Chainsaw: Chained Automated Workflow-based Exploit Generation

Abeer Alhuzali[*] , Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan
University of Illinois at Chicago
Chicago, IL
{aalhuz2, eshete5, rgjome1, venkat}@uic.edu

ABSTRACT

We tackle the problem of automated exploit generation for web applications. In this regard, we present an approach that significantly improves the state-of-art in web injection vulnerability identification and exploit generation. Our approach for exploit generation tackles various challenges associated with typical web application characteristics: their multi-module nature, interposed user input, and multi-tier architectures using a database backend. Our approach develops precise models of application workflows, database schemas, and native functions to achieve high quality exploit generation. We implemented our approach in a tool called CHAINSAW. CHAINSAW was used to analyze 9 open source applications and generated over 199 first- and second-order injection exploits combined, significantly outperforming several related approaches.

Keywords

Exploit generation; Web security; Injection vulnerabilities

1. INTRODUCTION

Web applications are the engines that drive modern electronic commerce and banking applications. Therefore, security of these applications is an important concern. Any vulnerability that can be exploited in these applications can be catastrophic in terms of financial losses to the online enterprise as well as privacy losses to the consumer. Vulnerability analysis research efforts try to proactively expose the flaws in these applications before attackers can leverage them, and therefore are crucial from a defense standpoint.

Injection vulnerability analysis efforts fall broadly into two categories: penetration-testing approaches and program analysis based approaches. Of these, penetration testing tools work by injecting inputs (which resemble common exploit strings) in the application to check for the presence of vulnerabilities. While they are easy to deploy, they lack broad-coverage of the application and therefore miss vulnerabilities. In fact, recent studies [4] have shown that the coverage of these tools is low.

Static analysis approaches do not suffer from the coverage problem, and are based on more directed exploration of an application's code. In particular, they are focused on identifying properties of certain sensitive sinks in the application. There has been a large body of work (e.g., [17], [20], [28], [26], [9]) for identifying web application vulnerabilities such as SQL injection (SQLI) and Cross-site scripting (XSS). However, due to technical and engineering considerations that go into modeling complex language features, static analysis tools often suffer from false positives.

A direction that has received interest in the recent years is automated exploit generation: approaches that convert *potential vulnerabilities* resulting from a static analysis tool into *concrete exploits*. Because an exploit is an actual input to an application that can be verified, and in most cases automatically, exploit finding has the potential to eliminate false alarms from static tools.

We pursue an automated exploit generation approach in the web application setting in this paper. That is, given an application, the goal is to automatically construct a sequence of malicious HTTP request inputs that direct an application's execution to a vulnerable sink. Given a web application's multi-module and multi-tier nature, the issues for automated exploit generation are quite different from those for binary applications that are studied in [1, 15].

Our main contribution in this paper is an exploit generation technique that is able to 'chain' a sequence of HTTP requests that, when issued in order, direct the execution of the application to a vulnerable sink. Note that the sink may be deeply nested in the module structure of an application that could only be reached by supplying input sequences starting from a publicly accessible module, and such paths may frequently consult data stored in persistent storage.

The starting point for our approach is static analysis: creation of models of the web application behavior along its paths that is based on symbolic execution. From here, there are two scalability challenges that must be overcome to make exploit finding successful. The first one involves *path selection*: what are the paths that we must explore to make opportunistic exploit generation successful? Our approach makes the observation that it is possible to *prioritize* the traversal of the paths by using their constraint solving costs, such that we efficiently identify paths that lead to a successful exploit.

[*]Also affiliated with King Abdulaziz University, Saudi Arabia

CCS'16, October 24-28, 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4139-4/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978380

The second issue is about *persistent database state*: how to deal with database queries that may be present along the paths that are explored. This issue becomes particularly important in the context of *second-order* attack creation, where a vulnerable query, say, is exploited to store some data that is subsequently read from a (second) exploit sink. Our approach develops the use of both static and dynamic techniques for dealing with persistent database state.

Our approach is implemented in a tool called Chainsaw, and to the best of our knowledge it is the first tool that generates injection exploits that span several HTTP requests. Chainsaw is built for the PHP language, but the concepts behind it are general and are applicable to other web platforms. Chainsaw leverages application workflow structures, database schemas and precise modeling of PHP native functions to achieve successful exploit detection, and is capable of detecting second order exploits. Chainsaw was tested on 9 PHP web applications of different complexities, generating 199 exploits, including 30 second-order exploits, and has no false positives by design. A detailed comparison of results from related tools shows that Chainsaw is comparable to, and in the most cases significantly outperforms, other state-of-art approaches.

This paper is organized as follows: Section 2 provides the background, running example and the associated challenges for exploit detection. We present a high-level overview of our approach in Section 3, and a discussion of its implementation in Section 4. We discuss our experiments and results in Section 5. Related work is presented in Section 6 and we conclude in Section 7.

2. BACKGROUND AND CHALLENGES

Assumptions and Goal. We assume the analysis system has complete access to a web application's source code. In order to test exploits, we assume that the analysis system also has a working installation of it, with appropriate login credentials. It can test any generated inputs on the application by sending requests to it and observing the results. The goal is to construct a sequence of malicious HTTP requests that navigate an application to a vulnerable sink in order to perform SQLI or XSS attacks. These attacks are constructed with the goal of exfiltrating information from database, inject malicious content in the database or obtain capability for stealing client side content (e.g., cookies).

Problem Definition. An exploit is successful when a malicious content is injected via a query to a database (SQLI) or when it reaches echo-like statements that send content (i.e. code) to a client (XSS). To trigger such behavior, the malicious content must reach or influence execution at one or more sensitive sinks along some path starting from the input. In web applications, such paths to the sinks may span several modules (sever-side scripts in PHP, similar to servlets in Java) and the attacker may need to send malicious input to all those modules.

The exploit generation problem can be formally stated as follows: Given a web application, find a sequence of pairs $((M_1, I_{E1}), (M_2, I_{E2}),..., (M_N, I_{EN}))$, where each I_{Ei} is the input that must be sent to the corresponding module M_i via an HTTP request to exploit the vulnerable sink in M_N.

2.1 Running example

We introduce an example of a chatroom application, which will be used throughout the paper to illustrate our approach (Listings 1-4). The example contains code snippets from different modules of the application, including several SQLI and XSS vulnerabilities to illustrate the approach.

In particular, room.php starts by including a file with function definitions (line 1). Next, it retrieves user input (lines 4-11) and it sanitizes those inputs using two built-in functions (lines 5, 12). Next, it builds a SQL query with the sanitized version of the user input (line 15). Based on the query results, the superglobal $_SESSION['room_name'] is set (line 17) and the execution proceeds to dashboard.php via the call to the header function (line 18). Superglobals are built-in constructs in PHP that can be accessed from all the modules. The module contains also a self redirection via the HTML form (lines 21-22).

```
 1 include_once('includes/function.php');
 2 if(!isset($_SESSION['username']))
 3   header( "Location: ./login.php" );
 4 if (isset($_GET['mode']))
 5  $mode = htmlspecialchars($_GET['mode']) ;
 6 if (isset($_POST['room_name']))
 7  $room_name = $_POST['room_name'];
 8 else  $room_name = "";
 9 if (isset($_POST['category']))
10  $category = $_POST['category'];
11 else  $category = "";
12 $room_name = mysql_real_escape_string ($room_name);
13 if( isset( $mode ) ) {
14  if( $mode == "enter" ) {
15   $result = mysql_query( "SELECT room_name FROM ROOM_TABLE WHERE
          room_name='$room_name'");
16   if (mysql_num_rows( $result ) == 1 ){
17     $_SESSION['room_name'] = $room_name;
18     header( "Location: ./dashboard.php?mode=addcat&cat_desc=
19     $category");} ....}}
20 else{...
21 <form method="post" action="room.php?mode=enter">
22 ...<td><input type='text' name='room_name'></td></tr>...
23 }
```

Listing 1: room.php (entry module): select the chatroom

```
 1 if(!isset($_SESSION['username']))
 2   header( "Location: ./login.php" );
 3 if (isset($_GET['mode']))
 4  $mode=$_GET['mode'];
 5 else $mode="";
 6 if (isset($_GET['cat_desc']))
 7  $cat_desc=$_GET['cat_desc'];
 8 else $cat_desc="";
 9 if (isset($mode) && isset($_SESSION['room_name'])) {
10  switch ($mode) {
11  case "addcat": header("Location: ./addcat.php?cat_desc=$cat_desc");
12  case "delcat": header("Location: ./delcat.php");
13 ...}}
```

Listing 2: module dashboard.php, dispatch execution to different functionalities (add category, delete category, etc)

```
 1 if(!isset($_SESSION['username']))
 2  header( "Location: ./login.php" );
 3 else if (isset($_SESSION['username']) &&
      isset($_SESSION['room_name'])){
 4  $room_name = $_SESSION['room_name']; // sanitized in room.php
 5  $sql = "SELECT room_name, level FROM ROOM_TABLE WHERE
        room_name='$room_name'" ;
 6  $result = mysql_query($sql);
 7  $room_row = $db->sql_fetchrow($result);
 8  $accesslevel = $room_row['level'];
 9 }
10 if ($accesslevel==1){
11  if (isset($_GET['cat_desc'])) {
12    $cat_desc = htmlspecialchars($_GET['cat_desc']);
13    $sql = "SELECT cat_desc FROM CAT_TABLE WHERE
           cat_desc='$cat_desc'";
14    $result = mysql_query($sql);
15  } }......
16  //check if the category description exists.
```

Listing 3: module addcat.php, retrieve a category description

```
1  if (!isset($_SESSION['username']))
2    header( "Location: ./login.php" );
3  else {
4      if (isset($_POST['room_name']))
5       $room_name = htmlspecialchars($_POST['room_name']);
6      else $room_name = "";
7      if (isset($_POST['level']))
8       $level = intval($_POST['level']);
9      if ($level==1 or $level==2) {
10       $sql = "INSERT INTO ROOM_TABLE (room_name, level) VALUES
              ('$room_name', '$level')";
11       $result = $db->sql_query($sql);
12     } ... }
```

Listing 4: module create.php, create a new chatroom

The code inside dashboard.php serves as a dispatcher to several functionalities of the web application, such as adding or deleting categories, creating chatrooms, and so on. Listing 3 shows a snippet from addcat.php, which checks if a category exists before adding it. The code executes a first query to retrieve the room name and access level (lines 5-6), and next a second query in line 13 if the $accesslevel value retrieved by the first query is equal to 1.

Note that the SQL query at line 13 in Listing 3 has a SQLI vulnerability because htmlspecialchars does not escape the single quote which leaves the query vulnerable to any attack pattern that has a single quote such as `1' OR '1'='1`. The other select queries (line 15 in Listing 1, line 5 in Listing 3) contain no such vulnerabilities since their input is sanitized by a stronger sanitization function (line 12 in Listing 1). The last code snippet contains an insert query, which inserts values of different chatrooms in ROOM_TABLE. Note that this query's input is also insufficiently sanitized.

2.2 Exploit Generation Challenges

Complex Workflow. In general, the execution of vulnerable queries depends on the application workflow, that is on the sequence of visited modules and on the shared state among those modules. In general, there can be several navigation sequences in a web application. Some of them may represent *intended workflows* where the sequence of module execution follows the developers envisioned workflow, while others may be *unintended workflows* where an attacker may issue arbitrary requests and skip the execution of certain modules from the intended sequence. Note that the intended workflows may be exploitable too by changing values that each module expects in input during a redirection. For instance, in our running example, the execution of the vulnerable query in module addcat.php depends on the execution of module room.php, which sets the value of the PHP variable $_SESSION['room_name'], which is checked in addcat.php (line 1). To be able to generate non-trivial exploits, an approach must take into account a wide range of possible workflows both intended and unintended.

Data Sanitization and Path Sensitivity. To generate a working exploit, the transformations and sanitizations of user input along different paths to the sinks must be precisely modeled. For instance, even though the example contains several queries, only two of them are vulnerable because of insufficient sanitization on some paths leading to them. For example, the path going through line 6 in Listing 4 is not vulnerable, while the path going through line 4 is vulnerable. Our approach must therefore take into account the different sanitizations along paths to sensitive sinks and generate exploits that are not prevented by those sanitizations.

Persistent Storage Effect. The database state is another important aspect that must be considered to automatically build a working exploit. In fact, the values stored in the database are additional inputs to the application that influence its control and data paths during execution. For instance, in the running example, the execution of the vulnerable query in line 13 in Listing 3 is dependent on the value of $accesslevel, which is retrieved from the database by a previous query. Therefore, we must be able to take into account the control and data flows that cross from the application to the database and vice versa.

In the next section, we provide details of our approach, which addresses these challenges.

3. APPROACH

In order to successfully generate exploits, we need to tackle the challenge of scale: what are the paths that we must explore to make opportunistic exploit generation successful? Our approach makes the observation that it is possible to *rank* the paths based on their constraint solving costs, and proceeds to construct exploits starting from those that require least analysis and solving complexity.

Our overall approach, as shown in Figure 1, is to gain greater coverage in an initial analysis step and construct working exploits in three subsequent steps. The first step, called *Seed Generation*, considers each module in isolation, and builds a precise model of the computation on its inputs. Specifically, the approach generates *exploit seeds*, which are a verified set of inputs that direct the execution to a vulnerable sink inside a module. Seed generation is local to a module and is guided by a detailed attack specification and database schema specification In the subsequent stages of the approach (i.e. *Workflow Inference* and *Workflow Refinement*), we use a confined exploration strategy focused at deriving a sequence of HTTP requests that navigates the attacker along a sequence of modules to reach one of the exploit seeds generated in the prior step. This is done through methods that selectively explore a subset of paths through the application in order to generate exploits. In the next subsections, we provide a detailed view of these steps.

3.1 Seed Generation

The goal of this phase is to identify sinks that are feasible targets for exploits and to prune out sensitive sinks that are not vulnerable. Specifically, an *exploit seed* is a pair (S, I), where S denotes a sensitive sink that is vulnerable and $I = \{(i_1, v_1), ...(i_n, v_n)\}$ is the set of variable-value pairs that must be sent in input to the module containing S in order to exploit S. The output of this phase is a list of exploit seeds present in various modules of the application. Since this phase represents only the starting point of our approach, we keep its discussion brief.

For every sink inside a module, *Chainsaw* explores symbolically all execution paths from sources (i.e. user input to that module) to that sink (i.e. mysql_query() and echo-like functions) and, for each sink, it builds a symbolic sink expression and a symbolic formula F_P. In the symbolic sink expression, the data arguments are represented as program constants or symbolic values computed from the operations performed on variables used in the sink. In other words, the symbolic execution constructs a symbolic expression for

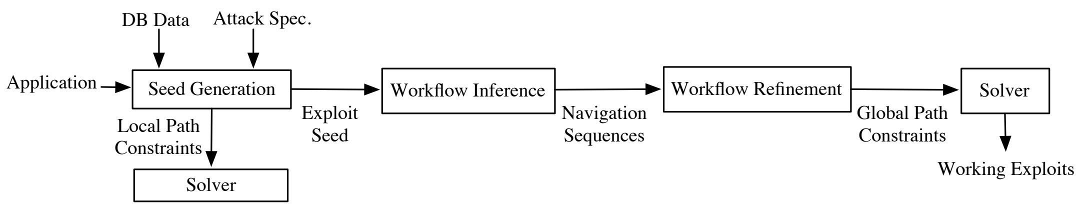


Figure 1: Approach Overview.

static (data arguments are constants) and dynamic sinks (data arguments are variables).

The formula F_P represents the (1) path conditions, (2) input transformations along the paths to the sink and (3) constraints arising out of the sink context. Finally, each symbolic formula F_P is joined with an attack specification F_A, which contains constraints over the variable values at the sinks. For instance, the symbolic formula F_P related to the sink at line 13 in Listing 3, and a possible attack specification F_A are as follows:

F_P: isset($_SESSION['username']) $\wedge$ isset($_SESSION['room_name']) $\wedge$ $room_name==$_SESSION['room_name'] $\wedge$ $accesslevel== 1 $\wedge$ isset($_GET['cat_desc']) $\wedge$ $cat_desc==htmlspecialchars($_GET['cat_desc'])

F_A: $cat_desc=="foo' OR '1'='1"

Our analysis to obtain the symbolic sink and formula is based on our past work [6]. It is inter-procedural, path sensitive as well as context sensitive, based on performing inter-procedural slicing of system dependency graphs (SDGs) [14]. For each sink, the corresponding SDG captures all program statements that construct the queries (data dependencies) and control flows among these statements. We then compute backward slices for sinks such that each slice represents a unique control path from a source to the sink along with path conditions. When this path contains statements in the static-single-assignment (SSA) form (which is the first step in our analysis), constructing components (1) and (2) above of the symbolic formula is straightforward (a discussion of component (3) above appears in section 4.1).

One important detail to mention at this point is that any PHP superglobal present in a module is left unconstrained in this phase, making the analysis local to the module. These superglobals may be constrained in this module, but may be assigned values in other modules. For example, in module `room.php` the superglobal `$_SESSION['username']` is constrained, but is set in `login.php`. To make a successful exploit out of an exploit seed, Chainsaw must find a sequence of module executions that produce appropriate assignments to superglobals in order to direct the execution to the seed sink. We explain this procedure in the next section.

3.2 Navigation Problem

This problem can be stated as follows: Given an exploit seed in a module M_N, is there an execution path along a *navigation sequence* of modules (M_1, ..., M_N) that satisfies the constraints of the exploit seed? In the rest of the paper we call such execution path that spans a navigation sequence (M_1, ..., M_N) *global execution path.* A global execution path is essentially composed as a concatenation of local execution paths belonging to each of the modules M_i. For instance, to trigger an exploit in line 13 in `addcat.php`, a local execution path from `room.php` must be executed first, followed by a local execution path from `dashboard.php`, which is finally followed by the local execution path in `addcat.php`.

One of the main challenges of this problem, indeed the main bottleneck for Chainsaw, is the number of global execution paths that need to be considered for every exploit seed. This number depends on the number of navigation sequences that lead to the module containing that exploit seed and on the number of local execution paths inside each module. More specifically, given an exploit seed inside a module M_N, every navigation sequence (M_1, ..., M_N), contains a number of global execution paths equal to the product of the numbers of local execution paths in each module. In the worst case, these global paths must all be computed and processed during the search. For this reason, to optimize the search for feasible paths, Chainsaw efficiently ranks the possible navigation sequences according to their number of global execution paths, in an increasing order and chooses the next possible navigation sequence based on this order. The intuition behind this choice is to be able to process as many navigation sequences as possible per unit of time.

Specifically, Chainsaw first builds a navigation graph called General Workflow Graph (GWFG), which represents all possible navigation sequences and derives the ranking of the navigation sequences from this graph. Next, for each navigation sequence, Chainsaw builds a Refined Workflow Graph (RWFG) whose goal is to enable the search for global execution paths that satisfy the exploit seed. We explain the two steps below.

3.2.1 Workflow Inference

The General Workflow Graph (GWFG) is a weighted digraph $G = (V, E)$ where each vertex $v \in V$ represents a module of the web application and each edge $e = (v_i, v_k) \in E$ represents a navigation from v_i to v_k. The weight on each edge $e = (v_i, v_k)$ represents the cost (as the increase in the number of global execution paths) incurred by Chainsaw if it were to explore that edge in the navigation sequence. Specifically, to build a list of navigation sequences ranked by the number of global execution paths, we use the *k shortest paths* algorithm [11] on the GWFG, which, given a weighted digraph finds the *k shortest paths* between two nodes ranked by path cost. To reconcile the *k shortest paths* algorithm cost definition as the sum of the weights along the edges with the Chainsaw' s cost definition as the product of the execution paths along the edges, in the GWFG we assign to each edge $e = (v_i, v_k)$ a weight equal to $log_2(n_k)$, where n_k is the number of execution paths in v_k.

For instance, in Figure 2, which represents our example (with one additional module: `check.php`), there are two navigation sequences between `room.php` and `create.php`, one via `dashboard.php` with a total cost of 10 ($= 2^{10} = 1024$ global

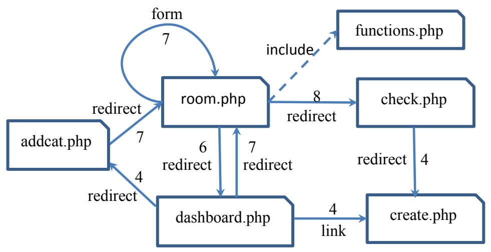


Figure 2: A simplified version of the running example General Workflow Graph. For simplicity, some modules are not shown such as `delcat.php`.

execution paths), and one via `check.php` with a total cost of 12 ($2^{12} = 4096$ global execution paths). Therefore Chainsaw processes the former first, since it has a smaller number of global execution paths.

The GWFG is constructed by identifying statements in the code responsible for navigation (e.g. HTML forms, links, PHP redirection, etc.) and their targets, and adding vertices and edges to the graph correspondingly. In short, the GWFG represents all possible navigation sequences in the application.

The output of the workflow inference module is a set of navigation sequences ranked by the total number of possible global execution paths. The next phase processes each sequence to find the global execution paths that satisfy the constraints of the exploit seed.

3.2.2 Refined Workflow Graph

Once a navigation sequence (M_1,...,M_N) is selected, the next task of Chainsaw is to find a global execution path inside that sequence that leads to an exploit seed. We note that this global execution path is composed as the concatenation of execution paths inside each of the modules in the navigation sequence.

To extract the global execution paths, Chainsaw builds a refined workflow graph (RWFG) for every navigation sequence. The RWFG is a directed graph $G = (V, E)$ where each vertex represents a *local execution path* from one of the modules in the sequence and each edge $e = (v_i, v_j) \in E$ represents a transition from one local execution path to another, that is the fact that the execution path v_j (in module M_j) follows the execution path v_i (in module M_i). Thus, a global execution path corresponds to a path on the RWFG.

An instance of the RWFG, representing the navigation sequence (`room.php`, `dashboard.php`, `addcat.php`), is shown in the left half of Figure 3. In this figure, every module is represented by a set of nodes, where each node represents a local execution path (for space constraints we do not show all local execution paths). A global execution path for this sequence is represented by the RWFG path (L2, A3, D2).

As mentioned before, the number of global execution paths that needs to be considered for every exploit seed is the main bottleneck of Chainsaw. We can reduce this number significantly by removing from the RWFG those edges that are clearly unfeasible. In other words, if we can show that a local execution path A_i in a module cannot follow another local execution path L_i in another module, then we can remove the corresponding edge from the RWFG. To do this, Chainsaw first builds a *path summary* for every vertex v_i of the RWFG. This summary includes the assignments to the superglobals in the path represented by v_i. Next, Chainsaw collects a set of *preconditions* for every vertex v_j. This set contains the conditions that are checked along the path represented by v_j. Using these two sets, Chainsaw does not connect v_i and v_j if there are inconsistencies between the path summary of v_i and the preconditions of v_j. When we cannot determine any inconsistencies, we connect the two vertices. As an example, the table in Figure 3 shows the summaries and preconditions of one possible global execution path that leads to the vulnerable sink in `addcat.php`.

For every global execution path that leads to an exploit sink, Chainsaw creates a symbolic formula that represents the computations and conditions along that global path. Next, it adds the constraints of the attack specification to this symbolic formula using the context of the exploit seed (See section 4.1 for details on context awareness), and uses the solver to look for a solution. The solver solution, if found, contains the values to all local and superglobal variables that need to be sent to every module in the navigation sequence in order to reach the exploit seed.

3.3 Second Order Exploits

The backend database of a web application is an important component that contains a large portion of that application's state. Such state often drives the control and data flow inside the application. When creating the symbolic formulas, Chainsaw includes the state of the database by adding constraints representing the database values. The main challenge in dealing with the database state is that it is always evolving, thus exploits may become feasible or unfeasible depending on it.

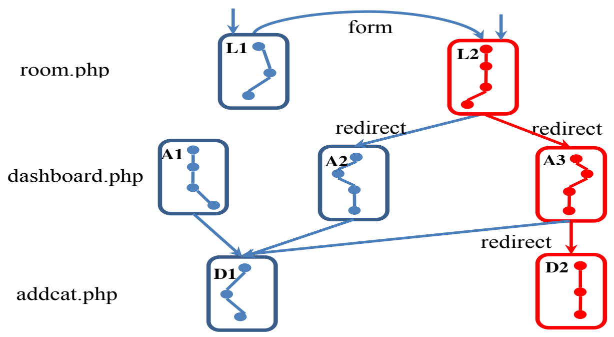
In addition, the database can be used as a channel in second order attacks, such as second-order SQLI and stored XSS. In these attacks, specific content is entered in the database using a SQL sensitive sink with the goal of reaching a second sensitive sink after it is retrieved from the database. This second sink can be another SQL query sink or an XSS sink.

One of the main strengths of Chainsaw is its ability to precisely model the database state. This capability enables a more precise generation of first and second-order exploits. We describe the techniques used by Chainsaw next.

Static Input Generation. Often, the feasibility of an exploit at a sensitive sink relies on conditions along the path to that sink that depend on the database state. For instance, the execution of the query in line 13 of Listing 3 depends on the value of the PHP variable `$accesslevel`, which in turn is retrieved from the database by the query in Line 5. Clearly, if the database does not contain a value equal to 1 for the room level, then execution will not proceed further.

To ensure that the generated exploits are feasible, Chainsaw generates new values for the database, when needed. More specifically, Chainsaw first builds (during analysis) a mapping between *writes* to a database (`insert` and `update` queries) and the `select` queries that read from the same table. Using this mapping, Chainsaw identifies the *write* statement that can be used to insert the needed values in the database. For instance, the `insert` query that inserts the value of the column `level`, is in Listing 4.

As a next step, Chainsaw constructs inputs that will take the application to insert the desired value in the database. More specifically: (i) it sets that *write* operation as a sink and derives the formula F_P relative to that sink, (ii) a constraint formula F_D that represents the desired value to insert, and (iii) sends the formula $F_P \wedge F_D$ to the solver. In our example,



Node	Preconditions	Summary
L2	1.isset($_GET['mode'])∧ 2.isset($_SESSION['username'] ∧ 3.isset($_POST['room_name']) ∧ 4.isset($_POST['category']) ∧ 5.isset($mode) ∧ $mode=="enter"	$_SESSION['room_name']
A3	1.isset($_SESSION ['username'] ∧ 2.isset($_SESSION ['room_name']) ∧ 3.isset($_GET['mode']) ∧ 4.isset($_GET['cat_desc']) ∧ 5.isset($mode) ∧ $mode=="addcat"	∅
D2	1.isset($_SESSION ['username'] ∧ 2.isset($_SESSION ['room_name']) ∧ 3.isset($_GET['cat_desc']) ∧ 4.$accesslevel==1	∅

Figure 3: A Simplified Refined Workflow Graph for the running example (on the left). The table (on the right) shows the preconditions and the summary of the global execution path (nodes: L2, A3 and D2) to the vulnerable sink in `addcat.php`

we ask the solver to provide the input values needed, in order to have the value 1 for the variable `$level` in the `insert` query in line 10 of Listing 4.

For generating second order exploits, the procedure is similar. Note that, for a second order exploit, there are generally two sensitive sinks of interest. The first sink is a *write* query, which inserts a payload in the database. The second sink is another query or statement, which uses the payload data without proper sanitization (e.g., in a second-order SQLI, or a stored XSS). For second order exploit generation, the small addition from the above procedure is that, Chainsaw uses an attack specification F_A that is suitable to generate an exploit at the second sink.

Dynamic Auditing. In addition to *static input generation*, which inserts data in the database, Chainsaw is also able to use existing database data of a live system by including constraints arising from the its state in the symbolic execution. We call this capability *dynamic auditing*. More specifically, Chainsaw augments the symbolic formulas in three steps: **(1)** DB-PHP mapping: by analyzing the symbolic query a mapping is created between PHP variables and column names of the corresponding table. In some cases these mappings can be inferred directly from the WHERE clause construction, which concatenates a database column name with a PHP variable name (e.g., `WHERE room_name='$room_name' in room.php`), while in other cases these mappings can be inferred from the code that processes the query's result set (e.g., the PHP variable `$room_row['level'] is mapped to the database column level in addcat.php`). **(2)** Data retrieval: once a mapping is created, the query is modified by removing the `WHERE` clause and by running a (Select *) query over the same table. This ensures that all possible values are retrieved from the database. **(3)** Constraints creation: a new term T_D is added to the current path formula F_P representing the constraints on the PHP variable values derived from the database.

To illustrate the above procedure, let us consider the query in Listing 3: `SELECT room_name,level FROM ROOM_TABLE WHERE room_name='$room_name'`. Suppose that `ROOM_TABLE` has the following records for the attributes (level, room_name): (1, room1) and (2, room2). In this case, we obtain the following terms:

$T_D = C_1 \vee C_2$ where
$C_1 = \$room\_row['level'] == 1 \wedge \$room\_name == "room1"$
$C_2 = \$room\_row['level'] == 2 \wedge \$room\_name == "room2"$

Static Input Generation vs. Dynamic Auditing. While the static input generation is a general technique, it may create a large overhead. Consider, for instance, the common problem of generating inputs for an `insert` query Q_1 in order to generate one feasible exploit for each query Q_i in a set, each dependent on Q_1. With static input generation, the input generation procedure (that is, path exploration, symbolic evaluation, and constraint solving), for Q_1 must be repeated for as many times as the number of the Q_i-s, while with dynamic auditing, such input generation is not needed, since the data can be directly retrieved from the database. Metaphorically speaking, dynamic auditing is similar to memorization, where results from previous computations are cached and reused when needed. The drawback of dynamic auditing, however, is that the obtained exploits are feasible only for a particular database instance.

For applications with databases that are initially populated from the code (e.g., a series of `insert` queries) rather than from external data entry procedures, static input generation has visibility into the tuples that are inserted and can generate the corresponding constraints. Therefore, for these applications, the results obtained from dynamic auditing are equivalent to those from static input generation with the advantage of smaller overhead.

4. IMPLEMENTATION

Our implementation is written on top of Pixy [18] and TAPS [6] for source code analysis and Z3 [10] for constraint solving. The input to Chainsaw is a web application source, its database schema and the attack specifications. The output is a set of HTTP requests that will exploit vulnerable sinks.

4.1 Context Awareness

Symbolic Parsers. Chainsaw symbolically evaluates each path from a source to a sensitive sink (SQL or XSS) and generates a symbolic expression by representing the sink parameters as symbolic values. When it encounters loops, Chainsaw symbolically executes them 0, 1 or 2 times.

To be able to derive the context of the user input in these symbolic expressions, we implement two additional (symbolic) parsers on top of the SQL and HTML parsers. Our two parsers are able to create the abstract syntax trees of the symbolic expressions and use those trees to derive the context of the user input. For instance, the analysis identifies the context of the variables used in the sink by determining whether each variable is single-, double-, or unquoted. As another example, if the symbolic sink is a `select` query and the user input is used in the WHERE clause, the parser is able to derive the parsing context of user input. Subsequently, this context is used to guide the expansion of the formula F_A with, say, a constraints expressing a tautology SQLI. In addition, it is worth pointing out that, since CHAINSAW automatically includes the semantics of the sanitization functions when creating the symbolic formula, it is sanitization-aware.

Context-awareness example. To see the relevance of the context-awareness, let us consider Listing 5 for an XSS attack scenario.

```
echo "<p> here is the user input $input''</p>;
echo "<p style='\$color\'>This is a paragraph.</p>'';
```

Listing 5: Examples on context awareness in XSS.

For line 1 in Listing 5, the context of the variable `$input` is: *unquoted, paragraph*, and any attack string such as `"<script>alert('xss')</script>"` will trigger the attack. In line 2, however, the user input \$color is the value of an attribute. Therefore, in the attack string, the quote and tag enclosing character (`>`) must first be closed and then followed by the XSS payload (e.g.,`"'><script>alert('xss')</script> "`). The symbolic parsers used in CHAINSAW are able to infer these different contexts and guide the construction of the final formula that is sent to the solver.

4.2 Analysis of Navigation Structure

General workflow graph construction. To construct the GWFG of an application, our approach uses an HTML parser as well as the generated CFG for each module to extract *workflow inference* features: HTTP links, forms, meta tags, iframes and PHP redirection functions. After the extraction, CHAINSAW creates two nodes that represent the source and destination modules of the workflow feature. Since our analysis is inter-procedural, and context-sensitive data and control flow analysis, variables in workflow function arguments can be resolved by consulting the data dependence graph generated for each workflow function. While the GWFG is constructed from HTML and PHP features, our extraction technique can be applied to languages with similar functions.

Informed traversal of the GWFG. The main challenge in traversing the RWFG is to intelligently decide which path to take. In our implementation, we use the notion of *summary history*, which is a mechanism to 'remember' the traversal history and to use it as a guide to search for the next compatible node (execution path). In particular, for each edge traversal from a node v_i to a node v_j, the summary of v_i is added to the summary history (initially empty). Intuitively, the summary history of a path on the RFWG is the collection of all the summaries of the nodes of that path.

A concrete example where we use such summaries is based on Figure 3. The vulnerable query is in two execution paths *D1* and *D2*. If the search starts at *L2*, then there are two possible nodes to traverse next: *A2* and *A3*. Our search technique, chooses *A3* over *A2* because the summary history at *L2* satisfies the preconditions of *A3*. Next, the search adds the summary of *L2* and *A3* to the summary history to further guide the search of the next node. After reaching *A3*, the search checks all *A3*'s outgoing edges (*D1* and *D2*) and repeats the same heuristics until it reaches the vulnerable sink in *D2*.

This search technique significantly reduces the number of edges in the RWFG by keeping only the feasible path transitions. In turn, this improves the efficiency of the generation of working exploits for deeply located exploit seeds in a global execution path.

4.3 Database Schema Analysis

In static analysis, modeling data flows that cross persistent storage is not trivial due to the absence of database constraints in the source code. To this end, CHAINSAW analyzes the database schema to capture the additional constraints imposed by the database. Specifically, tables names and columns definitions such as their names, types and values constraints (e.g., *NOT NULL* and length) are retrieved.

As an example, Listing 6 shows an *insert* query, which seems vulnerable. However, if columns `var1` and `var2` in table `TBL` are of type *enum*$\{1, 2\}$, then that query is no longer exploitable because the database will not accept any other values. CHAINSAW captures such database constraints (e.g., (`$v1==1` $\vee$ `$v1==2`) and (`$v2==1` $\vee$ `$v2==2`)) and adds them to PHP variables used in an insert or update query. Our database schema analysis is capable of analyzing a variety of SQL-based database schemas (e.g., MS SQL, MySQL, Oracle, etc.).

```
$v1= $_GET['v1'];
$v2= $_GET['v2'];
mysql_query(INSERT INTO TBL (var1, var2) VALUES ($v1, $v2));
```

Listing 6: Example that demonstrates the need for considering database constraints on PHP variables.

Another source of database constraints occur due to stored procedures. To analyze these, we need to statically analyze the stored procedure to extract constraints. Our implementation currently does not have this support. However, none of our evaluated applications make use of stored procedures.

4.4 Constraint Solving

To generate exploit seeds and the final working exploits, CHAINSAW uses the Z3 SMT [10] solver and its plug-in Z3-str [30]. For each global execution path, CHAINSAW preprocesses the path information to facilitate the translation to solver specifications. To do so, we take advantage of the generated Three-Address-Code (TAC) representation of source code during the CFG construction. Each TAC statement is then encoded as a formula.

Notes on Translation Our tool recognizes and translates binary and unary PHP expressions, built-in PHP functions, and conditional statements. Our approach models the semantics of the 70 most used PHP built-in functions. The support for several string operations in Z3 is coarse, so we have augmented these with our own Z3 specifications to model built-in functions and to infer data types.

As an example, Listing 7 shows a translation of the PHP statement `$cat_desc=htmlspecialchars($_GET['cat_desc'])` in Z3 performed by CHAINSAW. This function replaces certain characters (e.g., single quotes, ampersands) with their HTML entity encodings. To model this function using the solver language, we use the Z3-str `Replace` function. Since this function replaces only the first occurrence of a substring, CHAINSAW replicates the code shown in Listing 7 for each occurrence of a special character.

```
(assert (= temp_0 (Replace $_GET['cat_desc'] "&" "&amp;")))
(assert (= temp_1 (Replace temp_0 "\"" "&quot;")))
(assert (= temp_2 (Replace temp_1 "<" "&lt;")))
(assert (= $cat_desc (Replace temp_2 ">" "&gt;")))
```

Listing 7: Z3 specifications that model the semantic of `htmlspecialchars` PHP built-in function

5. EVALUATION

Subject Applications. We evaluated CHAINSAW on the 9 PHP applications shown in Table 1. The applications are of varying complexity, with SLOC ranging from 323 to 65302. Our dataset covers all test subjects used in two close works to CHAINSAW, namely *Ardilla* [19] and *CraxWeb* [16]. In addition, our dataset covers test subjects from multiple related works (e.g., 2 from [9], 1 from [8], 1 from [26], and 1 from [3]).

Application	PHP Files	PHP SLOC	SQL Queries	DB Tables	XSS Sinks
myBloggie (2.1.4)	56	9090	59	4	160
scarf beta	19	978	20	7	251
DNScript	60	1322	73	7	334
WeBid (0.5.4)	300	65302	616	63	450
Eve(1.0)	8	905	41	2	102
schoolmate (1.5.4)	63	15375	400	15	559
geccbblite (0.1)	11	323	23	1	40
faqforge (1.3.2)	17	1676	54	2	35
webchess (0.9)	29	5219	134	7	504

Table 1: Subject applications of our evaluation.

Setup. CHAINSAW was deployed on Ubuntu 12.04 LTS VM with 2-cores of 2.4GHz each and 40GB RAM. For each application, we first ran the seed generation and then the exploit generation step. We automatically verified each exploit seed and each working exploit to ensure they reach the sink, and manually confirmed each exploit to ensure the exploit had the desired effect. We also measured the run time of CHAINSAW. For these applications, the dynamic auditing mode is equivalent to the static input generation mode, and the former was used for efficiency.

5.1 Overview of Results

Table 2 shows the summary of our results. On the 9 subject applications, CHAINSAW identified a total of 181 exploit seeds and a total of 199 working exploits with no false positives. While 149 (75%) of the working exploits are SQLI exploits, 50 (25%) are XSS exploits. Of all the working exploits, 30 (15%) are second-order exploits. 13 of the 30 second-order exploits are second-order SQLI exploits and the remaining 17/30 are stored XSS exploits. We outperform most related approaches in the precision and coverage of vulnerability detection (Section 5.6).

5.2 Selected First-Order Exploits

For **webchess**, CHAINSAW generated multiple SQLI and XSS exploits located in different modules. One of these is the SQLI exploit generated for the vulnerable query in `viewmessage.php` (see Listing 8). This exploit allows the attacker to view the communication messages of other users. The global navigation sequence for this exploit is as follows:

$$\begin{tikzcd}\boxed{\text{index.php}} \arrow[r, "\text{form}"] & \boxed{\text{mainmenu.php}} \arrow[r, "\text{form}"] & \boxed{\text{viewmessage.php}}\end{tikzcd}$$

The number of generated constraints is 203 with exploit generation time of 156 sec. The exploit string is shown in Listing 21 in the Appendix. Note, for demonstration purposes, we represent all exploits as GET requests.

```
1 if(isset($_POST['messageID']))
2   {$messageID = $_POST['messageID']; //no sanitization
3   $SqlQuery="SELECT * FROM communication WHERE commID=$messageID";}
```

Listing 8: Vulnerable query in viewmessage.php of *Webchess*.

In `Addcat2.php` (see Listing 9) of the **DNScript** application, an attacker can insert a malicious domain name to the database. `AddCat2.php` does not apply sanitization on the user input before using it in the query. The navigation sequence is as follows:

$$\begin{tikzcd}\boxed{\text{Login.php}} \arrow[r, "\text{form}"] & \boxed{\text{DoLogin.php}} \arrow[r, "\text{header}"] & \boxed{\text{index.php}} \arrow[r, "\text{include}"] & \boxed{\text{inc/nav.php}} \arrow[r, "\text{link}"] & \boxed{\text{AddCat2.php}}\end{tikzcd}$$

CHAINSAW generated 2 exploits for this query. The first exploit follows the *intended workflow* and has a path length of 4. The number of constraints collected from the five modules is 27. Due to the presence of a CAPTCHA in `login.php`, to generate a working exploit, we made very minimal changes to the source code so that it generates a valid CAPTCHA string. The other exploit generated by CHAINSAW for the same query follows a shorter programmer unintended workflow path which hits `AddCat2.php` (a publicly accessible page) with exploit path length equals to 1 and 3 constraints. As we noted before, while most exploit paths follow the intended workflow of an application, an equally feasible exploit can be generated by breaking the workflow. In this particular context, the alternative exploit path requires visiting 1 module (compared to 4 modules).

```
1 $values = 'VALUES ("'.$_POST['cat_name'].'")'; //no sanitization
2 $insert = mysql_query("INSERT INTO gen_cat (cat_name) " .$values);
```

Listing 9: AddCat2.php in `DNScript`.

```
1 http://host/DNScript/AddCat2.php?cat_name=1' OR '1'='1
```

Listing 10: Exploit for the query at line 2 in Listing 9.

For **EVE**, CHAINSAW generated 5 SQLI in two modules:
Case 1: In `edit.php`, CHAINSAW generated an SQLI exploit that has a path length of 1 (`edit.php` is public) and the number of constraints is 40. This exploit enables retrieving other members' information. Listing 11 (line 2) shows the vulnerable query and the generated exploit (line 3).

```
1 $member = $_GET['id']; //no sanitization
2 mysql_query("SELECT MemberID, Name, Division, DateJoined, RankCorp,
      Vacation, Comment, Deleted FROM MembersMain WHERE MemberID=
      '".$member."'";
3 http://host/eveactive/edit.php?id=1' OR '1'='1
```

Listing 11: Exploit for the query at line 2 in `edit.php`.

Case 2: In `member.php`, CHAINSAW generated 4 SQLI exploits, all on `update` queries (see sample in Listing 12) in which the attacker can inject a malicious payload in `$comment` and `$name`. Other variables used in the queries are safe because the other columns in the `MembersMain` table are either integers or of *date* data type. CHAINSAW adds the type of the columns associated with each variable used in a query to the set of the constraints extracted for the source code as discussed in Section 4.3. All 4 exploits have the following navigation sequence:

$$\begin{tikzcd}\boxed{\text{add.php}} \arrow[r, "\text{form}"] & \boxed{\text{member.php}}\end{tikzcd}$$

Application	SQLI Seeds	1st Order SQLI Exploits	XSS Seeds	1st Order XSS Exploits	2nd order SQLI Exploits	2nd order XSS Exploits	Total Exploits
myBloggie	24	21	1	1	7	0	29
scarf	0	0	1	1	0	1	2
DNScript	4	1	1	1	0	3	5
WeBid	42	40	7	7	0	0	47
Eve	5	5	2	2	0	2	9
schoolmate	51	49	5	5	0	2	56
geccbblite	3	3	0	0	3	4	10
faqforge	4	4	4	4	0	2	10
webchess	15	13	12	12	3	3	31
Total	**148**	**136**	**33**	**33**	**13**	**17**	**199**

Table 2: Summary of results for automatic exploit generation.

```
$name = $_POST["name"]; //no sanitization
$comment = $_POST["comment"];//no sanitization
....
mysql_query ("UPDATE MembersMain SET Name = '".$name."', Division =
      '".$division."', RankCorp = '".$rankcorp."', Vacation =
      '".$vacation."', Comment = '".$comment."', Deleted =
      '".$deleted."' WHERE MemberID = '".$memberid."'");
```

Listing 12: Vulnerable query in `member.php, EVE`.

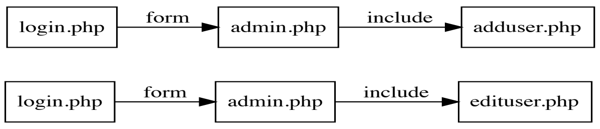
Chainsaw also generated two XSS exploits for `member.php`. Listing 13 shows the vulnerable sink (corresponding exploit string is shown in Listing 22 in the appendix.

```
$nname = $_GET["nname"]; //no XSS sanitization
echo "<font color=\"#00FF00\">User <a href=\"evemail:".$nname.\">";
```

Listing 13: Vulnerable XSS sink in `member.php, EVE`.

5.3 Selected Second-Order Exploits

For **myBloggie**, Chainsaw generated 7 second-order SQLI exploits. One of these exploits is demonstrated in listings 14 and 15. The first step exploits the vulnerable `insert` query in `adduser.php` (Listing 14) where `$user` is not sufficiently sanitized for SQLI vulnerabilities. The second step retrieves user data from `USER_TBL` without sanitization (see Listing 15). Without our comprehensive modeling of PHP functions, constraints that include calls to `isset, intval, md5` and `htmlspecialchars` would not captured correctly and that could result in false positive or negative. The number of generated constraints is 250 with exploit generation time of 174 sec. The exploit string for this sequence is shown in Listing 23 in the Appendix. The navigation to reach and execute the queries in Listings 14 and 15 are as follows:



```
if (isset($_POST['user']))
  $user=htmlspecialchars($_POST['user']);//insufficient sanitization
$password = $_POST['password'];
$repassword = $_POST['repassword'];
$level = intval($_POST['level']); //strong sanitization
$password = md5(trim($password)); //strong sanitization
if ($level==1 or $level==2) {
  $sql = "INSERT INTO ".USER_TBL." SET user='$user',
       password='$password', level='$level'";
  $result = $db->sql_query($sql); }
```

Listing 14: Step 1 (insert) in `adduser.php`.

```
 $sql = "SELECT * FROM ".USER_TBL." WHERE id='$id'";
```

Listing 15: Step 2 (select) in `edituser.php, myBloggie`.

For **geccBBlite** Chainsaw generated 7 second-order exploits in the modules `scrivi.php` and `rispondi.php`. The following discusses one stored XSS exploit, as the other exploits are similar to this one. Listing 16 shows the first step of this exploit where the malicious XSS payload is written to the database without sufficient sanitization for `$titolo` variable. The second step occurred in `leggi.php` where all messages stored in the `geccBB_forum` table are retrieved and rendered to users without sanitization as in Listing 17. This exploit has a navigation length of 2 (`rispondi.php` and `leggi.php`). The number of generated constraints is 40 with exploit generation time of 13 sec. The corresponding exploit string is shown in Listing 24 in the Appendix.

```
$titolo=$_POST[titolo];...
$query_ins_risposta="INSERT INTO geccBB_forum VALUES ('',
     '$postatoda','$data','$titolo','$testonuovo','$rispostadel')";
$r=mysql_query($query_ins_risposta);
```

Listing 16: Step 1 (insert) in `rispondi.php`.

```
$rd=$_GET[rd];
if($rd)
  $query_caga_messaggio="SELECT * FROM geccBB_forum WHERE id=$rd";
  ...
  echo "re: " . $leggi[titolo];
```

Listing 17: Step 2 (select and then echo) in `leggi.php`.

On **Schoolmate**, our tool generated 2 stored XSS exploits. Listings 18 and 19 show one of these exploits. In the first step, the attacker can execute the update query without any authentication or authorization. The application receives the attacker's input and stores it in `schoolinfo` table in which two columns `schoolname` and `sitetext` are of type *varchar* (i.e., allows malicious characters if no sanitization applied in the source code). In the second step, the malicious payload is executed when any user of the site visits `login.php`. This exploit has a navigation length of 2 (`header.php` and `login.php`). The number of generated constraints is 150 with exploit generation time of 98 sec. The exploit string is shown in Listing 25 in the Appendix.

```
 $query = mysql_query("select schoolname from schoolinfo")
      or die("Unable to retrieve school name: " . mysql_error());
 $schoolname = mysql_result($query,0);
 if($_POST["infoupdate"] == 1)
   $query = mysql_query("UPDATE schoolinfo SET schoolname=
        ".htmlspecialchars($_POST["schoolname"]).",address=
        '$_POST[schooladdress]', phonenumber=
        '$_POST[schoolphone]',sitetext='$_POST[sitetext]',
        sitemessage= '$_POST[sitemessage]', numsemesters=
        '$_POST[numsemesters]', numperiods= '$_POST[numperiods]',
        apoint= '$_POST[apoint]',bpoint= '$_POST[bpoint]',cpoint=
        '$_POST[cpoint]',dpoint= '$_POST[dpoint]',fpoint=
        '$_POST[fpoint]' where schoolname = '$schoolname' LIMIT 1");
```

Listing 18: Step 1 (update) in `header.php`.

```
 $query = mysql_query("select sitetext from schoolinfo");
```

```
$text = mysql_result($query,0);
print("... <div class='messagebox'>$text </div> ...);
```

Listing 19: Step 2 (select and print) in `login.php`.

5.4 Effect of Database Schema Analysis

Table 3 shows the contribution of the database schema analysis we implemented in Chainsaw. Overall, Chainsaw reduced the false positive rate on exploit seed generation by an average of 16% per application on 8 of the 9 applications in our dataset. For `WeBid`, Chainsaw's DB-schema analysis reduced the false positive rate on SQLI seed generation by nearly 60%. Note that for `Scarf` the entries are zero because we have not found SQLI seeds for it.

App.	Without Schema	With Schema	Confirmed
myBloggie	24	24	21
scarf beta	0	0	0
DNScript	4	4	1
WeBid	99	40	40
Eve	6	5	5
schoolmate	60	51	49
geccbblite	3	3	3
faqforge	4	4	4
webchess	28	15	15

Table 3: Effect of DB schema analysis on SQLI seed exploits

5.5 Overhead and Analysis Complexity

Table 4 summarizes the runtime overhead of Chainsaw for exploit generation. The analysis time ranges from 10 to 600 minutes. For small applications such as `geccbblite`, solving the final exploit formula took less than a minute while in big applications such as `WeBid` it took about 42 minutes. Generally, the increase in the number of possible execution paths in each module and number of modules in each application increases the analysis time. The maximum length of navigation sequences traversed by the generated exploits ranges from 2 to 5 modules. Note that for Webid, schoolmate, and webchess, the analysis time and solving time are exceptionally long because Chainsaw has to traverse 1.09 million, 1.93 million, and 1.5 million execution paths respectively. For each execution path, our system has to compute the symbolic formula and invoke the solver —thus contributing to the analysis and solving overheads.

Application	Analysis Time (m)	Solving Time (m)	Expl. Gen. Time (m)	Max. Len. of Nav. Seq.
myBloggie	125	95	20	3
scarf	10	39	2	2
DNScript	20	120	4	2
WeBid	600	345	42	5
Eve	19	5	3	2
schoolmate	400	159	23	4
geccbblite	10	10	<1	2
faqforge	20	8	3	2
webchess	136	120	14	3

Table 4: Runtime summary of Chainsaw. **Analysis Time**: total time before invoking solver. **Solving Time**: total time to solve all SMT formulas (including unsatisfiable ones). **Expl. Gen. Time**: total time for solving satisfiable SMT formulas that represent exploits. **Max. Len. of Nav. Seq.**: maximum length of navigation sequences for exploits.

5.6 Comparison with Related Work

Note that one-to-one comparison of Chainsaw with prior work is not easy due to (*i*) absence of working tools for some prior work (*ii*) outdated test subjects and (*iii*) lack of clarity on how results are counted. Nevertheless, wherever we could find a common ground (e.g., common subject applications, same attack types), we compare Chainsaw with prior work.

Regarding seed generation, we compare seeds generated by Chainsaw with vulnerabilities reported in [8], [9], and [26]. On the exploit generation front, we compare Chainsaw with [19] and [16] since both of their datasets are subsumed by the test subjects in Chainsaw. Tables 5 and 6 show the summary of Chainsaw's comparison with these works. Note that different from some of these works, we count the number of seeds based on the number of unique sinks not per path. For instance, for `n` vulnerable paths leading to a sink, Chainsaw reports only one seed for this sink.

Seed generation: Wassermann and Su [26] identified 5 SQLI vulnerabilities in `EVE` whereas Chainsaw generated 7 SQLI exploit seeds, all of which have been transformed into working exploits. RIPS [8] identified 8 SQLI vulnerabilities in `myBloggie` while Chainsaw generated 24 SQLI exploit seeds (3 times more exploits than RIPS) since it performs path-sensitive analysis, which in RIPS is not available.

Exploit generation: Ardilla [19] generated 60 attacks (23 SQLI and 37 XSS) on a dataset of 5 applications (see Table 5). On the same dataset, Chainsaw generated 116 (56 more) working exploits, which shows that Chainsaw has better coverage. For XSS exploits, Chainsaw generated 36 working exploits while Ardilla generated 37. On SQLI exploit generation, Chainsaw outperforms Ardilla by 47 more as it generated 80 exploits compared to 33 in Ardilla. The reason behind 1 less XSS exploit in Chainsaw with respect to Ardilla is traced to one case in the `schoolmate`.

For the code snippet in Listing 20 from `schoolmate`, Ardilla generated an exploit for the `print` sink whereas Chainsaw flagged it as unfeasible because of the presence of the query at line 1. This query needs to return some records that satisfy the *where* clause or the execution of the module is terminated and never reaches the `print` statement. However, Ardilla's input for `$_POST[selectclass]` will not make the query return any records because `courseid` column is integer and will not accept the supplied string input.

```
1 $query = mysql_query("SELECT aperc,bperc,cperc,dperc,coursename
      FROM courses WHERE courseid = $_POST[selectclass]") or
2 die("ClassInfo.php: Unable to get the class info-".mysql_error());..
3 print(..<input... name='selectclass' value='$_POST[selectclass]'/>);
```

Listing 20: Unfeasible exploit generated by Ardilla in `ClassSettings.php, schoolmate`

On `myBloggie`, Chainsaw generated 7 exploits while [9] generated 5 second-order vulnerabilities. On `Scarf`, both [9] and Chainsaw generated 1 vulnerability and working exploit respectively. CraxWeb [16] reports a total of 54 exploits (24 SQLI and 30 XSS) on 4 applications (`Schoolmate`, `Webchess`, `faqforge`, and `EVE`). Over this set of applications, Chainsaw generated 106 exploits (74 SQLI and 32 XSS). Chainsaw generates 3-fold SQLI exploits compared to CraxWeb, and 32 XSS exploits versus CraxWeb's 30.

Application	Ardilla [19]	CraxWeb [16]	Chainsaw
Eve	6	5	9
schoolmate	18	31	56
webchess	25	11	31
faqforge	5	7	10
geccbblite	6	-	10

Table 5: Comparison on working exploits generation (numbers reported are the sum of SQLI+XSS).

Application	[26]	[8]	[9]	Chainsaw
myBloggie	-	8	5	25
scarf	-	-	1	1
Eve	5	-	-	7

Table 6: Comparison on exploit seeds generation.

5.7 Discussion

Static analysis limitations: Our tool can handle dynamic workflow constructs where arguments are variables defined along an execution path. However, if the application code contains dynamically generated workflow features (e.g., hyperlinks generated using JavaScript), Chainsaw cannot resolve the dynamic arguments of such functions. As a result, when constructing the GWFG, Chainsaw may not infer the existence of such dynamic edges. Thus, it may miss some navigation links between modules. In our dataset, however, we missed at most 0.23% of edges in all GWFGs. Additionally, Chainsaw cannot construct symbolic sink expressions if the *structure* (i.e., code, not arguments) of a SQL query or an echo-like statement cannot be constructed statically (e.g., determined by user input). A detailed discussion on symbolic sink construction and its limitations can be found in our previous work [6].

Solver failures: In few instances, Chainsaw encountered a timeout or returned unknown (i.e., could not determine if formula was satisfiable). In our evaluation, the solver encountered undecidability in solving 28 out of 3463 seed SMT files (less than 1% of the overall seed SMT files). Additionally, the solver timed-out during the solving of 45 SMT files (about 1% of the overall seed SMT files).

Unsupported PHP features: Chainsaw uses Pixy [18] for control flow analysis in PHP web applications. Some PHP features such as dynamic inclusions and certain object-oriented features are not supported by Pixy. Consequently, Chainsaw does not handle such features. In our evaluation, however, these have not limited the applicability of Chainsaw to the elaborate dataset that is used. Note, even though Chainsaw is implemented for PHP (which is widely used), our approach is applicable to other web platforms.

6. RELATED WORK

Exploit generation for binary programs. Brumley et al. [7] developed an automatic exploit generation technique that identifies the difference between patched binaries and their unpatched versions to generate inputs that trigger the difference. Avgerinos et al. [1] developed AEG to generate control flow hijacking exploits on binary programs. FlowStich [15] generated data-oriented attacks by connecting data-flows in binary programs assuming that control flow integrity is intact. All these works focus on generating exploits for binaries while our approach deals with global navigation structure and web applications state.

Exploit generation for web applications. Ardilla [19] uses concolic execution and taint tracking (including database tracking) to construct SQLI and XSS attack vectors. Ardilla has three main limitations: low code coverage (<50%), simple attack library, and imprecise taint tracking via built-in functions. Chainsaw outperformed Ardilla in the number and the complexity of the generated exploits (second-order SQLI and deeply located exploits) due to the use of static analysis, search strategies and constraint solving to analyze application navigation state and to model built-in sanitization functions. CraxWeb [16] employs concrete and symbolic execution supported by a constraint solver to generate SQLI and XSS exploits. CraxWeb has lower code coverage than Chainsaw and does not generate second-order exploits which are harder to construct. Even though CraxWeb is platform independent, the techniques implemented in Chainsaw can be extended and applied to non-PHP web applications. QED [21] generates first-order SQLI and XSS attacks using static analysis and model checking for Java web applications. In contrast to Chainsaw, QED does not consider second-order exploits and it requires writing attack specifications in a specific language. [27] generates inputs that expose SQLI vulnerabilities using concolic execution of PHP applications. It finds only first-order SQLI bugs, requires source code instrumentation, and does not perform database analysis. EKHunter [12] combines static analysis and constraint solving to find exploits in for-crime web applications. The generated exploits do not span several HTTP requests, unlike Chainsaw. WAPTEC [5] generates exploits for parameter-tampering vulnerabilities, which are logical vulnerabilities as opposed to injection vulnerabilities handled by Chainsaw.

Vulnerability analysis. There is a large body of research that studied injection vulnerabilities detection. Broadly, there are static analysis related works (such as [17], [18], [26], [20] and [29]), dynamic analysis approaches (e.g., [22], [13]), and hybrid approaches (such as [2]). Other approaches focused on finding and fixing sanitation related errors, the main cause of injection vulnerabilities, as in [23] and [24].

MiMoSA [3] is a system that reasons across modules of a web application to find data and workflow attacks. Our navigation-aware analysis is inspired by MiMoSA but it advances the analysis to a systematic workflow exploration to build working exploits using key features such as *GWFG*, *RWFG*, *preconditions-summaries*, and *summary histories*. Sun et al. [25] propose a technique to find access control vulnerabilities in web applications. Their approach builds a graph similar to the GWFG, however, the purpose of that graph is different from the one generated by Chainsaw. [26] is an approach that identifies SQLI vulnerabilities by modeling database queries using context free grammars. Chainsaw is more precise as it factors in database schema analysis and a constraint solver to generate concrete exploitation for SQLI and XSS. In RIPS [8], a taint analysis system is proposed to detect injection vulnerabilities. [9] develops a static analysis technique that detects second-order vulnerabilities that cross persistent storage. Even though RIPS and [9] can detect vulnerabilities other than SQLI and XSS, in contrast to Chainsaw, they implement a path insensitive analysis which yields to high false positive rate and they do not generate navigation-aware working exploits.

7. CONCLUSIONS

In this paper, we present Chainsaw, a system that reasons systematically over the navigation structure and uses the database state of web applications to automatically generate working exploits. Chainsaw generated a total of 199 working exploits on our dataset, of which 30 (15%) are second-order exploits with no false positives. Finally, we also demonstrated that Chainsaw significantly outperforms prior work on vulnerability detection and exploit generation.

Acknowledgments

We thank Bridget Basan, Antonio Shelton and Kyle Tulipano for their help in modeling PHP functions. We also thank our shepherd Adam Doupé and the anonymous reviewers for their feedback. This material is supported in part by NSF under Grant Nos. CNS-1514472, DGE-1069311 and by DARPA under an AFOSR contract.

8. REFERENCES

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *NDSS*, volume 11, pages 59–66, 2011.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401, 2008.

[3] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module Vulnerability Analysis of Web-based Applications. In *the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 25–35, 2007.

[4] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 332–345, 2010.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *the 18th ACM conference on Computer and communications security*, pages 575–586, 2011.

[6] P. Bisht, A. P. Sistla, and V. Venkatakrishnan. Automatically preparing safe SQL queries. In *International Conference on Financial Cryptography and Data Security*, pages 272–288. Springer, 2010.

[7] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[8] J. Dahse and T. Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[9] J. Dahse and T. Holz. Static Detection of Second-Order Vulnerabilities in Web Applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 989–1003, 2014.

[10] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] D. Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, Feb. 1999.

[12] B. Eshete, A. Alhuzali, M. Monshizadeh, P. A. Porras, V. N. Venkatakrishnan, and V. Yegneswaran. EKHunter: A Counter-Offensive Toolkit for Exploit Kit Infiltration. In *22nd Annual Network and Distributed System Security Symposium, NDSS*, 2015.

[13] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *21st Annual Computer Security Applications Conference (ACSAC)*, pages 9–pp, 2005.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, 1988.

[15] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192. USENIX Association, 2015.

[16] S. Huang, H. Lu, W. Leong, and H. Liu. CRAXweb: Automatic Web Application Testing and Attack Generation. In *IEEE 7th International Conference on Software Security and Reliability, SERE*, pages 208–217, 2013.

[17] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web (WWW)*, pages 40–52, 2004.

[18] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis tool for Detecting Web Application Vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp, 2006.

[19] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic Creation of SQL Injection and Cross-Site Scripting Attacks. In *IEEE 31st International Conference on Software Engineering (ICSE)*, pages 199–209, 2009.

[20] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *14th USENIX Security Symposium*, Baltimore, Maryland, USA, 2005.

[21] M. Martin and M. S. Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium*, pages 31–43, 2008.

[22] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*, pages 295–307. Springer, 2005.

[23] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 587–600, 2011.

[24] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 601–614, 2011.

[25] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *USENIX Security Symposium*, 2011.

[26] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices*, volume 42, pages 32–41. ACM, 2007.

[27] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260, 2008.

[28] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security*, volume 6, pages 179–192, 2006.

[29] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 154–157, 2010.

[30] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124, 2013.

Appendix

```
http://host/chess/index.php?txtNick=user&pwdPassword=pass&remember=yes
http://host/chess/mainmenu.php?messageID=
http://host/chess/viewmessage.php?messageID=1 OR 1=1
```

Listing 21: Exploit string for the query at line 3 in Listing 8.

```
http://host/eveactive/add.php?name=user&comment=hiThere&id=2&division=1
    &rankcorp=1&vacation=0&deleted=0&action=add
http://host/eveactive/member.php?nname=<script>alert("error")</script>
```

Listing 22: Exploit for the query in Listing 13.

```
To insert:http://host/mybloggie/login.php?mode=login&username=user
    &passwd=pass
http://host/mybloggie/admin.php?mode=adduser&username=foo’ OR ’’=’
    &repassword=aaaa&password=aaaa&level=1&submit=yes
To select:http://host/mybloggie/login.php?mode=login&username=user
    &passwd=pass
http://host/mybloggie/admin.php?mode=edituser&id=1&pass=no
```

Listing 23: second-order SQLI exploit for Listings 14 and 15.

```
http://host/geccBBlite/ripondi.php?titolo=<script>alert("error")</scr
    ipt>&postatoda=aaaa&testo=aaaa&testonuovo=&ispostadel=0
http://host/geccBBlite/leggi.php?rd=1
```

Listing 24: Stored XSS exploit for Listing 17.

```
http://host/schoolmate/header.php?schoolname=name&schooladdress=23aaa&
    schoolphone=1234&sitetext=<IFRAME SRC=http://site.html>&
    sitemessage=hello&numperiods=2&numsemesters=2&numperiods=2&apoint
    =90.0&bpoint=80.0&cpoint=70.0&dpoint=60.0&fpoint=40.0&infoupdate=1
http://host/schoolmate/login.php?
```

Listing 25: Stored XSS exploit for Listing 19.