of an attribute cannot be feasibly enumerated? To address the problem, we use a projection on R on the appropriate domain as the initial recommendation for the set of all granted permissions. There is no loss of generality since DeGap is concerned with reducing G towards R. For example, if `AllowUsers` is missing (equivalent to `*`) and two subjects `user1` and `user2` are the only ones who successfully logged in, the tool will recommend narrowing down `*` to the set $\{user1, user2\}$.

5.1.3 Greedy Algorithm for Discovering a Maximal Patch The Config. Generator includes a greedy algorithm to compute a maximal patch to a configuration automatically. This is achieved by iteratively restricting each configuration and testing if the new set of granted permissions is a subset of the original one without rejecting any required permission. The algorithm is shown in Figure 8 in Appendix B.

5.2 DB Schema and Query Mapper

The DB Schema and Query Mapper loads a user query, then extracts tables and their attributes from the database, and dynamically generates an SQL query before submitting it to the database. A significant advantage of the Query Mapper is that the user does not have to manually craft SQL queries for basic uses. For our experiments, we found that the Query Mapper suffices for investigating permission gaps; however an advanced user may choose to query the database directly if it is required.

```
Object.path = ?
Object.type = (LIKE,"%private key%")
ReqPerm = ?
Right.label = (=,"other-read")
```

Fig. 3. Query for files with types matching SQL pattern "%private key%" and read by others.

A user can make three kinds of queries. The first kind of queries are SPJ queries and are made by the Query Mapper on the database directly. The second and third kinds of queries are gap analysis and configuration change queries respectively.

SPJ queries allow users to formulate their own queries, thus leveraging their domain knowledge to search for permissions granted to access certain objects. The user simply specifies constraints using pairs having the format `(<operator>, <constraint value>)`, where `<operator>` is one of SQL's comparison operators. For example, if a user knows that files having types containing keywords "private key" may be possible attack targets, she may use the query template shown in Figure 3 to query for all `other-read` accesses on types matching "%private key%" for further analysis.

Gap analysis queries pertain to asking questions regarding permission gaps and how they may be reduced. There are two kinds of gap analysis queries that we have found useful. Firstly, given a specific change to a configuration setting, a user can ask what are the permission gap changes with respect to R. This is helpful for a user who is deciding if a change should be made. The Query Mapper may inform the user that the permission gap is (i) *under-tightened*, i.e., there are subjects who are granted permissions and did not previously access the object, (ii) *over-tightened*, i.e., there are subjects who were previously granted access but were denied given the change, or (iii) *no permission gap*, i.e., all subjects who were previously granted permissions and accessed the object still have the permissions, and all subjects who did not access the object are now denied.

The second gap analysis query we found useful is that the user can request for a list of possible one-step changes to the configuration settings that lead to reduction in the permission gap without over-tightening. There may be different

ways to reduce permission gaps. For example, for SSHD, to disallow root login, the user can either set PermitRootLogin to "no", exclude root from AllowUsers, or both. The user may use the result to selectively modify the configuration file. One can think of this type of one-step analysis to have the same semantics as a breadth-first-search for a set of tightened configurations.

The last kind of queries, the configuration change queries, return a set of configuration settings that can reduce the permission gaps without over-tightening. This kind of queries provide a list of sequence of changes to the configuration settings that help reduce permission gaps without over-tightening. This uses results from the Greedy Algorithm discussed in Section 5.1.

6 Evaluation

We implemented DeGap in Python and used SQLite for the database for detecting permission gaps in SSHD, auditd, and user groups. The three scenarios have vastly different ways for specifying permissions. File permissions are specified in the file system's inode structure and retrieved using system calls. On the other hand, SSHD configurations and user groups are stored in configuration files. Also, the types of configuration values differ, i.e., binary versus ranges. Additionally, for SSHD, certain configuration settings, such as PermitRootLogin and AllowUsers, interact to determine if a certain permission should be granted.

6.1 Case Study: Tightening SSHD Configurations

We considered SSHD as a single object and subjects to be users attempting to connect with the service. An access right could be either password or public-key authentication. The permissions were set in /etc/ssh/sshd_config. For simplicity, we will only discuss AllowUsers, PermitRootLogin, PubkeyAuthentication, and PasswordAuthentication, whose uses were inferable from SSHD logs. We evaluated the SSHD logs from two servers, a source code repository and a webserver hosting class projects, in our department. Since accesses to SSHD are logged by default, we did not have to make any changes to the system.

```
PermitRootLogin = {type = oneof,
            values = yes | without-password | forced-commands-only | no,
            default = yes}
PubkeyAuthentication = {type = oneof, values = yes | no, default = yes}
PasswordAuthentication = {type = oneof, values = yes | no, default = yes}
AllowUsers = {type = setof(subject), default = *}
```

Fig. 4. Configuration generation rules used as input to DeGap.

For both servers, we used the configuration generation rules in Figure 4 for hinting to the Config. Generator how the values for each field should be tightened. PubkeyAuthentication and PasswordAuthentication could be either

Table 1. Partial `SSHD` configurations for Server 1 and 2, with their original and tightened values as suggested by DeGap.

Configuration	Server 1		Server 2	
	Original	Suggestion	Original	Suggestion
PermitRootLogin	yes	without-password	yes	no
AllowUsers	user1 user2 root user3	user1 root	<unspecified>	user1 user2
PubkeyAuthentication	yes	yes	yes	yes
PasswordAuthentication	yes	no	yes	yes

`yes` or `no`. For `PermitRootLogin`, if `without-password` was used, root would not be able to login using a password. If `forced-commands-only` was used, only public-key authentication would be allowed for root. If `AllowUsers` was specified, only specified users would be granted access; otherwise, all users had access.

The second column of Table 1 shows part of the original configurations used by `SSHD` on the first server. Both `publickey` and `password` authentication methods were allowed. As specified by `AllowUsers`, four (anonymized) users, user1, user2, user3, and root were allowed `SSHD` access. We used DeGap to automatically compute possible tightest configurations, as shown in the third column of Table 1. We manually verified the results using traditional tools such as `grep` to search for keywords such as "Accepted" in the log files. Only user1 and root did access the server, suggesting that permission creep had occurred for two users. A check with the server owner confirmed that the three suggested configuration changes should have been applied.

The fourth column of Table 1 shows part of the `SSHD` configurations used by Server 2. Except for `AllowUsers`, all configurations had a value of `yes`. The `AllowUsers` field was unspecified, implying that anyone could connect to Server 2 using either password or public-key authentication over `SSH`. DeGap suggested new configurations as shown in the last column of Table 1. DeGap recommended the `AllowUsers` field to be specified with user1 and user2, and the `PasswordAuthentication` and `PubkeyAuthentication` fields to retain their original values of "yes". Again, we manually verified that user1 and user2 did access Server 2 using the public-key and password authentication methods respectively. The owner concurred with the suggestions.

In summary, DeGap correctly discovered the changes to configuration settings for each server to reduce their respective permission gaps. For Server 1, an unused authentication method was proposed to be disabled. For Server 2, a much stricter setting for `AllowUsers` having only two users were suggested.

6.2 Case Study: Tightening File Permissions using `auditd`

In this section, we describe using DeGap for finding file permission gaps with `auditd` logs collected over seven days on 17 departmental servers that were installed with Fedora 15 and were patched regularly. We added a rule for recording `execve` and `open` syscalls on files, "`-a exit,always -F arch=b64 -S execve -S open`", to `auditd`'s rule file. A successful call to `open` indicates that the calling

process has acquired the `read` and `write` access rights specified by the open flags (`O_RDONLY`, `O_WRONLY`, `O_RDWR`). While one could log other syscalls such as `read` and `write` to ascertain the actual rights needed to access an object, the open flags already circumscribe these rights. Moreover, logging these syscalls leads to expensive disk operations. On average, we collected over 12 million file open and execute events for over 345,000 files by over 200 users for each server. The analysis was performed on two identical machines, both having two Dual-Core AMD Opteron 2218 Processors and 16GB RAM.

Leveraging Query Language Regex We performed a query that leverages the query language's regular expression matching capabilities. We searched for files with types matching the SQL pattern "%private key%" using the query shown in Figure 3. On all the machines, we found that the RSA private keys for Dovecot, a popular IMAP and POP3 server, on all analyzed servers were world-readable. We verified this by checking Dovecot's SSL configuration file, which is also world-readable. If an attacker obtains the key, she can generate a fake certificate and launch a man-in-the-middle attack. In fact, it is highlighted on Dovecot's SSL configuration page that no users, except root, require access to the key file [22]. The administrator agreed that the `other read` permission should be removed.

```
distinct = yes
count = ?
Right.label = "other-write"
GrPerm.has_gap = yes
```

Fig. 5. Query for finding number of files with `other-write` permission gaps.

File Permission Gaps In this part of the analysis, we take a macro view on the permission gaps. Using the query shown in Figure 5, we were able to identify a large number of files with permission gaps. Figure 6 shows the number of files and directories that have the original read, write, or execute permission set with the number of files whose corresponding permissions were actually used. For files, respectively, only 0.592%, 0.00432%, and 2.59% of the group read, write, and execute permissions were actually used. More importantly, only 0.0592% and 0.181% of the other read and execute permissions were used. No writes by other users were observed. As for directories, the percentages for group read (list), group write (modify), other read, and other write were 0.0128%, 0.0429%, 0.0354%, and 0.341% respectively.

World-writable files presented a huge security risk since an adversary could easily modify the files while violating the non-repudiation principle. While there were 14,348 world-writable files, fortunately they belonged to only 17 users. These files include research data, exams, reports, etc.

The risks of world-readable files are not necessarily lesser than those of world-writable files. In total, we observed 1,133,894 world-readable files. Based on filenames in the `auditd` logs, some of the user files potentially contained sensitive information such as visa applications, (hashed and plain-text) passwords, and emails. For one user, while the inbox was not world-readable, the outbox was.

We applied DeGap towards generating configuration settings for reducing the permission gaps for files. DeGap was able to propose the correct suggestions

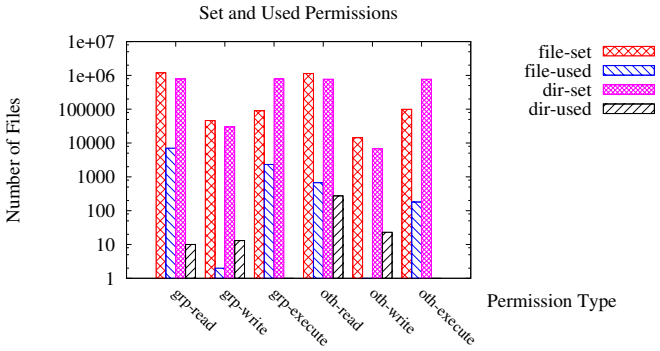

Fig. 6. Number of files and directories with permissions set and actually used.

for reducing the permission gaps. While it may be overly idealistic to remove every unused permission, the huge discrepancies in the number of files whose permissions were set and actually used illustrate the enormous potential for some of these redundant permissions to be removed. The large number of files may lead to manageability issues. The list can be fed into another system capable of extracting semantic information about the files to rank files according to potential criticality from access control perspective.

Generating New Groups An important aspect of tightening file permissions involves assigning files to appropriate groups such that accesses by others are minimized. We trivially extended DeGap to suggest such assignments by checking for directories whose files were accessed by others. For each candidate directory, the contained files would be considered as candidates for adding to a new group. The users who accessed these files could be added as members to the same group. Similar to SELinux, we considered files within the same directory to be in the same security context, i.e., DeGap does not propose creating new groups for files in the parent or children directories of a candidate directory. We only examined user directories, since in comparison to system directories such as `/bin` and `/etc`, they are more likely to contain sensitive data. Interestingly, we found that three out of the 17 servers contained user directories that had files being accessed (reads and writes) by others, and all of them were repositories (concurrent versions system, subversion, and mercurial). Indeed, online installation guides often recommend that (i) groups be created, (ii) the repository file groups be changed to those, and (iii) repository users be added to the groups.

6.3 Case Study: Tightening `/etc/group`

The `passwd` and `group` files in the `/etc` directory are used to manage users on Unix-based OSes. We now discuss how DeGap can be used to identify dormant groups, which can possibly be removed from `group`, during the monitoring period. DeGap can also be used to tighten `passwd`. However, this is less interesting and can be achieved easily using other means. Thus, we will not discuss DeGap's usage for the purpose of tightening `passwd` in the interest of space.

We considered a group to be dormant if it was not used to access files. We used the same logs in Section 6.2 as inputs [1]. However, we re-defined the object to be a collection of all the files (instead of each file being a single object). Also, there is only a single access (instead of read, write, and execute accesses).

The `group` files for all 17 servers were the same. This implied that a truly dormant group must be dormant across all servers. Thus, after finding the dormant groups for each server, we computed the intersection of these groups to determine a set of dormant groups for all servers. From this set, we removed groups having system users as members, since such groups are less likely to be susceptible to permission creeps. We identified system users as those whose home directories are not `/`, `etc`, `bin`, `sbin`, or `var`. 526 out of 565 groups were found to be dormant during monitoring. While it may be possible that there are false positives, as we acknowledged earlier to be a limitation of log-based approaches, it provides a starting point for administrators to tighten the `group` file.

7 Conclusion and Future Work

Permission gaps expose a system to unnecessary risks. This can be worsened by permission creep that is hard to eliminate in practice [23]. Without information about whether permissions are actually used, identifying and thus removing permission gaps is a challenge. Towards a framework using a common logic for extracting this information, we propose DeGap. We described the framework used, and demonstrated DeGap's capabilities in real-life analysis of `SSHD` permissions, file permissions and user groups from logs.

From `SSHD` logs, we found two users erroneously having access to a server, suggesting permission creep. Additionally, legitimate users only accessed the server using public key authentication. Despite this, the server had allowed password authentication while being subjected to password brute-force attacks. On another server, root was allowed access when it did not need it. DeGap proposed changes needed for the configuration settings to eliminate these gaps without over-tightening the permissions. The server owner concurred with the proposals.

From `auditd` logs, DeGap found that Dovecot's private key was world-readable on all the servers we tested, contradicting the recommendation on Dovecot's site [22]. DeGap also found a large number of user files with permission gaps for world-read or world-write. Some of these files appeared to contain sensitive information including passwords, suggesting that DeGap could be useful to both administrators and typical users for detecting permission gaps. DeGap also uncovered dormant user groups as candidates for removal from `/etc/group`.

Looking forward, DeGap currently automatically proposes the set of permissions that excludes operations not found in the logs. However, users may want more flexibility by specifying specific operations to exclude. Additional research is needed to achieve this flexibility.

[1] While other logs, e.g., `SSHD` logs, may allow us to estimate active users, they may not provide sufficient information for determining whether a certain group is dormant.

References

1. Andreas Beckmann. Debian 'openvswitch-pki' Package Multiple Insecure File Permissions Vulnerabilities. http://www.securityfocus.com/bid/54789, August 2012.
2. Andreas Beckmann. Debian 'logol' Package Insecure File Permissions Vulnerability. http://www.securityfocus.com/bid/54802, August 2012.
3. Andreas Beckmann. Debian 'extplorer' Package Insecure File Permissions Vulnerability. http://www.securityfocus.com/bid/54801/info, August 2012.
4. Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.
5. Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.
6. Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Automatically securing permission-based software by reducing the attack surface: An application to android. *CoRR*, abs/1206.5829, 2012.
7. Ryan Johnson, Zhaohui Wang, Corey Gagnon, and Angelos Stavrou. Analysis of android applications' permissions. In *SERE (Companion)*, pages 45–46. IEEE, 2012.
8. R.S. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40 –48, sept. 1994.
9. Pierangela Samarati and Sabrina de Vimercati. Access control: Policies, models, and mechanisms. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer Berlin / Heidelberg, 2001.
10. S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. NAI Labs Report #01-043, NAI Labs, Dec 2001. Revised May 2002.
11. Mick Bauer. Paranoid penguin: an introduction to novell apparmor. *Linux J.*, 2006(148):13–, August 2006.
12. Eugene H. Spafford Dan Farmer. The cops security checker system. In *USENIX Summer Conference*, pages 165–170, June 1990.
13. Tiger analytical research assistant. http://www-arc.com/tara/, 2002.
14. The Bastille Hardening program: increase security for your OS. http://bastille-linux.sourceforge.net/.
15. Files and file system security. http://tldp.org/HOWTO/Security-HOWTO/file-security.html, June 2012.
16. Jay Harrell James Cannady. A comparative analysis of current intrusion detection technologies. Technical report, Georgia Tech Research Institute, Atlanta, GA 30332-0800.
17. Open Source Tripwire. http://sourceforge.net/projects/tripwire/, 2012.
18. Butler W. Lampson. Protection. In *Princeton University*, pages 437–443, 1971.
19. Rainer Gerhards. Performance Optimizing Syslog Server. http://www.monitorware.com/common/en/articles/performance-optimizing-syslog-server.php, January 2004.
20. Django: The Web framework for perfectioniss with deadlines. http://www.djangoproject.com, 2012.
21. Amol Deshpande, Zachary Ives, and Vijayshankar Raman. *Adaptive Query Processing*, volume 1. 2007.
22. Timo Sirainen. Dovecot SSL configuration. http://wiki2.dovecot.org/SSL/DovecotConfiguration, April 2012.
23. Tom Olzak. Permissions Creep: The Bane of Tight Access Management. http://olzak.wordpress.com/2009/10/01/permissions-creep/, October 2009.
24. T. Vidas, N. Christin, and L. Cranor. Curbing Android permission creep. In *Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011)*, Oakland, CA, May 2011.
25. P.K. Manadhata and J.M. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371 –386, may-june 2011.

A Relationship between Permission Gaps, Permission Creep, and Attack Surfaces

Eliminating permission gaps inherently stops permission creep and reduces the attack surface of a system. It may therefore be useful to understand the relationship between these terms, and the implications of this relationship.

Permission creep is an insidious accumulation of permission gaps over time [23], for example, due to granting employees additional permissions beyond their normal roles for a temporary need, but failing to revoke permissions when employees leave or the need disappears. Resolving the permission gap problem implies that the permission creep problem will also be resolved. However, the converse is untrue since permission gaps can still exist in the absence of permission creep. For example, in the case of a single user system, permission creep is unlikely to occur while permission gaps may exist. The term "permission creep" is also used by Vidas et al. [24]. However, the semantics of that term would better match that of "permission gap" in our context as well as Felt et al. [4] and Bartel et al.'s work [6].

The attack surface of a system is the set of methods available to a potential attacker for gaining access to a system with the potential to inflict damage [25]. Clearly, having a smaller attack surface roughly corresponds to having a more secure system. Granted permissions contribute to a system's attack surface and permission creep increases the attack surface "area". Eliminating the attack surface entirely by removing all permissions is impractical, but in pursuing an adequate level of security, it should be the system owner's objective to reduce their system's attack surface as much as is practical. DeGap provides necessary information to help achieve this.

B Algorithms for Config. Evaluator and Discovering Maximal Patch

```
function ConfigEvaluator(C′, G, R)
    G′ = Permissions granted by configuration C′
    if G′ ⊆ G and R ⊆ G′ then
        SaveResults(C′, G′)
        if (G′ \ R) ≡ ∅ and (R \ G′) ≡ ∅ then
            return tightest                          ▷ Can't tighten further
        end if
        return tighter
    end if
    return bad
end function
```

Fig. 7. Algorithm for Config. Evaluator, E.

```
function GreedyConfigSpec(C, G, R)
    for all c_j ∈ C do
        ret = ok
        while ((c'_j = restrict_setting(c_j)) is valid)
    and (ret ≠ tightest) do
            C' = (C − {c_j}) ∪ {c'_j}                    ▷ Replace c_j with c'_j
            ret = ConfigEvaluator(C', G, R)
                                          ▷ Also saves good results to database
            if ret ≠ bad then
                C = C'
            end if
            c_j = c'_j
        end while
    end for
end function
```

Fig. 8. Greedy Algorithm for Discovering a Maximal Patch

Table 2. Comparison between number of entries in log file and number of tuples in database for `SSHD`.

Server	# of Log Entries	# of tuples in DB Table			
		Subjects	Rights		Object
			Success	Failure	
sshd-server1	579,788	74	32	73,395	1
sshd-server2	262,223	80	44	49,213	1

C Space Efficiency for `SSHD` Daemon Logs

Table 2 compares the number of lines in the `SSHD` logs and the corresponding number of tuples for the Subject, Object, and Right tables. Comparing the large number of log entries and relatively smaller number of tuples in the Rights table, there was a large number of duplicated accesses, many due to password brute-force attacks. For example, for sshd-server1, 73,395 accesses that failed using the password authentication method originated from 66 subjects. Except for one, the remaining 65 IP addresses do not belong to our university's network. The savings in space was reflected in the log sizes, where the ratio of the log sizes to the database sizes were 8.31 and 6.47 for `sshd-server1` and `sshd-server2` respectively.

D Space Efficiency for `auditd` Logs

We compared the average number of entries in the log files to the average number of tuples for Principal, Operation, and Resource tables as shown in Table 3, using a bounded-buffer for 327,680 hashes. The ratios of the log sizes to the database sizes are shown in Figure 9. On average, the logs were found to be approximately five times larger than the databases for our datasets. If logs from a modified `auditd` were used, the cost of parsing the logs could potentially be cut by approximately 80%, given the smaller size of the logs.

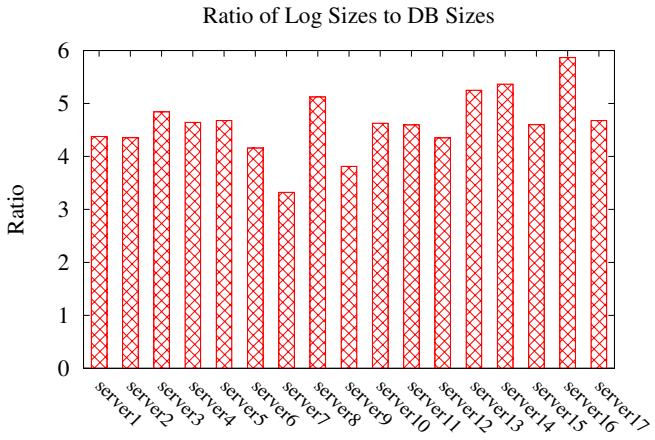

Fig. 9. Ratio of log sizes to database sizes for `auditd`.

Table 3. Comparison between average number of entries in log files and average number of tuples in databases for `auditd`.

# of Log Entries	# of tuples in DB Table			
	Principals	Operations		Resource
		Success	Failure	
12,607,326	215	5,454,845	136	345,669