Multi-Module Vulnerability Analysis of Web-based Applications

Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, and Giovanni Vigna
Computer Security Group
University of California, Santa Barbara
Santa Barbara, CA, USA
{balzarot, marco, rusvika, vigna}@cs.ucsb.edu

ABSTRACT

In recent years, web applications have become tremendously popular, and nowadays they are routinely used in security-critical environments, such as medical, financial, and military systems. As the use of web applications for critical services has increased, the number and sophistication of attacks against these applications have grown as well. Current approaches to securing web applications focus either on detecting and blocking web-based attacks using application-level firewalls, or on using vulnerability analysis techniques to identify security problems before deployment.

The vulnerability analysis of web applications is made difficult by a number of factors, such as the use of scripting languages, the structuring of the application logic into separate pages and code modules, and the interaction with back-end databases. So far, approaches to web application vulnerability analysis have focused on single application modules to identify insecure uses of information provided as input to the application. Unfortunately, these approaches are limited in scope, and, therefore, they cannot detect multi-step attacks that exploit the interaction among multiple modules of an application.

We have developed a novel vulnerability analysis approach that characterizes both the *extended state* and the *intended workflow* of a web application. By doing this, our analysis approach is able to take into account inter-module relationships as well as the interaction of an application's modules with back-end databases. As a result, our vulnerability analysis technique is able to identify sophisticated multi-step attacks against the application's workflow that were not addressed by previous approaches. We implemented our technique in a prototype tool, called MiMoSA, and tested it on several applications, identifying both known and new vulnerabilities.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification

General Terms: Security

Keywords: Web Applications, Multi-step Attacks, Vulnerability Analysis, Static Analysis, Dynamic Analysis

1. INTRODUCTION

Web applications are growing in popularity. The introduction of sophisticated mechanisms for the handling of asynchronous events in web browsers and the availability of a number of frameworks for the rapid prototyping of server-side components have fostered the development of new applications and the transition of "traditional" applications (e.g., mail readers) to web-based platforms.

While new technologies have brought in significant advantages in terms of support to the development process, improved performance, and increased interoperability, little has been done to tackle security issues. Therefore, as the complexity of web applications increases, the possibility for abuse increases as well. For example, a simple analysis of the CVE vulnerability database [4] shows that the percentage of web-based attacks rose from 25% of the total number of entries in 2000 to 61% in 2006.

This situation is made worse by the fact that web applications are usually reachable through firewalls by design, and, in addition, the server-side logic is often developed under time-to-market pressure by developers with insufficient security skills. As a result, vulnerable web applications are deployed and made available to the whole Internet, creating easily-exploitable entry points for the compromise of entire networks.

To address the security problems associated with web applications, the research community has proposed a number of solutions. A first class of solutions focuses on detecting (and possibly blocking) web-based attacks. This can be done by analyzing the requests sent to web applications [13, 2, 21, 17, 18] or, in some cases, by analyzing the data delivered by the applications to the clients [11, 8]. These solutions have the advantage that they do not require any modification to the application being protected. However, they have a significant impact on the system's performance, and, in case of false positives (i.e., wrong detections), they may block legitimate traffic.

A second class of solutions focuses on identifying flaws in the implementation of a web application *before* the application is deployed. These approaches utilize static and dynamic analysis techniques to identify vulnerabilities in web applications [7,9,14]. Most of these approaches are based on the assumption that vulnerabilities in web applications are the result of insecure data flow. Therefore, these techniques attempt to identify when data originating from outside the application (e.g., from user input) is used in security-critical operations without being first checked and sanitized.

Even though these approaches are effective at detecting suspicious uses of unsanitized data, they suffer from three main limitations. First, their scope is limited to a single web application module, such as a single PHP file or a single ASP component. Therefore, these techniques are not able to identify vulnerabilities that are caused by the interaction of multiple modules. Second, these approaches are not able to correctly model the interactions among multiple technologies, such as the use of multiple languages in the same application, or the use of back-end databases to store persistent data. Third, and most important, these techniques do not take into account either the *intended workflow* of a web application or its *extended state*.

The intended workflow of a web application represents a model of the assumptions that the developer has made about how a user should navigate through the application. Web applications are often designed to guide the user through a specific sequence of steps. For example, an e-commerce site could be structured so that the user first logs in, then browses a catalog and chooses some goods, and eventually checks out and purchases the items. The constraints among operations (e.g., one has to select some goods before purchasing them) define the application's intended workflow.

A number of mechanisms have been devised to track the progress of a user through the intended workflow of a web application. These mechanisms provide ways to store information that survives a single client-server interaction and define the extended state of the application. For example, in a LAMP application[1] the extended state could include the request variables used in each module and, in addition, the PHP session data and the database tables, which are shared between modules. The extended state can also include information that is sent back and forth between the client and the server to keep track of a user session, such as hidden form fields and application-specific cookies. Therefore, the extended state of an application is a distributed collection of session-related information, which is accessed and modified by the modules of a web application at different times during a user session.

Unfortunately, it is possible that different modules of an application have different assumptions on how the extended state is stored and handled, leading to vulnerabilities in the application. We call these vulnerabilities *multi-module vulnerabilities* to emphasize the fact that they originate from the interaction of multiple application modules, which communicate by reading and modifying the application's extended state.

In this paper, we present a novel vulnerability analysis approach that combines several analysis techniques to identify sophisticated multi-module vulnerabilities in web applications. In our approach, we first leverage dynamic approaches to analyze block-level properties in the code of web application modules. We then use static analysis to extract properties at the module level. Finally, we use model checking techniques to identify possible paths in a web application's workflow that could lead to an insecure state.

The contributions of our approach are the following:

- We introduce a novel model of web application extended state that characterizes permanent storage and is not limited to the variables and data structures defined in a single procedure or code module.
- We present a novel approach to analyze the interaction between the application's code and back-end databases, which allows for the identification of sophisticated data-driven attacks.
- We introduce an approach to derive the intended workflow of a web application and an analysis technique to identify multi-step attacks that violate the expected inter-module workflow of a web application.

We implemented our approach in a prototype analysis tool, called MiMoSA[2], for PHP-based web applications, and we evaluated it on a number of real-world applications, finding both known and new vulnerabilities. The results show that our approach is able to identify complex vulnerabilities that state-of-the-art techniques are not able to identify.

The rest of the paper is structured as follows. In Section 2, we present some examples of the vulnerabilities that are the focus of our approach. In Section 3, we introduce the web application model that is at the basis of our analysis. Section 4 and 5 describe our approach to the identification of multi-module vulnerabilities in web applications. Then, Section 6 presents the results of applying our analysis to real-world applications. Finally, Section 7 presents related work, and Section 8 briefly concludes.

2. MULTI-MODULE ATTACKS

Multi-module attacks can be categorized into two classes: data-flow attacks and workflow attacks. Data-flow attacks exploit the insecure handling of user-provided information that is stored in the web application's state and passed from one module to another. In workflow attacks, an attacker leverages errors in how the state is handled by the application's modules in order to use the application in ways that violate its intended workflow.

Data-flow Attacks.

In multi-module data-flow attacks, the attacker uses a first module to inject some data into the web application's extended state. Then, a second module uses the attacker-provided data in an insecure way[3]. Examples of multi-module data-flow attacks include SQL injection [3] and persistent (or stored) Cross-Site Scripting attacks (XSS) [12].

A web application is vulnerable to a SQL injection attack when it uses unsanitized user data to compose queries that are later passed to a database for evaluation. The exploitation of a SQL injection vulnerability can lead to the execution of arbitrary queries with the privileges of the vulnerable application and, consequently, to the leakage of sensitive information and/or unauthorized modification of data. In a typical multi-module SQL injection scenario, the attacker uses a first module to store an attack string containing malicious SQL directives in a location that is part of the application's extended state (e.g., a session variable). Then, a second module reads the value of the same location from the extended state and uses it to build a query to the database. As a result, the malicious SQL directives are "injected" into the query.

In cross-site scripting attacks, an attacker forces a web browser to evaluate attacker-supplied code (typically JavaScript) in the context of a trusted web site. The goal of these attacks is to circumvent the *same-origin policy*, which prevents scripts or documents loaded from one site from getting or setting the properties of documents originating from different sites. In a multi-module XSS attack, a first module is leveraged to store the malicious code in a location that is part of the extended state of the application, e.g., in a field of a table in the back-end database. Then, at a later time, the malicious code is presented to a user by a different module. The user browser executes the code under the assumption that it originates from the vulnerable application rather than from the attacker, effectively circumventing the same-origin policy.

Workflow Attacks.

Most web applications have policies that restrict how they can

[1]A LAMP application is a web application based on the composition of Linux, Apache, MySQL, and PHP.

[2]MiMoSA stands for Multi-Module State Analyzer.

[3]As it will be clear later, this second module can be a second invocation of the module that performed the first step of the attack.

be navigated to ensure that their functionality and data is accessed in a well-defined and controlled way. Usually, to implement these restrictions a module stores in the web application's extended state the current navigation state, e.g., whether or not the current user has logged in or has already visited a certain page. Other modules, then, use this portion of the state information to deny or authorize access to other parts of the application.

Workflow attacks attempt to circumvent these navigation restrictions. For example, a workflow attack could try to directly access a page that is not reachable through normal navigation mechanisms, such as hyper-textual links[4]. These attacks may allow one to bypass authorization mechanisms (e.g., gaining access to restricted portions of a web application) or to subvert the correct business logic of the application (e.g., skipping a required step in the checkout sequence of operations on an e-commerce web site).

3. A FORMAL CHARACTERIZATION OF MULTI-MODULE VULNERABILITIES

In the previous sections, we described how the state of a web application can be maintained in a number of different ways. In order to abstract away from the various language- or technology-specific mechanisms, we introduce the concept of *state entity*. A state entity E is similar to a variable in a traditional programming language, in that it can be used to store parts of the application's state. Different modules can share information by accessing the same state entities. The set of all the state entities corresponds to what we defined in the introduction as the application's *extended state*.

We classify the state entities into two classes: server-side and client-side. Server-side entities model the part of the extended state that is maintained on the server. For example, a server-side entity can represent a field in a database or a PHP session variable. Client-side entities are instead used to model the part of the extended state stored in and/or generated by the user's browser. Cookies, GET and POST parameters are examples of this type of entities.

3.1 Module Views

To summarize the operations that each module performs on the application's extended state, we introduce the concept of *Module View* (or simply *view* hereinafter). Each view represents all the state-equivalent execution paths in a single module, i.e., all the paths in the control-flow graph (CFG) of a module that perform the same operations on the state entities. When an application module is executed, e.g., as a consequence of a user request, the path followed by the execution in that module is completely included in one and only one of its views. In this case, we say that the view that contains the executed path is "entered" by the user. We describe the algorithm used to summarize a module into its views in Section 4.3.

Consider, for example, the login module of an application. When a user provides correct credentials, the module may define a set of new session variables (e.g., to track that the user is authenticated and to load her preferences). On the contrary, the module may redirect unauthorized users to an error page without changing the extended state. These two different behaviors depend on the current extended state of the application, namely on the values of the request parameters and the content of the database that stores the information about the users. The view abstraction allows us to associate with each behavior a compact representation that summarizes its effect on the extended state of the application.

Formally, a view V is represented as a triple (Φ, Π, Σ) where:

[4]This attack is sometimes referred to as "forceful browsing."

- Φ is the view's pre-condition, which consists of a predicate on the values of the state entities. The program paths modeled by the view can be executed only when the view pre-condition is true (evaluated in the context of the current extended state).

- Π is the set of post-conditions of the view. These conditions model, as a sequence of write operations on state entities, the way in which the extended state is modified by the execution of the program paths represented by the view. Each write operation has the following form:

  $$write(E_L, E_R, \Psi).$$

  This operation copies the content of the left entity E_L (which can also be a constant value) to the right entity E_R. The set Ψ contains the sanitization operations applied to the left entity before its value is transferred to the right entity. If the sanitization set is empty, no sanitization is applied.

- Σ is the set of *sinks* contained inside the view. Each sink is a pair (E, Op) where E is a state entity and Op is a potentially dangerous operation (such as a SQL query or an `eval` statement) that uses the entity unsanitized. Note that the unsanitized use of an entity is not necessarily a vulnerability, since the sanitization process may take place inside one of the other views (belonging to the same module or to another module).

The extended state of an application may change as the user moves from one web page to another, clicking on links, submitting forms, following redirects, or just jumping to a new URL. In fact, when a view is entered, the extended state S is updated by applying the view's post-conditions to the extended state in which the application was before entering the view. Let $V_i = (\Phi_i, \Pi_i, \Sigma_i)$ be the view entered at step i of the user's navigation process, then:

$$S_{init} = \emptyset \qquad S_i = apply(\Pi_i, S_{i-1}).$$

In addition to the set of the entity values, the extended state also keeps track of the current sanitization state of each entity. An entity E is sanitized in the application state S_i (represented by the predicate $san(E, S_i)$) if its value is set by sanitizing write operations. In this work, we take the standard approach of assuming that sanitization operations are always effective in removing malicious content from user-provided data.

3.2 Application Paths

The presence of the pre-condition predicate in each view limits the possible paths that a user may follow inside the web application. We say that a path $P = \langle V_0, V_1, \ldots, V_n \rangle$, where V_i is a view, belongs to the set of *Navigation Paths* $\mathcal{N}$ if and only if:

$$\forall i < n, \;\; S_i \models \Phi_{i+1},$$

that is, if and only if the state at each intermediate step satisfies the pre-condition of the following step.

Since at the beginning of the execution the application state is empty, it must be $\emptyset \models \Phi_0$. In order for this to happen, the pre-condition Φ_0 must be empty or it must contain only predicates on client-side entities. This is justified by the fact that pre-conditions containing only client-side entities (for example, those requesting a particular value for a certain GET parameter) can always be satisfied if the user provides the right value. We define the set of

Application Entry Points η as the subset of views that can be used as starting points in a navigation path:

$$V_i \in \eta \qquad \text{iff} \qquad \emptyset \models \Phi_i.$$

The subset of navigation paths allowed by the application design is called the *Intended Path* set, $\mathcal{I} \subseteq \mathcal{N}$. These paths represent the workflow of the web application, expressed either through the use of explicit links provided by the application or through other common user navigation behaviors. We say that a navigation path $\langle V_0, \ldots, V_n \rangle$ belongs to the intended path set of the application if and only if:

$$\forall i < n \left(V_{i+1} \in \eta \vee \exists Link(V_i, V_{i+1}) \vee V_{i-1} = V_{i+1} \vee V_i = V_{i+1} \right).$$

In other words, at each step of the path the next view satisfies one of the following: it is an application entry point, is reachable through a link, is the same as the previous view (which corresponds to the user pressing the *back* button in her browser), or is the same as the current view (which corresponds to the use of the *refresh* button).

Given the previous definition, we can now provide a formal characterization of the two classes of vulnerabilities we introduce in this paper. A violation of the intended workflow of the application occurs when:

$$\exists p \in \mathcal{N} \mid p \notin \mathcal{I},$$

that is, when there exists a valid navigation path that is not an intended path.

A multi-module data-flow vulnerability is defined as:

$$\exists p = \langle V_0, \ldots, V_n \rangle \in \mathcal{N}\ ,\ \exists E_x \in \Sigma_n \mid \neg\, san(E_x, S_{n-1}),$$

that is, there is a path in the application such that some portion of the application's extended state is used in a security-critical operation without being properly sanitized.

4. INTRA-MODULE ANALYSIS

The analysis performed by MiMoSA consists of two phases: an intra-module phase, which examines each module of the application in isolation, followed by an inter-module phase, where the application is considered as a whole.

The goal of the intra-module analysis is to summarize each application module into a set of views, by determining its pre-condition, post-conditions, and sinks. From each module, we also extract the list of all outgoing links and we associate them with the views they belong to. This information is then used by the inter-module analysis to reconstruct the intended workflow of the application.

The main steps of the intra-module phase are shown in Figure 1. Note that these steps are obviously language-dependent. Even though in this paper we focus on applications written in the PHP language, our approach can be easily extended to extract views from modules written in other programming languages.

To better illustrate our technique, we will refer to a simple web application whose code is presented in Figure 2. The application is written in PHP and consists of three modules: `index.php`, which is the application entry point, `create.php`, which allows new users to create an account, and `answer.php`, which provides some information that should be accessible only to registered users. The application state is maintained using both a relational database, which contains the users' accounts, and a PHP session variable, i.e., `_SESSION["loggedin"]`.

Even though the application is very simple, it contains representative examples of the security problems that our approach is able to identify. In particular, the application contains two vulnerabilities. The first vulnerability is caused by the fact that the `index.php` module uses usernames retrieved from the database as part of its output page. Usernames are strings arbitrarily chosen by users during the registration process implemented by the `create.php` module. Since these strings are never sanitized in any module, the application is vulnerable to XSS attacks. The second vulnerability is contained in the `answer.php` module. The module incorrectly checks the value of the `loggedin` variable instead of `_SESSION["loggedin"]` in order to verify the user status. However, if the PHP `register_globals` option is activated and the `_SESSION["loggedin"]` variable has not been defined (i.e., the user is not logged in), an attacker can include a `loggedin` parameter in her GET or POST request, effectively shadowing the session variable with a value of her choosing. This could be leveraged to bypass the registration mechanism and access the restricted `answer.php` module without being previously authenticated, thus violating the intended workflow of the application.

As it is clear from the examples above, these vulnerabilities are carried out in multiple steps and involve multiple modules. The ultimate goal of our analysis is to detect these multi-module vulnerabilities. However, in order to analyze the interactions between modules, it is first necessary to analyze the properties of each module. This analysis is the focus of the rest of this section.

4.1 Control-Flow and Data-Flow Graphs Extraction

The first step of the intra-module analysis is the extraction of the control-flow and data-flow graphs from each module of the application. Our implementation leverages Pixy [9], a static analysis tool for detecting intra-module vulnerabilities in PHP applications. We adopted Pixy's PHP parser, control-flow graph derivation component, and alias analysis component. In addition, we extended Pixy with a data-flow component that computes the def-use chains for a module using a standard algorithm [1]. The resulting tool provides all the information needed for the following steps of the analysis. The main limitation of Pixy, besides being limited to intra-module analysis only, is the lack of support for object-oriented code. Where needed, we manually pre-processed input modules to work around this problem.

4.2 Database Analysis

Databases are often used by web applications to store data permanently. This data is usually accessible by every module of the application. Therefore, it is important to characterize module-database interactions as they could be leveraged to perform a multi-module attack.

The goal of the database analysis is to translate the interaction between an application module and the back-end database into a set of variable assignments. By doing this, the following steps of the analysis (e.g., the view extraction process) can handle database operations and assignments to variables in a uniform way.

For example, consider the following SQL query that writes the content of the variable `uname` to the column `username` in the database table `users`:

```
UPDATE users SET username=$uname WHERE...
```

As a result of the database analysis, a new assignment is added after the call to the function that executes the query. In our example, MiMoSA generates the following assignment node:

```
$DB_dbname_users_username = $uname;
```

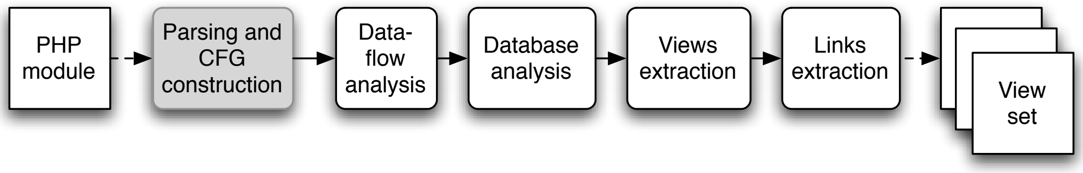

Figure 1: The main steps of the intra-module analysis. The parts in gray are implemented by Pixy.

Note that `DB_dbname_users_username` is a new variable created by our analysis to model the part of the database modified by the `UPDATE` operation.

The PHP language provides a number of internal functions to connect to different types of relational databases. In our prototype implementation, we focused on the MySQL library because of its popularity. However, if the target application uses a different database, our technique can be easily adapted to address a different set of primitives. In PHP, access to the MySQL database is usually performed by first calling the `mysql_query` function to execute a query, and then by using one of the `mysql_fetch` functions to access the results of the query in an iterative fashion.

The main challenge in the database analysis is to properly reconstruct the values that a query can assume at runtime, so that we can determine the tables and columns that are modified by the operation. To achieve this, we traverse the control-flow graph of the module, looking for calls to the `mysql_query` function. Since, in general, static analysis cannot provide the value that the query will assume at runtime, we apply a dynamic analysis technique to the block of PHP code that precedes the function call to derive the names and fields of the tables involved in the query. The analysis extracts the largest deterministic path $\widetilde{P}$ that precedes the `mysql_query` call. A deterministic path is a sequence of nodes in the control-flow graph that only contains branch instructions whose conditional expressions can be statically determined. We then remove from $\widetilde{P}$ any input/output related operation, and we replace any undefined variable in $\widetilde{P}$ with a placeholder.

The resulting code is passed to the PHP interpreter in order to dynamically determine the value that the query string can assume along the path $\widetilde{P}$. If the resulting query performs an *UPDATE* or an *INSERT* operation, it is immediately parsed to extract the assignment nodes as shown before. Queries that contain a *SELECT* statement are instead analyzed only when the analysis finds that the corresponding `mysql_fetch` function is used to assign the result values to one or more PHP variables.

Consider for instance the `mysql_fetch_assoc` call at line 16 of `index.php` of our sample application. Following the data-flow edges we reach the corresponding query string at line 12. The dynamic analysis along the deterministic path reconstructs the query `"SELECT * FROM users"`. The database analyzer then checks the database schema to resolve the `"*"` symbol to the corresponding list of column names and it finally generates the resulting assignments nodes:

```
$row["username"] = $DB_dbname_users_username;
$row["password"] = $DB_dbname_users_password;
```

Once these assignments are introduced to the module, the following analysis steps are able to treat the application state stored in a back-end database and the state stored in program variables in a uniform way.

4.3 Views Extraction

The goal of this step is to summarize a module into a set of views. This is a key step in our intra-module analysis, because it produces the module meta-information necessary to perform the inter-module vulnerability analysis.

To extract a module's views, we first perform *state analysis* to determine all statements in the control-flow graph that are state-related, i.e., that either contain state entities or are control- or data-dependent on state-related statements. We consider state entities of a PHP application the variables used to refer to request parameters (`_GET, _POST, _REQUEST`), cookies (`_COOKIE`), session variables (`_SESSION`), and the database variables generated by the database analysis step. This allows us to exclude from further analysis statements that do not depend on or modify the application state. Therefore, in the rest of the analysis we consider only the subgraph of the CFG that contains state-related nodes. The algorithm we use in this step is based on the functional data-flow analysis framework of [19], as implemented in Pixy.

4.3.1 Identifying Sinks and State Entities

To identify sinks, we determine all nodes in the CFG that contain an operation relevant to our analysis. In particular, we look for two types of operations: state-related operations and sink-related operations. State-related operations are those statements that modify the server-side state. For example, we identify uses of the session mechanism, that is, assignments to the `_SESSION` array or calls to the `session_register()` function. Sink-related operations are statements where state entities are used in sensitive sinks. Our technique focuses on identifying inter-module XSS and SQL injection attacks, and, therefore, we keep track of state entities displayed to the user or used in a database query. Consider, for example, the `create.php` module in our example. The analysis identifies two relevant operations: at line 19, a database query is executed, and, at line 21, the variable `_SESSION["loggedin"]` is modified.

After the relevant operations have been identified, we derive their conditional *guards*, i.e., the conditions associated with the branches in the CFG that must be taken in order to reach the statement associated with the operation. Note that we only keep track of state-dependent conditions, as identified by the state analysis. In our example, the two operations that we identified in `create.php` are guarded by the conditional statement at line 9. The analysis also recognizes that the true branch of the conditional must be taken to trigger the operations.

Then, for each variable that occurs in a conditional guard or in a state- or sink-related statement, we reconstruct its dependency with respect to state entities. We currently model several types of dependencies. In particular, *propagation dependencies* model the assignment of one variable to another; *call dependencies* denote the fact that a variable takes its value from the result of a function call (in particular, we currently model sanitization functions); *binary dependencies* model the composition of two variables, for

```
1  <html>
2  <head>
3     <title>The answer to Life, the
4     Universe, and Everything</title>
5  </head>
6
7  <body>
8
9  <?php
10    echo "People that know the answer:";
11
12    $sql = "SELECT * FROM users ";
13    mysql_select_db("dbname");
14    $res = mysql_query($sql);
15
16    while($row =  mysql_fetch_assoc($res))
17       echo $row["username"];
18 ?>
19
20 <a href="create.php">Create User</a>
21
22 </body>
23 </html>
```

index.php

```
1  <?php
2     session_start();
3
4     if ($loggedin != "ok") {
5        header("Location: index.php");
6        exit;
7     }
8
9     echo "42";
10 ?>
11
12 <html>
13 <head>
14    <title>The final answer is:</title>
15 </head>
16
17 <body>
18  <a href="index.php">Homepage</a>
19 </body>
20 </html>
```

answer.php

```
1  <html>
2  <head>
3     <title>Create a new user</title>
4  </head>
5
6  <body>
7
8  <?php
9     if (isset($_POST["user"])) {
10
11       $user = addslashes($_POST["user"]);
12       $pass = addslashes($_POST["pass"]);
13
14       session_start();
15
16       $sql = 'INSERT INTO users ' .
17          'VALUES (\'' . $user .
18          '\', \'' . $pass . '\' )';
19       mysql_query($sql);
20
21       $_SESSION["loggedin"] = "ok";
22
23       header("Location: answer.php");
24       exit;
25    }
26 ?>
27
28 <form action="create.php"
29       method="POST">
30
31    UserName:
32    <input name="user" type="text"><br>
33    Password:
34    <input name="pass" type="password"><br>
35    <input name="create" type="submit">
36
37 </form>
38
39 </body>
40 </html>
```

create.php

```
Table: users
+----------+-------------+
| Field    | Type        |
+----------+-------------+
| username | varchar(32) |
| password | varchar(32) |
+----------+-------------+
```

Database schema

Figure 2: Example application.

example through mathematical or string operators; *constant dependencies* denote that a variable takes a constant value; *superglobal dependencies* indicate that a variable takes a value from one of the superglobal objects in PHP, e.g., from a request or session variable. Multiple dependencies are composed together until each variable is reduced to either a constant or a state entity.

Note that an additional set of conditional guards can be discovered during the dependency reconstruction analysis: for example, a variable used in an operation might assume different values depending on some conditions. Such conditions are added to the set of conditional guards for the operation.

In our example, the variable `_SESSION["loggedin"]`, used in the state-related statement at line 21 in `create.php`, is associated with a constant dependency that models the fact that it was assigned the constant value `ok`. The conditional guard at line 9 is reduced to the composition of a call dependency (to the `isset()` function) and a superglobal dependency (to the `_POST["user"]` variable).

4.3.2 Creating the View

After all sensitive operations and their complete set of conditional guards have been identified, we translate them into pre-conditions, post-conditions, and sinks. Currently the following predicates are used in pre-conditions: *Exist(v)* is true if and only if the entity *v* is defined in the current application state. *Compare(v, u, op)*, where *v* and *u* are state entities and *op* is an operator, is true if and only if the expression *v op u* is true. MiMoSA currently supports the operators $<$, $>$, $=$, and their combinations. The *Propagate* predicate is used in post-conditions: *Propagate(v, u, San)* denotes that the value of the entity *v* is propagated to *u* applying the sanitization operations specified by the set *San*. For sinks, the following predicates are used: *InSql(v)* denotes that the state entity *v* is used in a SQL query; *Displayed(v)* indicates that *v* is displayed to the user. Conditions can be combined with the use of *and*, *or*, and *not* operators.

In addition, we introduce the special *Unknown* predicate, which is assumed to be always satisfiable, to model the cases where we cannot resolve the dependency of a program variable to a state entity. This happens, for example, when a variable takes its value from a complex series of calls to functions that we do not model.

As an example of the view creation process, consider the module `create.php` of our sample application. MiMoSA summarizes it into two views, corresponding to the two branches of the conditional statement at line 9. One view (corresponding to the false branch) has pre-condition *not Exist($_POST["user"])* and empty post-conditions and sinks. The other view (corresponding to the true branch) has pre-condition *Exist($_POST["user"])*. The assignments introduced by the database analysis step to model the SQL query at line 19 are modeled with the post-conditions *Propagate($_POST["user"], DB_dbname.users.username, {addslashes})* and *Propagate($_POST["pass"], DB_dbname.users.password, {addslashes})*. In both cases, the analysis keeps track of the sanitization operated by the `addslashes()` function. Finally, the assignment to the session variable `_SESSION["loggedin"]` is modeled with the post-conditions *Exist($_SESSION["loggedin"])* and *Propagate("ok", $_SESSION["loggedin"], ∅)*. The complete set of views for our example application is shown in Table 1.

In a module, the number of extracted views is exponential in the number of state-related conditional statements. As a consequence, the view extraction process is slow when dealing with very complex modules. Therefore, whenever the number of views is determined to be larger than a certain threshold, MiMoSA can be configured to switch to a simplified view construction approach. In this approach, instead of generating views for all the paths in the CFG of a module, we only generate the views corresponding to a number of paths sufficient to include all the state- and sink-related operations contained in the module. As a result, all the post-conditions and sinks of the module are extracted and will be analyzed during the detection phase. However, since not all their possible combinations are considered, the simplified approach might introduce inaccuracies.

4.4 Links Extraction

The last step before starting the inter-module vulnerability analysis is to extract the links contained in the module and associate them with the views they belong to.

We parse both PHP and HTML code looking for HTML hyperlinks, form actions and inputs, source attributes of frames, and calls to the PHP function `header()`[5]. We also have a limited support for link extraction from JavaScript code. If the URL of the link is dynamic, i.e., it is generated using a block of PHP code, the link extraction routine tries to determine its runtime value by applying a dynamic analysis technique similar to the one used in the database analysis phase.

Once all the links have been extracted, we identify the set of views to which each link belongs. In order to do this, we determine the conditional branches in the CFG that must be taken in order for a link to be shown to the user and we compare these branch expressions with the pre-conditions of the extracted views. Consider, for instance, the link to `answer.php` contained in the `create.php` module of our example application. Our analysis recognizes that it is displayed only if the execution follows the true branch of the conditional statement at line 9. `<create.php>.view_0` is the only view compatible with this execution and, therefore, it is identified as the source view of the link.

To correctly model the application workflow, in addition to having the names of the modules to which one can navigate from a given view, we also need to extract the set of inputs that are submitted along the link. In particular, we need to determine which GET and POST requests parameters are submitted if a user follows the link. For example, in our sample application, if a user submits the form at line 28 of the `create.php` module, the user-provided parameters `user` and `pass` are submitted as a part of the POST request to `create.php`.

5. INTER-MODULE ANALYSIS

In the second phase of our analysis, we connect the views extracted during the intra-module analysis into a single graph. This graph models the intended workflow of the entire web application. We then use a model checking technique to identify multi-module data-flow vulnerabilities and violations of the intended workflow.

[5]The `header()` function in PHP is commonly used to set the HTTP `Location` header to redirect users to a different page.

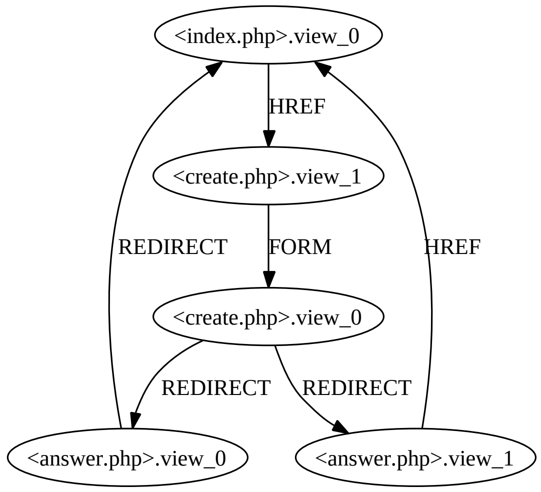

Figure 4: Intended workflow of our example application.

The main steps of the inter-module phase are shown in Figure 3. Note that since this phase is built on top of the view abstraction, it is completely independent of the programming languages in which the modules are developed.

5.1 Intended Workflow

In the first step of the inter-module phase, we use the link information extracted during the intra-module analysis to connect all the views of the application into a single graph.

We connect a source view V_i to a target view V_j if V_i contains a link `l` that references V_j's module and the parameters provided by `l` satisfy the pre-condition of V_j. In particular, we adopt the following two rules:

1. If V_j's pre-condition contains predicates over client-side state entities, we check that the extracted link satisfies these requirements. For example, if the pre-condition requires the presence of a particular GET parameter, we check that the link provides a parameter with the required name.

2. If V_j's pre-condition contains predicates over server-side state entities, we assume that these predicates are always satisfied. The rationale is that, in general, it is not possible to determine the extended state of the application considering the two views in isolation, because it depends on the path that the user has followed to reach V_i. Therefore, we conservatively assume that the state can satisfy V_j's pre-condition.

When both conditions are satisfied, we assume that there is an intended path between the two views and we connect them together. For example, the link in `<index.php>.view_0` (line 20) is connected to the view `<create.php>.view_1` but not to `<create.php>.view_0`. In fact, the pre-condition of `<create.php>.view_0` requires the existence of a POST parameter named `user` that is obviously not provided if the user clicks on the link in `index.php`. The intended workflow for our example application is given in Figure 4.

Finally, the analysis identifies the application's *entry points*. We exclude the modules that appear inside an `include` statement from this step of the analysis, because they are generally not intended to be directly accessed by the user. Of the remaining modules, we consider as entry point any view that has either an empty pre-condition or a pre-condition that contains only predicates over `GET` parameters (see Section 3).

Module	View ID	Pre-conditions	Post-conditions	Sinks
`index.php`	*view*_0	∅	∅	*Displayed(DB_dbname.-users.username)*

`create.php`	*view*_0	*Exist($_POST["user"])*	*Propagate($_POST["user"], DB_dbname.users.username, {addslashes})*
			
			Propagate($_POST["pass"], DB_dbname.users.password, {addslashes})
			
			Exist($_SESSION ["loggedin"])
			
			Propagate("ok", $_SESSION["loggedin"], ∅)
				∅

`create.php`	*view*_1	not *Exist($_POST["user"])*	∅	∅
`answer.php`	*view*_0	not (*Exist($loggedin)* and *Compare($loggedin, "ok", =)*)	∅	∅
`answer.php`	*view*_1	*Exist($loggedin)* and *Compare($loggedin, "ok", =)*	∅	∅

Table 1: Views generated for the example application of Figure 2.

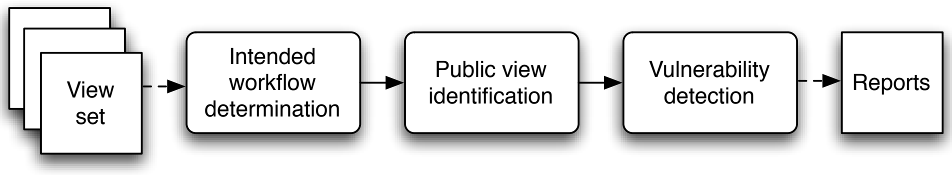

Figure 3: The main steps of the inter-module analysis.

Unfortunately, in some cases it is not possible to differentiate between an application's entry point and the developer's failure to put the necessary safety checks into a module. For example, in our experiments we tested a web application where in one of the administration pages the developer forgot to put a check to verify that the user was actually logged in as administrator. Our technique classified the views of this module as entry points since they did not have any pre-condition at all. Nevertheless, the user of our tool can easily detect these vulnerabilities by inspecting the automatically generated list of entry points.

5.2 Detecting Public Views

The *intended path* introduced in Section 3 did not model a very important concept of a web application: the existence of *publicly-accessible* pages. These pages (such as the FAQs pages) are very common in many web sites but they are rarely intended as entry points to the application. Therefore, we do not generate any security alert if it is possible to access these pages violating the intended workflow of the application.

For this reason, we adopted the following rules to detect and mark the publicly-accessible views:

- Starting from one of the application entry points, all the views that are reachable along some intended path traversing only views that have empty post-conditions are marked as *public*. This models the fact that if it is possible to reach a view through a path that does not change the extended state of the application, the access to the view is not supposed to be restricted.
- Any empty redirect view is *public*. An empty redirect view is a view that does not have any post-condition, any sink, and only contains a redirect link. This models all the views used to detect and redirect unauthenticated users that try to access a restricted page.

In the example, our algorithm marked `<create.php>.view_0`, `<create.php>.view_1`, and `<answer.php>.view_0` as public views. The first two because they are reachable without any change in the application state and the last one because it is an empty redirect.

5.3 Detection Algorithm

Our graph exploration mechanism simulates a user that moves from one view to another. At each step, we select a new view to add to the current path, we evaluate its pre-condition against the current state, and, if the pre-condition is satisfied, we update the state to reflect the effects of the view's post-conditions.

Each path is analyzed to check if it satisfies the definition we provided in Section 2 for multi-step data-flow vulnerabilities and workflow violations. In general, if the graph is correct, it is possible to find all the vulnerabilities simply by trying each possible *navigation path* in the application. Our solution is similar to a model checking approach, and, unfortunately, it suffers from the same path explosion problem. Therefore, we limit our analysis to paths that contain up to one loop and with a total length limited by a user-defined upper bound. In our experiments, in fact, we observed that most of the vulnerabilities can be exploited using a very limited number of steps (usually less than 5).

Our detection algorithm traverses the graph following the intended paths. At each step it checks if it is possible to jump to one of the views that should not be reachable from the current position. If it succeeds, it raises a workflow violation alert and it does not go any further along that path. This means that some vulnerabilities may not be discovered because they are hidden "behind" other vulnerabilities. In this case, the user should fix the discovered vulnerability and run the analysis again.

By applying MiMoSA to our sample application, we identify the two existing vulnerabilities. Figure 5 shows the reports produced by MiMoSA for the example application of Figure 2.

```
Workflow Violation:
Path:
  - index.php[view_0]
  - answer.php[view_1]
```

```
DISPLAY of unsanitized entity:
Entity: DB_dbname.users.username
Example of Exploitable Path:
    - create.php[view_0]
    - index.php[view_0]
```

Figure 5: Vulnerabilities detected in the sample application of Figure 2.

6. EVALUATION

To prove the effectiveness of our approach in detecting multi-module data-flow vulnerabilities and violations of the intended workflow of a web application, we ran our tool on five real-world web applications.

The selected applications satisfy three requirements: i) they are written in PHP and they contain multiple modules, ii) they use both session variables and database tables to maintain the application state, and iii) they do not contain object-oriented code. The list of chosen applications is shown in Table 2. The table also shows the list of known vulnerabilities for each application.

For each application we ran the intra-module analysis in order to extract the set of views corresponding to the application modules. We then ran the inter-module analysis to connect together the views and calculate the intended application workflow. Finally, we applied our detection algorithm to find anomalies in the possible navigation paths and to detect multi-module data-flow vulnerabilities.

The results of our tests are summarized in Table 3. For the intra-module phase, the table reports the number of views extracted and the time required by the analysis[6]. In the inter-module phase, we explored up to one hundred million paths, covering at least all the paths of length 3. The table reports the time required to generate the paths and the alert messages raised by our tool. The alerts are grouped according to the entities involved (for the data-flow vulnerabilities) and the modules (for the workflow violations). For both data-flow and workflow vulnerabilities, we report the number of violations detected by our tool, the number of false positives, and how many of the remaining violations correspond to exploitable vulnerabilities.

MiMoSA was able to find all the known vulnerabilities and to discover several new ones.

With regard to multi-module data-flow vulnerabilities, we had only one false positive. In fact, in the MyEasyMarket application, the PHP variable `REMOTE_ADDR` is saved in the database and later printed to the user. Even though the value of the variable is never sanitized, it is automatically set to the IP address of the client's machine by the PHP engine. Therefore, it only has a limited range of valid values (numbers and dots) that do not allow a user to mount an attack against the application.

[6]All the experiments were executed on a Pentium 4 3.6GHz with 2G of RAM.

MiMoSA also reported several violations of the intended workflow of the web applications. Even though in most of the cases they corresponded only to anomalous paths into the application (e.g., directly jumping from the login to the logout page), we were also able to confirm that some of the reported violations correspond to actual vulnerabilities that could be exploited to gain unauthorized access to a restricted page.

While the inter-module analysis is the more time consuming phase, the intra-module analysis is certainly the more fragile, since it is where the static analysis techniques that we use introduce most of the approximations. Any imperfection in this phase can result in an increasing number of both false positives and false negatives. For instance, during the construction of the intended paths, we observed that some of the views were isolated, with no connection to any other part of the application[7]. This was probably caused by an error in the view extraction, such as a missing link or a wrong pre-condition predicate.

To better test the accuracy of our intra-module analysis and evaluate its impact on the final results, we selected one of the applications in our test suite (i.e., SimpleCMS) and manually analyzed the output of each step of the view extraction phase. The results are shown in Table 4. MiMoSA achieves a high accuracy in the extraction of database operations, links, post-conditions, and sinks. Also the rate of unknown conditions, i.e., the pre-conditions that MiMoSA was not able to correctly reconstruct, is reasonable, considering that we are using a static analysis technique.

In this application, the number of generated views is, instead, considerably higher than the number of views actually present in the application code. This happens because of two main reasons. First, MiMoSA might generate views corresponding to paths that are infeasible in the program, such as the ones that traverse nodes with conflicting conditions. The presence of these views does not affect the final results since they are never entered during the detection phase. The second reason is that MiMoSA can generate duplicate views, i.e., views with different but equivalent pre-conditions. Even though this may lead to inaccuracy in the final results, in most of the cases its main effect is just to slow down the path generation phase.

7. RELATED WORK

In the introduction, we briefly mentioned some recent works in the areas of intrusion detection and application firewalls that focus on detecting and blocking web-based attacks. Since our work focuses on vulnerability analysis, and, consequently, deals with a different class of problems than the detection of attacks at runtime, we are not going to further review these works here.

There is a number of recent works in the area of vulnerability analysis of web-based applications. Most of these approaches are based on taint propagation analysis applied to application written in PHP [7, 9, 10, 22] or Java [6, 14].

The WebSSARI tool [7] is one of the first works that applies static taint propagation analysis to find security vulnerabilities in PHP. WebSSARI targets three specific types of vulnerabilities: cross-site scripting, SQL injection, and general script injection. The tool uses flow-sensitive, intra-procedural analysis based on a lattice model and typestate. When the tool determines that tainted data reaches sensitive functions, it automatically inserts *runtime guards*,

[7]These views were not taken into consideration by our path exploration algorithm since they could not provide any useful information to the user.

Application Name	PHP Files	Description	Known Vulnerabilities
Aphpkb 0.71	59	Knowledge-base management system	–
BloggIt 1.01	24	Blog engine	CVE-2006-7014
MyEasyMarket 4.1	23	On-line shop	–
Scarf 2006-09-20	18	Conference administration	CVE-2006-5909
SimpleCms	22	Content management system	BID 19386

Table 2: PHP applications used in our experiments. Vulnerabilities are referenced by their Common Vulnerabilities and Exposures ID (CVE) or their Bugtraq ID (BID).

Application	Intra-Module Analysis		Inter-Module Analysis				
	Views	Time	Time	DF Violations-(FP)	DF Vulnerabilities	WF Violations-(FP)	WF Vulnerabilities
Aphpkb	4680	31:24m	3:00h	0-(0)	0	17-(10)	-
BloggIt	339	2:12m	0:31h	14-(0)	14	3-(0)	-
MyEasyMarket	449	1:12:00h	6:36h	2-(1)	1	1-(0)	1[a]
Scarf	1721	7:30m	1:10h	3-(0)	3	3-(0)	1
SimpleCms	417	0:22m	2:50h	8-(0)	8	5-(0)	4

[a] Detected through inspection of the entry point list, as discussed in Section 5.1.

Table 3: Results of the experiments. DF: Data Flow, WF: Work Flow, FP: False Positives.

i.e., sanitization routines.

Xie and Aiken [22] use intra-block, intra-procedural, and inter-procedural taint propagation analysis to find SQL injection vulnerabilities in PHP code. This approach uses *symbolic execution* to model the effect of statements inside functions. These effects are summarized into the pre- and post-condition sets for each analyzed function. The function pre-conditions contain a derived set of memory locations that have to be sanitized before the function invocation, while the post-conditions contain the set of parameters and global variables that are sanitized inside the function. To model the effects of sanitization routines, the approach uses a programmer-provided set of possible sanitization functions, considers certain forms of casting as a sanitization process, and, in addition, keeps a database of sanitizing regular expressions, whose effects are specified by the programmer.

Pixy [9,10], which we have described in Section 4.1, specifically targets the identification of intra-module XSS vulnerabilities. This tool seems to be the most complete static PHP analyzer in terms of the PHP features modeled. To the best of our knowledge, it is the only publicly available tool for the analysis of PHP-based applications.

None of the described approaches performs inter-module analysis, that is, all the vulnerabilities identified by these approaches are local to a single application module. Unlike our approach, these techniques do not have any notion of the application's extended state, and, therefore, they are unable to capture the workflow vulnerabilities described in Section 2. By considering all inputs generated from outside of an application as being tainted, these approaches should be able to identify some types of multi-module data-flow vulnerabilities. However, because of the locality of the analysis, they are incapable of tracing the origins of multi-steps attacks, and, as a result, are subject to a much higher false positive rate.

There is also a number of works that apply dynamic analysis techniques to the analysis of web-based applications. For example, approaches that use dynamic taint propagation analysis, conceptually similar to Perl's taint mode but often with a more refined granularity, have been applied to other languages as well: Nguyen-Tuong et al. [15] propose modifications of the PHP interpreter to dynamically track tainted data in PHP programs, and Haldar et al. [5] apply a similar approach to the Java Virtual Machine.

Pietraszek and Vanden Berghe [16] present a unifying view of injection vulnerabilities and describe a general approach for detecting and preventing injection attacks. This approach is based on instrumenting the platform, such as the PHP interpreter, to track the flow of untrusted data inside the applications. A context-sensitive string evaluation is then performed at each sensitive sink to detect injection attacks.

All dynamic approaches described above either are able or, at least in theory, can be extended to detect multi-module data-flow attacks. The main difference with our approach is that we are able to detect such vulnerabilities statically, considering all the possible application's paths. Also, none of these approaches can detect workflow vulnerabilities because they do not model or take into account the application's intended workflow.

There are also several recent approaches that try to identify SQL injection attacks by building models of legitimate queries that can be performed by an application and comparing these models to the dynamically-generated queries. Whenever these queries structurally violate the static model, an attack is detected. For example, the AMNESIA tool [6] targets SQL injection attacks in Java-based applications. AMNESIA defines a SQL injection attack as the attack in which the logic or semantics of a legitimate SQL statement is changed due to malicious injection of new SQL keywords or operators. Thus, to detect such attacks, the semantics of dynamically-generated queries is checked against a derived model that represents the intended semantics of the query.

Su and Wassermann [20] propose another approach that uses the syntactic structure of the program-generated output to identify *injection attacks*, such as XSS, XPath injection, and shell injection attacks. The current implementation, called *SqlCheck* is designed to detect SQL injection attacks only. The approach works by tracking sub-strings from the user input through the program execution. The tracking is implemented by augmenting input strings with special characters, which mark the start and the end of each sub-string. Then, dynamically-generated queries are intercepted and checked

Views		Accuracy				Rate of
Extracted	Optimal	DB Operations	Links	Post-Conds	Sinks	Unknown Conditions
417	47	96%	78%	100%	100%	15%

Table 4: Accuracy of the view extraction step for SimpleCMS.

by a modified SQL parser. Using the meta-information provided by the sub-string markers, the parser is able to determine if the query's valid syntactic form is modified by the sub-string derived from user input, and, in that case, it blocks the query.

Both AMNESIA and SqlCheck can successfully detect SQL injection attacks at the time of injection; however, without a significant implementation effort, none of them can detect data-flow vulnerabilities such as persistent XSS attacks. Obviously, both approaches, as being based on the syntactic structure of legitimate output, are incapable of detecting workflow vulnerabilities/attacks.

8. CONCLUSIONS

As web applications that perform security-critical tasks become more sophisticated, there is an increasing need for techniques and tools that can address the novel security issues introduced by these applications. In particular, because of the heterogeneous nature of web applications, it is important to develop new techniques that are able to analyze the interaction among multiple application modules and different technologies.

In this paper, we presented a novel vulnerability analysis approach that takes into account the multi-module, multi-technology nature of complex web applications. Our technique is able to model both the *intended workflow* and the *extended state* of a web application in order to identify both workflow and data-flow attacks that involve multiple modules.

We developed a prototype tool, called MiMoSA, that implements our approach and we tested it on a number of real-world applications. The results show that by modeling explicitly the state and workflow of a web application, it is possible to identify complex vulnerabilities that existing state-of-the-art approaches are not able to identify.

Future work will focus on two main directions. First, we will include additional technologies so that we can cover a larger class of applications. Second, we plan to leverage the findings of the static analysis to automatically generate test drivers to reduce the number of the false positives.

Acknowledgments

This research was partially supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] M. Almgren, H. Debar, and M. Dacier. A Lightweight Tool for Detecting Web Server Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 157–170, February 2000.

[3] C. Anley. Advanced SQL Injection in SQL Server Applications. Technical report, Next Generation Security Software, Ltd, 2002.

[4] Common Vulnerabilities and Exposures. `http://www.cve.mitre.org/`, 2006.

[5] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'05)*, pages 303–311, December 2005.

[6] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the International Conference on Automated Software Engineering (ASE'05)*, pages 174–183, November 2005.

[7] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the International World Wide Web Conference (WWW'04)*, pages 40–52, May 2004.

[8] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing Cross Site Request Forgery Attacks. In *Proceedings of the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)*, pages 1–10, September 2006.

[9] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 258–263, May 2006.

[10] N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'06)*, pages 27–36, June 2006.

[11] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 330–337, April 2006.

[12] A. Klein. Cross Site Scripting Explained. Technical report, Sanctum Inc., 2002.

[13] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS '03)*, pages 251–261, October 2003.

[14] B. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium (USENIX'05)*, pages 271–286, August 2005.

[15] A. Nguyen-Tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the International Information Security Conference (SEC'05)*, pages 372–382, May 2005.

[16] T. Pietraszek and C. Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID'05)*, pages 372–382, 2005.

[17] I. Ristic. ModSecurity. `http://www.modsecurity.org/`, November 2006.

[18] D. Scott and R. Sharp. Abstracting Application-Level Web Security. In *Proceedings of the International World Wide Web Conference (WWW'02)*, pages 396–407, May 2002.

[19] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In N. Jones and S. Muchnick, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice Hall, 1981.

[20] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the Annual Symposium on Principles of Programming Languages (POPL'06)*, pages 372–382, January 2006.

[21] G. Vigna, W. Robertson, V. Kher, and R.A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, pages 34–43, December 2003.

[22] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium (USENIX'06)*, pages 271–286, August 2006.