MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications

Maliheh Monshizadeh
Department of Computer Science
University of Illinois at Chicago
Chicago, IL
mmonsh2@uic.edu

Prasad Naldurg
IBM Research India
Bangalore, India
pnaldurg@in.ibm.com

V. N. Venkatakrishnan
Department of Computer Science
University of Illinois at Chicago
Chicago, IL
venkat@uic.edu

ABSTRACT

We explore the problem of identifying unauthorized privilege escalation instances in a web application. These vulnerabilities are typically caused by missing or incorrect authorizations in the server side code of a web application. The problem of identifying these vulnerabilities is compounded by the lack of an access control policy specification in a typical web application, where the only supplied documentation is in fact its source code. This makes it challenging to infer missing checks that protect a web application's sensitive resources. To address this challenge, we develop a notion of *authorization context consistency*, which is satisfied when a web application consistently enforces its authorization checks across the code. We then present an approach based on program analysis to check for authorization state consistency in a web application. Our approach is implemented in a tool called MACE that uncovers vulnerabilities that could be exploited in the form of privilege escalation attacks. In particular, MACE is the first tool reported in the literature to identify a new class of web application vulnerabilities called Horizontal Privilege Escalation (HPE) vulnerabilities. MACE works on large codebases, and discovers serious, previously unknown, vulnerabilities in 5 out of 7 web applications tested. Without MACE, a comparable human-driven security audit would require weeks of effort in code inspection and testing.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.8 [**Access Control**]: Metrics—*Software Engineering,*

General Terms

Security Languages

Keywords

Access Control, Web Security, Authorization

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660337.

1. Introduction

Web applications are powerful engines that drive modern societies, as they play a pivotal role in e-commerce, social networking and finance. Due to their open nature and wide deployment, they make appealing targets to criminals who want to gain access to users' data and resources. Security of web applications therefore has become an important concern.

To protect from such threats, web applications implement *access control* (a.k.a. authorization) policies. A typical authorization check in a web application involves verifying whether a given authenticated user with an associated functional *role* has the required *privilege* to access a given resource such as a database table. Since authorization is expected to be performed before every resource access, it therefore forms the basis for security of the web application.

Several high-profile data breaches were caused due to the authorization errors in web applications. A most notable one is the Citibank data breach [1], wherein more than 360k credit card numbers were stolen. Such breaches suggest that web application authorization errors could be disastrous for organizations. Furthermore, such vulnerabilities appear to be widespread, as a recent Cenzic technical report [6] listed that authorization vulnerabilities occurred in 56% of the applications that were tested in the 2013 study.

There are several reasons why such authorization errors are numerous. First, unlike conventional operating systems, web applications (such as those written using PHP) do not come with built-in support for access control. The access control policy is often coded by a developer into the application. Developers often focus on other key functionalities of the applications, and often make errors in programming authorization code, as illustrated by the 2011 CWE / SANS report [3], in which missing authorization and improper authorization are ranked 6th and 15th in the top 25 most dangerous software errors. Second, a web application (such as one written using PHP and SQL) often connects directly to the database resource as a superuser who enjoys all administrative privileges on the database, and any flaws in the authorization logic often lead to catastrophic data breaches. Further, web application developers often implement roles [16] as a privilege management solution. However, the unavailability of a standard framework, and the lack of developer's knowledge of access control design, have led to buggy and inconsistent role implementations in applications [24].

The academic and industrial communities have identified several solutions to the problem. Virtual private databases [5] provide a way for applications to execute queries on behalf of users, and provide effective privilege separation. Web application frameworks such as Rails [23] provide software engineering solutions to structure the access control logic of an application effectively. Despite these advances, a vast majority of web applications continue to be

developed in languages such as ASP, Java and PHP where the onus of developing and enforcing the access control policy largely falls on developers.

In this landscape, we take a closer look at the problem of identifying whether an existing web application contains authorization errors. This vulnerability analysis problem is often a challenge for open source web applications that come with almost no documentation (except their code) of their access control policies. It is indeed possible for a vulnerability analyst to look for errors by understanding the access control policy, inspecting the source for missing authorizations. However, in order to arrive at the access control policy of the application by studying its source in detail, the manual effort involved is significant and could be time-consuming and tedious for large web applications. Therefore automated solutions that identify authorization errors are desirable, and we present one such strong approach in this paper.

The crux of our approach to vulnerability analysis is rather than try to uncover the unwritten access control policy of a given application, to instead turn our focus towards identifying whether the application enforces its policy *consistently* across various resource accesses. For example, consider a banking web application that enforces a complex policy based on principals such as managers, tellers, and customers. Instead of trying to uncover the application's intended authorizations, we examine if it consistently enforces the same authorization rules for accesses to a resource by the same principal. When such authorization checks are inconsistently checked along two different paths of an application, it is a strong indication that access control may be incorrect in one, leading to *privilege escalation* vulnerabilities and resulting in loss of sensitive data to outside attackers or malicious insiders.

Our technical approach to find authorization inconsistencies in applications involves defining a notion of *authorization context* for web applications that is associated with every program point in the application. We then develop a notion called *authorization context consistency*, which is satisfied when the application uses the same authorization context in order to access the same resource along different paths of a web application. When there is a mismatch in authorization contexts along two different paths, we flag that as an potential access control violation.

We develop algorithms for computing authorization contexts and checking for authorization context consistency. These algorithms involve a variety of program analysis techniques that include control flow analysis, data flow analysis and symbolic evaluation. These algorithms are implemented in a tool that we call MACE (Mining Access Control Errors). These algorithms are bootstrapped by a small set of annotations provided by the vulnerability analyst, and we show that the effort for providing these annotations is small.

Using our approach, we are able to detect two kinds of privilege escalation vulnerabilities: the conventional (1) *Vertical Privilege Escalation (VPE)* when an attacker (outsider) or a malicious user (insider) tries to change her privilege level (access more than they are entitled to, say according to their role) and (2) *Horizontal Privilege Escalation (HPE)* when a malicious user tries to access the system resources of other users. In particular, our modeling of authorization context and our detection algorithms facilitate the detection of the latter kind of privilege escalation, thus making MACE the first tool in the literature that is capable of identifying HPE vulnerabilities in web applications.

Evidence of the usability and usefulness of MACE is demonstrated by running it against a large number of open-source codebases. We test our tool against 7 applications and detect both horizontal and vertical privilege escalations in 5 of them. Without MACE, a security audit would have to manually inspect hundreds of thousands of lines of code.

The rest of the paper is organized as follows: Section 2 describes the problem with a running example. In Section 3, we explain the key ideas behind our approach, including the notion of the authorization state, as well as computing authorization contexts. We present implementation details and related challenges in Section 4. Section 5 presents results of our analysis on our testbed of web applications. Section 6 is related work, and we summarize our contributions in Section 7.

2. Running Example

In this section, we illustrate the key aspects of the authorization problem for web applications with the help of an extended example. The traditional authorization or privilege escalation problem is tied to the functional role of a user in this context. If this user can exercise privileges that are not usually associated with their functional role, a vertical privilege escalation vulnerability is detected. In addition, as described in CWE-639 [2] the horizontal authorization problem describes a situation where two users may have the same role or privilege level, and must be prevented from accessing each other's resources.

Listings 1 to 7 present the source of a running example that illustrates these authorization vulnerabilities. The example is a simplified version of real-world code samples analyzed by our tool, and describes typical vulnerabilities that were discovered. The particular web application here is a blog that permits its registered users to insert, edit, delete, or comment on blog articles. There are two functional roles: admin and user, with the admin having control over all posts in the blog, whereas the individual users should only be able to insert, edit, or delete their own blog, and comment on other blogs.

Listings 1, 2 and 3 refer to a secure implementation of the application. Function `verifyUser`, shown in Listing 1, checks if the request is coming from an authenticated user. In Listing 2, an article is being added to the `articles` table in the database. The user name of the current logged-in user specifies the owner of the article, and the request includes the article text that is inserted into the database. Note that this insert implementation is secure, as the user is verified, and is found to have the required permission. Listing 3 refers to the delete operation, where the user can delete any post that she owns. Additionally, an admin user, as specified by the role `userLevel`, can delete all entries in a blog as shown by the second `DELETE` operation.

Listings 4, 5, 6 and 7 show example PHP files that implement the delete operation. Each implementation of the delete operation is vulnerable as described below:

- *No authorization* In Listing 4, the application performs a delete without checking if the user is authorized.
- *Improper permissions*. In Listing 5, the application does not check if the user has the appropriate permissions to delete an article.
- *Improper Delete-all* In Listing 6, the application does not check if the user trying to delete-all belongs to the `Admin` role, and therefore permits a privilege escalation attack.
- *Improper Delete* In Listing 7, the application does not check whether the user requesting the delete is the owner of the article, and is authorized to delete it. It therefore allows the currently logged in user to delete articles owned by any other user in the system, as long as the (public) article-ID is supplied as query argument.

The last two examples in the above list deserve special mention. Listing 6 is the conventional form of privilege escalation allowing an ordinary user to assume admin privileges, i.e., vertical privilege escalation (VPE). In contrast, Listing 7 allows for an ordinary user to assume privileges of any other ordinary user in the system, a form of privilege escalation known as *horizontal privilege escalation* (HPE) [4]. To the best of our knowledge, this paper is the first to discuss an approach for detecting HPE vulnerabilities automatically, in addition to detecting VPEs.

```
1  function verifyUser(){
2        if(!isset($_SESSION['userID']))
3              header('Location: /login.php');
4        else $userID = $_SESSION['userID'];
5        return;
6  }
```

Listing 1: verifyUser.php

```
1  verifyUser();
2  if($permission['canWrite']&&$action == 'insert')
3        query("INSERT INTO tbl_articles VALUES (
4        sanit($_GET['article_code']),
5        $_SESSION['userID'],
6        sanit($_GET['article_msg']))");
```

Listing 2: insert.php

```
1  verifyUser();
2  if($permission['canWrite']&&$action=='delete')
3        query("DELETE FROM tbl_articles WHERE
4        article_ID = '" + sanit($_GET['article_ID']) + "'
               and
5        author_ID = '" + $userID + "'");
6  else if($_SESSION['userLevel'] == 'Admin' && $action ==
       'deleteAll')
7        query("DELETE FROM tbl_articles");
```

Listing 3: delete.php

```
2.a if($action == 'delete')
3.a       query("DELETE FROM tbl_articles WHERE article_ID =
               '" + sanit($_GET['article_ID']) + "'");
```

Listing 4: delete1.php (vulnerable version)

```
1.b verifyUser();
2.b if($action == 'delete')
3.b       query("DELETE FROM tbl_articles WHERE article_ID =
               '" + sanit($_GET['article_ID']) + "'");
```

Listing 5: delete2.php (vulnerable)

```
1.c verifyUser();
...
6.c if($permission['canWrite'] && $action == 'deleteAll')
7.c       query("DELETE FROM tbl_articles");
```

Listing 6: delete3.php (vulnerable)

```
1.d verifyUser();
2.d if($permission['canWrite'] && $action == 'delete')
3.d       query("DELETE FROM tbl_articles WHERE article_ID =
               '" + sanit($_GET['article_ID']) + "'");
```

Listing 7: delete4.php (vulnerable)

Lack of Policy Specification Note that our techniques are designed to work directly on the source code of our target applications, without relying on the existence of a well-articulated policy manifest to clarify these functional roles. In order to know whether the web application is implementing its access control correctly, one needs to know what access control policy is implemented. Unfortunately, the only documentation of this policy is in fact the source code of the web application. Furthermore, we also face the problem that this policy implementation can be incomplete or incorrect. This makes the problem of checking for access control errors quite challenging.

3. Approach

To reason about a web application's authorization correctness, one must examine each sensitive operation (e.g. each SQL query execution) of the program and examine the authorization information required to perform that operation. Recall from Section 2, the running example identifies what can go wrong in the implementation of access control, including the absence of any authorization checks, improper ownership or privileges corresponding to user role, and untrusted session variables.

Authorization state Applications should ideally have a well-defined policy manifest of what authorizations should be granted to what users, taking into account the session context, but unfortunately this is not always explicit. Even in applications that manage to have policy documents, the implementation may not match the specification. The best understanding of access policy therefore is the operating context of each access request in the implementation. For each access request in a user session, corresponding to a particular control and dataflow in the program execution, we argue that the four tuple $\langle U, R, S, P\rangle$ represents the associated access control rule explicitly, with U the set of authenticated users, R the set of roles defined over the users, capturing different authorizations, S the set of session identifiers or session variables, and P the permissions defined on the resources (e.g., `read`, `write`). This set $\langle U, R, S, P\rangle$ is our authorization state. We illustrate this in our running example:

- In Listing 4 the identity of user is not checked. The value of `$action` (i.e., delete) comes from the input form and therefore is controllable by the user, and cannot be trusted. We infer that the access rule checked here is $\langle -, -, -, -\rangle$, which means any user in any role can actually execute this `DELETE` query providing any article if they know the corresponding *article_id*, which is a placeholder for session context.
- In Listing 5, the following access rule is being checked $\langle user, non\_admin, -, -\rangle$. Access is allowed to any user, and it is not checked if they are an admin or not.
- In Listing 6 the appropriate role is not being checked, the incorrect (inferred) access rule here is $\langle user, -, -, canWrite\rangle$ whereas the actual rule needs to include a check that the user is admin.
- In Listing 7 the correct ownership information, corresponding to the *user* who created the *article_id* is missing in the access check. Instead, the rule inferred here is $\langle user, non\_admin, -, canWrite\rangle$.

From these examples, we now see that the correct access rule associated with the delete query on *tbl_articles*, depending on the role of the user, should be:

- $\langle user, non\_admin, -, canwrite\rangle$ and
- $\langle user, admin, -, -\rangle$

However, determining that this is the access rule, and that this is correct is not at all obvious. As mentioned earlier, all we have is the implementation, where the access rule is both control and data sensitive. Depending on whether the user is admin or not, different rules apply, indicating dependence on control. The ability to delete an article also depends on whether the same user had created the article, or had permissions to create it, requiring knowledge of data variables using data-flow analysis. Also implicit is the notion of the underlying access model. In this example, though the `admin`

Table 1: The Authorization Context for different queries in Running Example

Query	Authorization Context
Listing 2	$_SESSION['userID'],$permission['canWrite'],Column<$_SESSION['userID']>
Listing 3 Line 3	$_SESSION['userID'],$permission['canWrite'],Column<author_ID>==$_SESSION['userID']
Listing 4	∅
Listing 5	$_SESSION['userID']
Listing 7	$_SESSION['userID'],$permission['canWrite']
Listing 3 Line 8	$_SESSION['userID'],$userLevel == 'Admin'
Listing 6	$_SESSION['userID'],$permission['canWrite']

user may not be the owner of the article, an implicit role hierarchy lets her delete items and use the permission `canWrite`. Unfortunately, if the implementation is incorrect, the task of finding authorization errors becomes even more difficult.

Authorization Context One of the main ideas in our approach is the notion of matching what we call the authorization context, across related or complementary security sensitive operations, in terms *four-tuple* we have identified. We have no prior assumption about the authorization policy used by the web application authors. This authorization context is garnered by examining the code and trying to fill out the four-tuple at a given program point automatically. To do this we will first need to annotate the code to identify some of these fields manually. We populate our analysis by tagging the variables corresponding to user-ids, roles, session identifiers (i.e., those sets of variables that change every session) and permissions.

With each security sensitive operation identified, our goal is to try to infer what access rule is being enforced by the code. Using the annotated variables and the clauses in the query, we can now compute the *actual* authorization context at a given program point using a combination of control flow and data flow analysis. More details of these techniques are presented in Section 4. In Table 1 we show the actual context from a correct program from Listings 2 and 3 for `INSERT` and `DELETE`, and the actual context inferred from each of the incorrect inserts and deletes from Listings 4 through 8. Independent of what is correct, we observe that there are missing gaps in the conditions checked for access, across `INSERT`s and `DELETE`s to the same rows in the same table. Once we construct the authorization context, this acts as a specification for the access policy as implemented by the developer. The obvious question now is whether this policy is correct. However, we do not have any information about whether this is the case or not. It is possible for us to take this actual context to the developers and ask them to establish its validity, but this may not be always possible.

Authorization Context Consistency We observe that we can also compare the authorization context for different, but matching queries on the same tables. When the application uses the same authorization context in order to access the same resource along different paths of a web application, we term its authorization contexts to be *consistent* across the application. Any inconsistencies identified in this manner could indicate a potential problem with the access control implementation. Note that we do not know the correct access policy here, what we are trying to do is detect inconsistencies across related operations. We illustrate this idea with an example:

`INSERT` queries in a database are a good example of code segments that contain rich authorization information. For example, during creation of a row in a database table, we can expect to find some information about the *owner* of the row. Consider table `articles` with columns `(article_id, article_author, article_text)`. The following query adds an entry to this table:

```
INSERT INTO articles (article_author, article_text)
    VALUES ($_SESSION['userID'],
    sanitize($_GET['post']));
```

The variable `article_id` is incremented automatically. The ownership information, as to who can insert into this table can be inferred from `article_author` and the value for this column comes from `$_SESSION['userID']`. This ownership information is being checked on access, the authorization column tuple being $\langle$`$_SESSION['userID']`$,-,-,-\rangle$.

Let us now examine the corresponding `DELETE` query on the same table `article`. Here, the only parameter for delete comes from the user input (via `GET`).

```
DELETE FROM article WHERE article_id = $_GET['post_ID'];
```

From the listing, it is clear that the authorization context for the delete does not check ownership, i.e., the inferred context is $\langle -,-,-,-\rangle$. If any user can guess the range of the current IDs in the table, she can delete any row owned by any other user. This simple example now suggests that it is useful to make the constraints or authorization states for these queries *consistent* and add the userID to the authorization context of `DELETE` as:

```
DELETE FROM article WHERE article_id = $_GET['post_ID']
    AND article_author = $_SESSION['userID'];
```

The notion of computing the actual context and comparing it with those obtained from matching rules as discussed, using the authorization state four-tuple is both powerful and general. It accommodates a variety of different application access control models, being agnostic to the actual models directly. No a priori definitions or models are required, and the violations detected can encompass scenarios such as dynamic authorization and separation of duty (SoD), and the DAC model as shown. In fact, the DAC model is implied with the ownership information in the `INSERT` query. As long as the attributes that determine access can be captured by the authorization state abstraction, rich context variables such as time of day, location, integrity constraints, keys and shared secrets, etc., can all fit easily with the techniques discussed.

Note that normal sanitization of the user input, without associating it with the current session token is not sufficient. Of course, there are many challenges associated with this kind of matching. The obvious one is that this could be intended behavior, i.e., `DELETES` may have different permissions from Inserts. The actual context on the `INSERT` could be incorrect. Further, there could be more than one insert, corresponding to different roles or different session characteristics and the corresponding delete has to be matched up accurately. Nevertheless, as we show in this paper, searching for inconsistencies in matching operations helps us resolve errors in real applications. The question is how often they lead to false positives or negatives and we explore this in detail in Section 5.

Algorithm Overview We now describe our algorithm to compute the authorization context, and compare contexts across matching requests to discover inconsistencies. To start the analysis, we need to find all the queries in the code, and proceed to compute the context at these query locations. Next, we compare the contexts of similar access locations (i.e. query locations). Inconsistency in the contexts or in the way the authorization tokens are used in accessing the DB, e.g., using `where` clauses can lead to detecting vulnerabilities as described.

Algorithm 1: Algorithm Overview

```
input : Application source, Authorization variables, Possible values for role
        variables
1  cfg := ControlFlowAnalysis();
2  dda := createDependencyGraphs(cfg);
3  sinkPaths := enumeratePaths(dda);
4  foreach sp ∈ sinkPaths do
5  |   AuthzContextAnalysis(sp);
6  |   if sink ∈ {INSERT, UPDATE, DELETE} then
7  |   |   queries += <symbolicQuery(sink), authz-c(sink)>;
8  analyzeInserts();
9  analyzeDeletes();
10 analyzeUpdates();
```

As shown in Algorithm 1, the main input to our analysis is the application source code (PHP code), annotated variables that correspond to user ids, roles, session specific attributes and permissions appropriately. Once we have these annotations we perform a control flow analysis to identify paths involving these authorization sensitive variables. Next, we construct a data dependency graph using the annotated control flow graph to capture dataflows between the identified variables. A source-sink graph corresponding to entry points (sources) in web applications to particular sensitive queries (sinks) is now ready. On this graph, we gather the constraints at each sink as well as the annotated information flow context and construct our query context (lines 4-8). Context for different queries is computed in Algorithms 2 and 3 and checked for consistency to find errors. Further details are presented in Section 4.

4. Implementation

Figure 1 shows the architecture of MACE, identifying the various components of our tool, with (numbered) outputs produced by each component that are subsequently used as (numbered) inputs to other components.

Inputs. There are two sets of inputs to MACE: (1) source files and (2) annotations or hints provided by the developer / end user. Specifically, the set of hints provided by the developer is a small set of super global variables that constitute the various components of the authorization 4-tuple as described earlier. In PHP, typically these annotations are on super-global variables such as `SESSION`. In our running example, the hint provided to the tool is that the super-global `userID` constitutes the specification of the user component of our 4-tuple. In our experience, the effort required to specify these hints is not high, as it took only a few minutes for each application that we tested (as discussed in our evaluation). With this information, together with the source code, MACE is able to identify potential privilege escalation errors.

Control Flow Analysis To identify authorization errors in the program, MACE uses static analysis methods to analyze the code. The advantage of using static analysis is that it can identify *all* sensitive accesses to important resources of a given type (e.g. SQL queries), and analyze all execution paths that lead to them. MACE includes a front-end to parse source files (in PHP). Subsequently a control-flow analysis is performed that results in a CFG (numbered 3 in figure) for the application, which explicitly identifies control flows throughout the whole application. In addition, this component also identifies a set of sensitive *sinks* in the application. Currently, sink identification in MACE is performed for SQL query locations (as identified by calls to `mysql_query`).

Data Dependency Analysis The next step in MACE is to compute a data dependency analysis. To illustrate the need for data dependency analysis, let us consult the running example. In Listing 1, the variable `$userID` holds the user information, which it receives from the super-global `$_SESSION['userID']`. `$userID` is subsequently used, and we need to capture these types of dataflows to reason about authorization. This requires a data dependency analysis, which is done by constructing data dependence graphs (DDGs) for each procedure.

In addition, MACE's analysis is inter-procedural. To see the need for inter-procedural analysis, let us consult the running example again. The assignment of `$userID` happens in procedure `verifyUser` (Listing 3), whereas the use of `$userID` happens in another file (`delete.php`). Since this analysis requires us to see such data-flows across all procedures, MACE also builds a system dependence graph (SDG) [18], which is essentially an inter-procedural DDG. The output of this step is a SDG (numbered output 5) in Figure 1.

Slicing In order to look for authorization errors at a particular sink, MACE analyzes paths that lead from the sources (entry points in a web application) to that sink. Such analysis needs to be *path sensitive*. Consider the running example in Listing 1. In this function, there are two paths, one that successfully checks if the user ID has been set (through a prior authentication step, not shown in the example for brevity), and the other that exits the application. The authorization context therefore exists in only one of the paths, and our analysis must be able to select such paths for further analysis, as well as ignore the other path, as it would not lead to a sink. Therefore we require a path sensitive analysis.

In order to analyze each path, we perform inter-procedural slicing using the SDG [18]. Intuitively, for a given sink such as a SQL query, the corresponding SDG captures all program statements that construct these queries (data dependencies) and control flows among these statements. MACE performs backward slicing for the sinks such that each slice represents a *unique* control path to the sink. Each of these control paths is therefore an instance of sensitive resource-access. A number of steps are performed during the slicing operation. Loops are expanded by unrolling, either 0 or 1 or 2 times. In addition, the conditional expressions are preserved in the SDG by nop nodes, so that the information that is checked by these conditional expressions can be used in computing the authorization context. For instance, in our running example, the condition `isset($_SESSION['userID'])` is stored along each control path. Paths that do not reach a sensitive sink are omitted from subsequent analysis. At the end of this slicing step, MACE outputs a list of source-sink paths (numbered 6 in Figure 1. For our running example involving Listings 1, 2 and 3, we have three, one that reaches the `INSERT` query, another that reaches the `DELETE` query, and a third that reaches the same query but corresponds to deleteAll.

Authorization Context Analysis Using the paths computed during the slicing step, MACE computes the authorization state along each such path from the source to the sink. To do this, it starts with the super-globals identified from the user provided annotations (numbered by 2 in Figure 1) and checks if they (or other

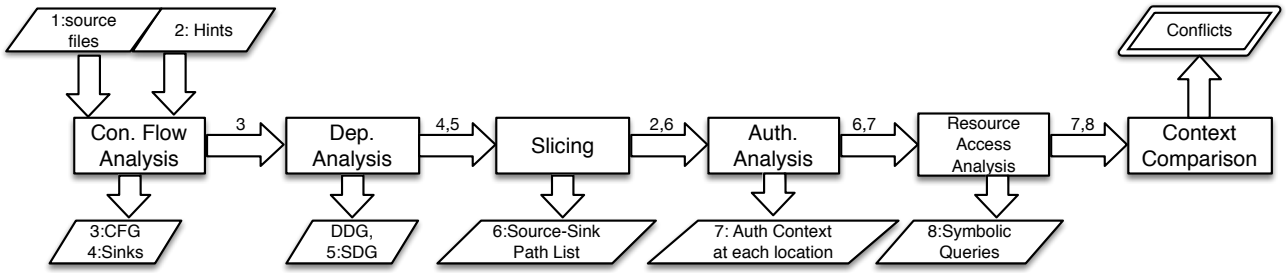

Figure 1: System Architecture. The numbers shown refer to outputs produced during various components, which are used as inputs for subsequent components.

program variables that get receive values from these super-globals through data-flows) are consulted in conditions along the path from the source to the sink. If so, that information is symbolically represented in the authorization context 4-tuple. For instance, for our running example involving Listings 1 and 3, deleting (Line 3) involves an authorization context that checks both the user has logged on, the permission (`canWrite`) and therefore the corresponding context is inferred: $\langle$`userID`$, \_, \_, \{canWrite\}\rangle$ whereas Listing 1 and 5 involve the following authorization context: $\langle$`userID`$, \_, \_, \_\rangle$. This step is computed using our sliced SDGs, which provide the data-flow information. At the end of this step, MACE outputs source-sink paths with annotated authorization context at each location along the path.

Resource Access Analysis Having computed the authorization context at all program points in any given path, the next step in MACE is to see how these are used towards accessing application resources. The main resources in web application are DB tables, and we need to check whether the authorization contexts across all resources are consistent. However, the access control context at the query location may still be incomplete in capturing the access restrictions the application places along the current path being analyzed. To see this, we refer to Listing 3. In the first `DELETE` operation, the query is constrained by two `WHERE` clauses: (1) the specific article to be deleted and (2) the `author_ID` field from postAuthor of the table restricted to the current `userID`. The latter constrains the authorization context by way of *row restriction*: i.e., each article can be deleted only by a user who has her `userID` stored in the same row of the table. This happens only when the corresponding `INSERT` query for the same row (shown in Listing 2) inserts the `userID` in the field `author_ID`. This implicit form of "ownership" needs to be captured by MACE in the authorization context. Thus, the relationship between `author_ID` and `userID` that is implicit in the code needs to become explicit. The tool captures this information as *Authorization Columns* for each table. Each *authorization column* element captures the column name and the symbolic value used to constrain the query. Returning to our running example, the *authorization columns* for the table `tbl_articles` is: `[author_ID = $_SESSION['userID']]`

Note that, in cases where the developer did not specify column names in `INSERT` queries, MACE uses DB schemas, which are automatically generated by parsing database creation files (`CREATE TABLE` queries) to mark the *authorization columns*.

For this purpose, MACE symbolically evaluates the source-to-sink path that leads to the query so that the relationship between the super-global variables that eventually reach the query becomes explicit. Symbolically executing the path that leads to the first

Algorithm 2: Analysis of INSERT Queries

input : List of queries and their authorization contexts
output: List of Tables, Tables' authorization contexts, Tables' authorization columns

```
1  sortSymbolicQueries();
   // based on Table names
2  foreach table t do
3      foreach ins-q ∈ getInsertQueries(t) do
4          authzContexts(t)+=getAuthzContext(ins-q);
           authzCols+=getAuthzColumns(ins-q);
5      if AuthzContext(t).size > 1 then
6          diff := compare(authzContexts(t));
7          if diff then
8              raiseWarning(INSERT_Conflict);
9      if authzCols(t).size > 1 then
10         diff := compare(authzCols(t));
11         if diff then
12             raiseWarning(INSERT_AuthzColumn_Conflict);
13 return authzCols, authzContexts;
```

`DELETE` query in Listing 3 makes this relationship explicit. For instance, we obtain the following symbolic query for the above example.

```
DELETE FROM tbl_articles WHERE article_id =
    $_GET['article_ID'] AND author_ID =
    $_SESSION['userID'];
```

Notice that the query is now entirely expressed in terms of symbolic super-globals such as user inputs `$_GET['post_id']` and `$_SESSION['userID']`. Having the symbolic query for each query location allows us to compare similar accesses to the column `authorID` in the DB, and compare the authorization contexts for *similar* accesses to the same table.

In addition to identifying inconsistencies, the previous steps allow one to see if there is a possibility of privilege escalation due to insecure use of user-supplied parameters in authorization decisions. User supplied parameters such as `$_COOKIE` can be tampered, and therefore authorization decisions must not refer to them. Having a symbolic query that makes any such use explicit also facilitates MACE to identify these types of errors. At the end of this step, MACE outputs a set of symbolic queries. These queries contain the resources (tables) and together with the authorization context at each resource along every path that leads to the resource. The specific type of access (`INSERT, UPDATE, DELETE`) is also available through the symbolic query. As mention earlier, paths that lead to `SELECT` queries or those that do not lead to sensitive resources are discarded.

Context Comparison The goal of this step is to compare the authorization context at each resource access and identify inconsistencies. To this point, we have gathered two sets of information with respect to authorization in previous steps: 1) the authorization

Algorithm 3: Analysis of DELETE Queries

```
   input : authzCols, authzContexts
1  foreach table t do
2      foreach del-q ∈ getDeleteQueries(t) do
           // Compare Authorization Contexts
3          if getAuthzContext(del-q) == getAuthzContext(t) then
               // Compare Authorization Cols
4              authzClause = getAuthz(getWhereClauseCols(del-q));
5              if authzClause != getAuthzColumns(t) then
6                  raiseWarning(HORIZONTAL_ESC);
7          else if getRole(getAuthzContext(q)) < getRole(getAuthzContext(t))
               then
8              raiseWarning(VERTICAL_ESC);
9          else
               // Other inconsistencies may have various
                  authorization vulnerabilities
10             raiseWarning(INCONSISTENCY);
```

contexts for query locations and 2) the resource access parameters (authorization columns) for each table.

The first step in context comparison is to group the query-path pairs based on the *table names* in the query (Line 1 in Algorithm 2. During symbolic execution analysis done in the previous phases, we are able to resolve the table names for static and dynamic queries in each possible sink-path pair. Then, for each table, we gather the authorization context and authorization columns, which are used in table accesses. The number of distinct authorization contexts and authorization columns may be more than one, for different queries in different program locations. Therefore, we need to resolve these differences and in any case report these conflicts. Lines 7-11 in Algorithm 2 compare the contexts for `INSERT` queries on a given table. The discussion about the analysis of these conflicts comes in Section 4.1. After finishing all `INSERT` queries, we proceed to analyze `DELETE` and `UPDATE` queries. The reason we start this way is because *ownership information* is added to the resources at their creation time, in a DAC model, which is typical of such applications. `INSERT` queries show us where the ownership information comes from in the program and in which column of the table they are going to be stored.

In the next step, to analyze the rest of the queries (i.e. `DELETE` and `UPDATE` queries) we compare their authorization contexts shown by 1) the resource accesses (where clauses in queries) and 2) the authorization context annotations, with the information we gathered from `INSERT` queries. Algorithm 3 shows the analysis of delete queries. The analysis of `UPDATE` queries is done in the same way.

The data in database tables should be changed exclusively with a privilege level equal or more than the level specified in their authorization context so that the integrity of these tables remains intact. Lines 9 and 10 in Algorithm 3 show how we check for vertical privilege escalation vulnerabilities in our tool.

In addition to the authorization context, we check table access parameters (lines 4-6 in Algorithm 3) to detect horizontal privilege escalation vulnerabilities. These attacks happen in the same privilege level as the legitimate users, however, a malicious insider can manipulate the data stored in DB tables owned by other users. This additional check of accesses tries to prevent such attacks.

Lines 12-15 in Algorithm 3 detect any other inconsistency in the authorization contexts. These inconsistencies are also reported back to the user for further analysis.

4.1 Conflicting Contexts

During the comparison phase, both for authorization contexts and the resource access comparisons, there may be scenarios in which the contexts or access parameters do not match entirely. In these cases, the authorization context or `WHERE` clause with *more restrictions* is viewed as the *stronger* context / clause. This is intuitive – quite often, the *more restrictive* the context, the more specific (or precise) it is about the access rules regarding the resource being referred to. Below, we discuss different conflict scenarios and how MACE addresses these conflicts.

In the case of one `INSERT` query present in the application for a given resource, we assume that the authorization information extracted from the query must be present in the authorization context of further accesses (`UPDATE`s or `DELETE`s). In case of any conflict after the comparison, if the authorization context of the `INSERT` query is *stronger* than the other query's context, we raise a warning. Depending on the missing element in the authorization 4-tuple, the type of the warning may vary. A missing or weaker *role* information is generally an indication of a vertical privilege escalation vulnerability caused by the current privilege role level being less than what was present in the `INSERT` query's context. A missing *user* element in the context, while it was present at the time of the insert, indicates a horizontal privilege escalation vulnerability. Relying on user provided inputs (such as `GET`, `POST` or `COOKIE` variables) in the 4-tuple is an indication of a general mismanagement of sessions and authorization in the application.

4.2 Precision and minimizing warnings

MACE is a 'best effort' tool to detect missing or inconsistent authorization information. It is based on the intuition that developers aim to get most cases right, and some occasional cases wrong. In the rare case that the developer gets *none* of the cases right, then MACE's approach cannot detect errors.

To have a more effective tool with a better confidence rate, there are a few general steps we took in order to improve the overall precision and lower the false positive rate.

`INSERT` queries with missing authorizations So far, we set authorization contexts in `INSERT` queries as the base for consistency analysis. If an `INSERT` query misses some crucial authorization information, then our tool may not report faulty `DELETE` or `UPDATE` queries (as long as they are consistent with the faulty `INSERT`), causing possible false negatives.

User-controllable query parameters. There are some applications that permit user-controlled parameter (such as those of `GET, POST, COOKIE`) values in authorization decisions. Since MACE uses data flow analysis, it is able to observe these incorrect authorizations and reports them (we identify and report 10 such errors in our evaluation). In such cases of vulnerabilities, MACE does not further proceed to analyze the queries that are impacted by these flaws, so that the number of warnings reported by the tool is minimized. After fixing these vulnerabilities, the user can re-run MACE to identify if there are still missing authorizations.

`SELECT` queries. Technically speaking, it is possible for MACE to include `SELECT` queries and analyze them for inconsistencies, as the analysis required to identify authorization for a `SELECT` query (e.g. dataflow analysis) is no different compared to an `INSERT`. However, including `SELECT` queries is primarily a question of the user's tolerance of the signal / noise ratio for an policy-agnostic tool such as MACE. To see this, let us consider an example of a news article website. The news articles are publicly viewable and so at the corresponding `SELECT` query there would be no authorization information, whereas the users of the website often have to authenticated and authorized to be able to post news articles. Comparing such `SELECT` queries with `INSERT` queries often will lead to false alarms. Therefore, in order to eliminate such false alarms, a user might decide to omit analyzing `SELECT` queries, as we did in the evaluation of MACE. Another choice that

Table 2: PHP Applications

Application	SLOC	# php files	# query Locs	# DB tables	Analysis time (s)
phpns 2.1.1alpha	4224	30	40	13	8220
DCPPortal 5.1.44	89074	362	308	34	982
DNScript	1322	60	27	7	35093
myBloggie 2.1.3	6261	59	24	5	373
miniBloggie 1.1	1283	11	5	2	35
SCARF 1.0	978	19	13	7	54
WeBid 1.0.6	27803	266	687	47	1492

Table 3: Overview of Vulnerabilities

Application	# query conflicts		VPE	HPE	known, unknown
	TP	FP			
phpns	7	0	✓	✓	0, 7
DCPPortal	46	0	✓	✓	0, 46
DNScript	0	0	-	-	-
myBloggie	6	0	✓	✓	3,3
miniBloggie	1	0	✓	-	1, 0
SCARF	11	0	✓	✓	1, 10
WeBid	0	0	-	-	-

a user has, which involves additional manual effort, is to provide additional annotations that identify the tables that store sensitive data (and therefore require authorization on `SELECT` queries).

4.3 Other Issues

Unsupported `PHP` features MACE is implemented for PHP, and makes use of the Pixy [19] tool for control flow analysis. A small set of features in PHP language are not handled by Pixy and therefore MACE does not deal with them. For instance, dynamic inclusions and certain object-oriented features of PHP are not handled entirely. However, these have not limited the applicability of MACE to the application suite that used in our evaluation. Note that, while MACE works in the context of PHP, which is widely used, our technique (using authorization context differentials, symbolic execution, dataflow analysis) is independent of any platform.

Counting the number of vulnerabilities. Many vulnerable queries might be fixed by a single common authorization check at a shared program location. However, MACE treats each query location as an independent operation and reports and counts vulnerable queries separately. We prefer to do so because we think each of the reported vulnerable queries might lead to a different instance of attack. By treating the queries in isolation, we can identify the vulnerability type with more precision. As we see in section 5, we may report large number of vulnerable locations because of one single missing authorization check, but in each such case of multiple vulnerabilities due to a single reason, we explicitly indicate so.

5. Evaluation

Implementation MACE is designed to analyze PHP Web applications. MACE is implemented in Java and is about 10K lines of code. We use an open-source tool and library (`TAPS` [9]) to get the control flow graphs and enumerate execution paths for PHP applications. The experiments described in this section were performed on a MacBook Pro (2.4 GHz Intel, 4.0 GB RAM).

5.1 Effectiveness

Experiments. We ran our tool to analyze the effectiveness of MACE on a suite of seven small to large `PHP` free and open-source Web applications. As shown in Table 2, the applications range from approximately 1k to 89k source lines of code (SLOC). These applications were used as benchmarks in previous research studies [28, 8, 24]. The results of our evaluation fall under the following categories: (1) vulnerabilities identified by MACE and detailed statistics about the vulnerabilities identified in our experiments, (2) performance, scalability of MACE and (3) the annotation effort required from the developers. We have verified by hand all of the authorization vulnerabilities in the applications, which were reported by the tool.

For each application, we annotate the authorization tuple variables and then run MACE with given annotations and the source code. MACE then lists all of the conflicts, the vulnerabilities, their locations in the source code and the values, which cause the inconsistencies.

Results summary. Table 3 presents the summary of our experiments. The table lists the number of conflict reports, and also shows how many of these were indeed vulnerabilities (true positives - TP) and how many of them were reported incorrectly (false positives -FP) by our tool. Furthermore, the breakdown of the vulnerabilities (true positives) between the two types of privilege escalations namely HPEs and VPEs is presented in columns 4 and 5. The last column in this table gives the breakdown of the identified vulnerabilities were known (i.e. previously reported in CVEs or by previous studies) versus unknown (i.e. zero-day vulnerabilities).

As reported in the last column of the table, MACE is able identify zero-day authorization vulnerabilities in the following applications: `phpns`, `DCPportal`, `mybloggie` and `SCARF`. In the following subsection, we will go through the details of these vulnerabilities.

5.2 Vulnerabilities Identified

phpns The `phpns` application is an open-source news system. The application allows three roles in the system for the users: 1) guest users (unauthenticated users) who can only view the news articles; 2) normal users who must be logged into the system and use the article management panels and 3) *admin* user who also must be logged into the system and can access both article and user management panels. Basic permissions such as adding, deleting and updating articles are set by default for the new users of the system.

```
$new_res = general_query('INSERT INTO articles
                (article_title, artcle_sbtitle,
                article_author, article_cat,
                article_text,...)
            VALUES('.$data['article_title'].','
            .$data['article_subtitle'].','
            .$_SESSION['username'].','
            .$data['article_cat'].','
            .$data['article_text']'');
```

Listing 8: Inserting an article item in in inc/function.php, `phpns`

```
$items = $_POST; //get vars
...
$sql = general_query("DELETE FROM
     ".$databaseinfo['prefix']."".'articles'." WHERE id
     IN (".$items.")");
```

Listing 9: Deleting an article item in `article.php`, `phpns`

Using MACE, we found seven vulnerabilities in this application, all of which are previously unknown, two of which we describe below. Consider the actual code for the inserting and deleting users, shown in Listings 8 and 9 respectively. The first vulnerability allows an unauthenticated user can delete any comment without any authorization checks. MACE was able to identify this because the relevant authorization context is not consulted at the delete operation. The implication of this vulnerability is that an outside attacker (who has no credentials in a given installation of `phpns`) can delete *any* comment item in the application. This is an example of a verti-

Table 4: Details of Warnings

Application	Number of violations		
	insert-insert	insert-update	insert-delete
phpns	0	5	2
DCPPortal	0	21	25
DNScript	0	0	0
myBloggie	0	3	3
miniBloggie	0	0	1
SCARF	1	8	3
WeBid	0	0	0

cal privilege escalation attack (VPE). Another detected vulnerability is found in `manage.php` that allows for an *authenticated* user to delete other users' news articles by providing arbitrary article IDs (which are available to all users through inspection of URLs). This vulnerability is a horizontal privilege escalation (HPE). We have reported these and other vulnerabilities in `phpns`.

dcp-portal The `dcp-portal` application is an open-source content management system. This application allows two authenticated roles: admin user and non-admin user (normal user). Consider Listings 10, 11, and 12, which refer to the authorization operation, insertion and deletion of agenda items in a calendar table. Variable `$_COOKIE["dcp5_member_admin"]` is being used to determine whether the user is an admin user or not. While inserting an item in the agenda, this variable is consulted, and the agenda item is entered in the table `t_agenda`. However, while deleting the item, while the authorization function is consulted, the deletion is based on a (user supplied) value `$_REQUEST["agid"]`, thus making the requests inconsistent. The implication of this vulnerability is that it allows any user in the system to delete another user's agenda entries, thus making it a HPE, which was a previously unknown vulnerability.

```
1 if (UserValid($_COOKIE["dcp5_member_id"])) {
2       ...}
```

Listing 10: Authorization function in lib.php, `dcp-portal`

```
1 if ((isset($_REQUEST["action"])) && ($_REQUEST["action"]
      == "add") && ($_REQUEST["mode"] == "write")) {
2       $sql = "INSERT INTO $t_agenda (user_id,
3       subject, message, date) VALUES
              ($_COOKIE['dcp5_member_id'] ," .
4       htmlspclchars($_REQUEST['subject']).",".
5       htmlspclchars($_REQUEST['aktivite']).",
6       $date)";
7       $result = mysql_query($sql);}
```

Listing 11: Inserting an agenda in calendar.php, `dcp-portal`

```
1 if ((isset($_REQUEST["action"])) &&
      ($_REQUEST["action"]=="delete")) {
2       $sql = "DELETE FROM $t_agenda WHERE id =
              '".$_REQUEST["agid"]."'";
3       $result = mysql_query($sql);}
```

Listing 12: Deleting an agenda in calendar.php, `dcp-portal`

We also found 44 other VPEs due to the incorrect implementation of `UserStillStillAdmin` function in `dcp-portal`. The first argument of this function takes the value of `$_COOKIE["dcp5_member_id"]` and determines whether the user with this userID is an admin. The value for the userID comes from a cookie variable and not from an established authorization state at the server side, which makes all 44 distinct queries in the admin path vulnerable to VPE.

myBloggie The `MyBloggie` application is an open-source blogging software. When we ran MACE on this application, we found six privilege escalation vulnerabilities. In three of these vulnerabilities, the validity of a session is not checked in many instances as the check shown in Listing 13 does not appear in del.php, delcat.php, deluser.php files. Even in the files that do check this constraint, MACE found horizontal escalation attacks. The parameters used to delete rows do not check for authorization information. For instance, the parameter used to access and delete the rows in `POST_TBL` is coming from user-supplied values such as `GET["post_id"]` and is prune to HPE. MACE found three such unreported vulnerabilities in this application.

```
1 if (!isset($_SESSION['username']) &&
       !isset($_SESSION['passwd'])) //go to login;
```

Listing 13: authorization Check in addcat.php, `MyBloggie`

miniBloggie The `miniBloggie` application is also a blogging Web application. In this application, there is no role or privilege level defined for the users. Thus, users are either authenticated (`$_SESSION['user']` is set) or not. Even with this simple authorization rule, the application is vulnerable to privilege escalation as detected by MACE. These scenarios involved missing checks that need to be present in order to ensure the user is a valid one before access to table rows is granted. In `del.php` (Listing 15), function `verifyuser()` is omitted, making way for the vulnerability, which was previously unknown.

```
1 session_start();
2 if (!verifyuser()){
3       header( "Location: ./login.php" );
4 }else {...
5       if (isset($_POST["submit"])) {
6             $sql = "INSERT INTO blogdata SET
7             user_id='$id',
8             subject='$subject',
9             message='$message'"...";
```

Listing 14: Inserting a blog user in add.php, `miniBloggie`

```
1 session_start();
2 if (isset($_GET['post_id'])) $post_id = $_GET['post_id'];
3 if (isset($_GET['confirm'])) $confirm = $_GET['confirm'];
4 if ($confirm=="yes") {
5       dbConnect();
6       $sql = "DELETE FROM blogdata WHERE
              post_id=$post_id";
```

Listing 15: Deleting a blog user in del.php, `miniBloggie`

SCARF The `SCARF` application is an open-source conference management software which helps the user to submit and review papers. The possible roles in `SCARF` are admin and normal user. Both roles must be authenticated to interact with the software. Variable `$_SESSION['privilege']` indicates whether a user is an admin or not.

MACE detects several types of authentication and authorization bypass vulnerabilities in `SCARF`. For example, in `generaloptions.php`, the admin can delete users and modify the `option` table. The page has no authorization check before it proceeds to performing `admin` tasks. As a result of this vulnerability, a normal user of the system who is legitimately authenticated can delete other users. To fix the problem this method, `require_admin()`, should be added at the beginning of the file which verifies whether the current session is the admin session or not. If it is not the admin session, the program exits.

```
1 if (isset($_GET['delete_email'])) {
2       query("DELETE FROM users WHERE email='" .
              escape_str($_GET['delete_email']) . "'");
3 }
```

Listing 16: Deleting a user in generaloptions.php, `SCARF`

Table 5: Analysis of Queries

Application	# query-path pairs			# query-authzInfo pairs
	insert	update	delete	
phpns	2564	222	920	78
DCPPortal	56	60	58	158
DNScript	8	2	13	26
myBloggie	5	0	2	40
miniBloggie	3	9	1	3
SCARF	4	26	12	19
WeBid	131	22	7	323

```
function require_admin() {
      if (!is_admin()) {
            die ("...");}
}
function is_admin() {
      if ($_SESSION['privilege'] == 'admin') return TRUE;
      else return FALSE;
}
```

Listing 17: Missing Authorization in generaloptions.php, SCARF

Ten other vulnerabilities reported by MACE in this application can be attributed to a single reason. The reason for these vulnerabilities being reported is that the constraining parameter used in certain UPDATE or DELETE queries derives it value from $_GET['session_id'], which is an untrusted source (i.e., the HTTP client). The corresponding INSERT query uses the $_SESSION['user_id'] which is an authorization variable as shown in the following code snippets. The column session_id in table sessions is an auto-increment key. Since untrusted values are never part of server authorization state, the authorization contexts for these queries were reported empty. Since the parameter $_GET['session_id'] is provided by the user, and the values are guessable (auto-incremented value), an attacker can impose himself on any guessable session.

```
INSERT INTO sessions (name, user_id, starttime,
     duration) VALUES
     (mysql_real_escape_string($_POST['name']),
     $_SESSION['user_id'], $date, $duration)
```

Listing 18: Inserting a session addsession.php, SCARF

```
UPDATE sessions SET user_id=$_POST['chair'] WHERE
     session_id=$_GET['session_id']
```

Listing 19: Updating a session in editsession.php, SCARF

Webid and DNScript MACE did not report any conflicts in these two applications.

Vulnerability & Inconsistency Reports. Table 4 shows the breakdown of the number of inconsistencies reported by our tool. The inconsistencies between various types of query pairs (insert-insert, insert-update and insert-delete). Together with table 3, we see that MACE is precise and produces no false positives. This low FP rate is due to the use of authorization 4-tuple to model the authorization state of sessions at the server. Using the reports generated by the tool (including the locations of the queries and the missing authorization), a developer can proceed to fix the application.

5.3 Performance & Scalability

We evaluated MACE on a suite of Web applications with different sizes ranging from 1K to 90K. Columns 2-3 in Table 2 show the size and number of php files in the applications, and column 4 gives an estimation of the number of query (insert-update-delete) locations (in source).

Table 2 (column 6) shows the total analysis time for each Web application ranging from 35 seconds to 35093 seconds. About %95 of the analysis time has been spent to create the dependency graphs and enumerate execution paths.

The increase in the number of possible paths increases the number of created symbolic queries. However, the number of distinct symbolic queries may still remain relatively low as shown in the last column of table 5, where we present the number of unique symbolic queries and their authorization information 4-tuple. Currently, MACE analyzes each file separately and builds the aggregated contexts when all the queries are gathered. The performance of MACE can be improved, especially if we summarize recurring contexts for basic user-defined functions. Since MACE is a static tool, the analysis times are quite acceptable for the benefits provided by the tool.

5.4 Annotation Effort

To run MACE, we manually identify the 4-tuple variables for each of the applications as hints for our tool. Developers typically use global and super-global variables (e.g. in SESSION or COOKIE) to represent user roles, user IDs, and the possible permissions for the logged-in users. These variables are further used to hold the authentication and authorization-related values throughout the program. Table 6 in Appendix A shows the variables we identified as hints for our programs.

The manual annotations are developed by observing the session management functions in login procedures. In our experience, developing these annotations is not hard for users familiar with the application, and certainly for developers who coded the application. To objectively measure the annotation effort, we performed a user-study experiment. To assist this experiment, MACE was extended to automatically generate a list of global and superglobal variables, which are used in if-statements, which is a superset of authorization variables. This list is then refined to exclude user input variables (such as GET and POST superglobals), and is provided as a starting point to the user.

To measure the effort needed to identify the 4-tuple, we asked a graduate student who had basic knowledge about Web applications to develop these annotations. We provided the application sources and the globals list generated by MACE. The student was provided the mybloggie and phpns applications, which are mid-tier applications in our benchmark suite. She was able to produce annotations that matched our own annotations for both the applications, and took about 50 minutes for generating and verifying the annotations. This experiment lends evidence that only modest efforts are required in providing annotations. We also note that our experience with providing such annotations is consistent with prior work in web access control that makes use of similar annotations [11, 29]. Given the number of unknown vulnerabilities identified by MACE, we believe such annotation-assisted automated bug finding is an attractive alternative to weeks of human effort and manual code inspection.

6. Previous work

We summarize related work in three broad categories: research focused on prevention of access control vulnerabilities, research aimed at detecting access control bugs in legacy web applications, and general program analysis techniques to find vulnerabilities in software. We contrast how MACE differs from many of these approaches, while uses some techniques in common.

Prevention of Authorization Vulnerabilities Nemesis [11] uses Dynamic Information Flow Tracking (DIFT) to establish a shadow authentication system that tracks user authentication state. Access control lists can be specified by programmers, which help the sys-

tem enforce authorization properties at run-time. CLAMP [21] uses virtual web servers to prevent authorization vulnerabilities in web applications: by migrating the user authentication module of a web application into a separate, trusted virtual machine (VM). All database access requests (queries) are mediated by a trusted VM that enforces defined access control rules and restricts the queries if necessary. Diesel [15] provides a proxy-based framework to limit database accesses at run-time. It uses the principle of least-privilege to secure the database through developer-defined policies. Capsules [20] develops a language based technique which uses Object-Capability languages to isolate objects from each other, in order to separate web applications into components. While these works are focused on *dynamic prevention* of access control errors, MACE is focused on *static detection* of access control vulnerabilities.

Swaddler [10] is an dynamic anomaly detection tool which is able to detect several types of bugs including workflow bugs. Swaddler has a Daikon-based invariant learning phase followed by an analysis phase which checks the invariants against the application state model. Although the Swaddler tool uses the notion of session and checks for the presence of session variables in execution paths, it does not take into account the access control model of the application with respect to various resources.

Finding Authorization Bugs in Legacy Web Applications Ganapathy et al. [17], where they add checks to enforce authorization rules in legacy software systems, such as X SERVER. They use a reference monitor for enforcing defined authorization policies at run-time.

RoleCast [24] uses common software engineering patterns to model authorization requirements and develops techniques to check if any sensitive operation is performed after authorization. While the advantage of using patterns is that it frees the need for developer annotations, we have noticed that the RoleCast patterns do not hold consistently across all web applications. The approach proposed by Sun et al. [26] detects vertical escalation vulnerabilities using static analysis. This approach builds a sitemap of the Web application, modeling the accesses to *privileged* webpages per role. It then checks if forced browsing cause the privileged pages to be accessed. Both approaches ([24] and [26]) use course-grained modeling of authorization requirements through grouping the roles. They only accommodate detection of vertical escalation vulnerabilities. In contrast, MACE employs a precise and fine-grained authorization model that is supported by user annotations of modest effort, giving it the ability to detect a larger class of vulnerabilities, including horizontal privilege escalation.

Doupe et al. [12] present an analysis of Execution after Redirect (EAR) vulnerabilities in web applications. They discuss a static control flow analysis for web applications that detect EAR attacks. While MACE is not built to detect EARs, the analysis infrastructure of MACE could be extended in a straightforward way to detect EAR vulnerabilities. In addition, the context inference for sinks in MACE could form the basis for automatically distinguishing benign EARs from vulnerable EARs.

Vulnerability Analysis in Applications Waler [14] uses a combination of static and dynamic analysis techniques to extract program specifications in terms of *likely invariants* and then uses model checking to verify the extracted invariants. MACE is similar in its objectives to Waler as both approaches aim to work with source code as the only specification. However, being focused on logic errors, Waler is not built to precisely compute authorization contexts. This limits its ability to identify access control discrepancies that require global reasoning across the entire web application, especially related to how a particular resource is accessed in various operations.

Engler et al. [13] also try to extract program specifications, through behavioral patterns called *beliefs*. They use static analysis techniques to infer these patterns and rank them using statistical analysis of the patterns. The patterns specified can be used to detect certain types of vulnerabilities caused by inconsistency in the programs, such as pointer dereference and use of locks on resources. Srivastava et al. [25] detect security vulnerabilities through comparing different implementations of the same API. They use security policies as input to their analysis. Any inconsistency between the security policy and any of the implementations or any inconsistency between different implementations are being reported as errors. AutoISES [27] can detect bugs in standard C libraries through mining for common security-related patterns and identifying deviations from these as vulnerabilities.

Blackbox approaches (NoTamper [7] and the approach proposed by Pellegrino et al. [22]) have some potential to reason about access control vulnerabilities in an application, but they are inherently limited in their ability to reason about authorization errors that manifest as a result of missing checks along specific paths present in source code which can only effectively gleaned through access to the application source code.

7. Conclusion

We present MACE, a program analysis tool for automatic detection of authorization vulnerabilities in Web applications. The tool is based on our study and characterization of different authorization attacks and the underlying vulnerabilities. We find privilege escalation vulnerabilities by finding inconsistencies in the authorization contexts at access request points without knowing the correct access control policies. While the analysis is best effort, the greatest value of MACE is in identifying flaws in these applications using fundamental abstractions, in the absence of any policy specifications, with the benefit of finding important vulnerabilities that were not discovered earlier.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments. We would also like to thank Kalpana Gondi, Abeer Alhuzali and Ivan Brugere who helped us with the annotation experiments.

This material is based upon work supported in part by the National Science Foundation under Grant Nos. 0845894, 1069311 and 1065537. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the U.S. government.

8. References

[1] Citi breach: 360k card accounts affected. `http://www.bankinfosecurity.com/citi-breach-360k-card-accounts-affected-a-3760`.

[2] Cwe-639. http://cwe.mitre.org/data/definitions/639.html.

[3] Mitre top 25. http://cwe.mitre.org/top25/.

[4] OWASP: Testing for privilege escalation. `https://www.owasp.org/index.php/Testing\_for\_Privilege\_escalation\_(OWASP-AZ-003)`.

[5] Virtual private database. `http://www.oracle.com/technetwork/database/security/index-088277.html`.

[6] Application vulnerability report. Tech. rep., `http://www.cenzic.com/downloads/Cenzic_Vulnerability_Report_2014.pdf`, 2014.

[7] BISHT, P., HINRICHS, T., SKRUPSKY, N., BOBROWICZ, R., AND VENKATAKRISHNAN, V. N. Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the*

17th ACM Conference on Computer and Communications Security (New York, NY, USA, 2010), CCS '10, ACM, pp. 607–618.

[8] BISHT, P., HINRICHS, T., SKRUPSKY, N., AND VENKATAKRISHNAN, V. N. Waptec: Whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 575–586.

[9] BISHT, P., SISTLA, A. P., AND VENKATAKRISHNAN, V. N. Taps: Automatically preparing safe sql queries. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 645–647.

[10] COVA, M., BALZAROTTI, D., FELMETSGER, V., AND VIGNA, G. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2007), RAID'07, Springer-Verlag, pp. 63–86.

[11] DALTON, M., KOZYRAKIS, C., AND ZELDOVICH, N. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *USENIX Security Symposium* (2009), USENIX Association, pp. 267–282.

[12] DOUPÉ, A., BOE, B., KRUEGEL, C., AND VIGNA, G. Fear the ear: Discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 251–262.

[13] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 57–72.

[14] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 10–10.

[15] FELT, A. P., FINIFTER, M., WEINBERGER, J., AND WAGNER, D. Diesel: Applying privilege separation to database access. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 416–422.

[16] FERRAIOLO, D. F., SANDHU, R., GAVRILA, S., KUHN, D. R., AND CHANDRAMOULI, R. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur. 4*, 3 (Aug. 2001), 224–274.

[17] GANAPATHY, V., KING, D., JAEGER, T., AND JHA, S. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 458–467.

[18] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1988), PLDI '88, ACM, pp. 35–46.

[19] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IN 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY* (2006), pp. 258–263.

[20] KRISHNAMURTHY, A., METTLER, A., AND WAGNER, D. Fine-grained privilege separation for web applications. In *Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 551–560.

[21] PARNO, B., MCCUNE, J. M., WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. Clamp: Practical prevention of large-scale data leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2009), SP '09, IEEE Computer Society, pp. 154–169.

[22] PELLEGRINO, G., AND BALZAROTTI, D. Toward black-box detection of logic flaws in web applications. In *NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA* (San Diego, UNITED STATES, 02 2014).

[23] Ruby on rails website. `http://rubyonrails.org/`, 2011.

[24] SON, S., MCKINLEY, K. S., AND SHMATIKOV, V. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011), OOPSLA '11, ACM, pp. 1069–1084.

[25] SRIVASTAVA, V., BOND, M. D., MCKINLEY, K. S., AND SHMATIKOV, V. A Security Policy Oracle: Detecting Security Holes using Multiple API Implementations. In *PLDI'11: Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, CA, USA, 2011).

[26] SUN, F., XU, L., AND SU, Z. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 11–11.

[27] TAN, L., ZHANG, X., MA, X., XIONG, W., AND ZHOU, Y. Autoises: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), SS'08, USENIX Association, pp. 379–394.

[28] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.

[29] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 291–304.

APPENDIX

A. Appendix

Table 6 shows the set of input information (hints) we provided for our tool MACE. Gathering this information about each application requires minimal effort and some familiarity with the applications, as discussed in Section 5.4.

Table 6: Provided Annotations to MACE

Application	Input Variables	Role Values
phpns	`$globalvars['rank']`, `$_COOKIE['cookie_auth']`, `$_SESSION['auth']`, `$_SESSION['username']`, `$_SESSION['userID']`, `$_SESSION['permissions']`, `$_SESSION['path']`	(dynamic)
DCPPortal	`$_COOKIE["dcp5_member_id"]`, `$_COOKIE["dcp5_member_admin"]`, `$HTTP_COOKIE_VARS-->["dcp5_member_admin"]`	(dynamic)
DNScript	`$_SESSION['admin']`, `$_SESSION['member']`	1for admin, 0 for non-admin
myBloggie	`$_SESSION['username']`, `$userid['level']`, `$_SESSION['user_id']`	1(for admin), 2 (for normal)
miniBloggie	`$_SESSION['user']`	-
SCARF	`$_SESSION['privlege']`, `$_SESSION['user_id']`	'admin', 'user'
WeBid	`$_SESSION['WEBID_LOGGED_IN']`, `$user_data['groups']`, `$_SESSION['WEBID_ADMIN_USER']`, `$_SESSION['WEBID_ADMIN_IN']`, `$group['can_sell']`, `$group['can_sell']`, `$group['auto_join']`	admin role flag, user groups have dynamic values