NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications

Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete,
and V.N. Venkatakrishnan, *University of Illinois at Chicago*

https://www.usenix.org/conference/usenixsecurity18/presentation/alhuzali

**This paper is included in the Proceedings of the
27th USENIX Security Symposium.**

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

**Open access to the Proceedings of the
27th USENIX Security Symposium
is sponsored by USENIX.**

NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications

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

Abstract

Modern multi-tier web applications are composed of several dynamic features, which make their vulnerability analysis challenging from a purely static analysis perspective. We describe an approach that overcomes the challenges posed by the dynamic nature of web applications. Our approach combines dynamic analysis that is guided by static analysis techniques in order to automatically identify vulnerabilities and build working exploits. Our approach is implemented and evaluated in NAVEX, a tool that can scale the process of automatic vulnerability analysis and exploit generation to large applications and to multiple classes of vulnerabilities. In our experiments, we were able to use NAVEX over a codebase of 3.2 million lines of PHP code, and construct 204 exploits in the code that was analyzed.

1 Introduction

Modern web applications are typically designed as multi-tier applications (i.e., client, server, and database). They include many dynamic features, which generate content ”on the fly” based on user interaction and other inputs. Such dynamism helps the usability as well as the responsiveness of the application to the user. These features, however, increase the complexity of web applications and raise the difficulty bar of analyzing their security.

很多研究如何检测现代Web应用的漏洞但是他们都存在大量误报

Currently, several approaches exist for analyzing the security of modern web applications such as [9, 15, 18, 29]. These approaches use a series of analysis techniques to identify vulnerabilities such as SQL Injection (SQLI) and Cross-Site Scripting (XSS). However, a drawback of these approaches is that they generate false alarms, therefore require manual efforts to check whether each one of the reported vulnerabilities is indeed exploitable.

其他研究更进一步，通过自动产生exp的方式来去误报，但是他们的研究都是静态的，仍

Other approaches take a further step and try to include methods for automatically verifying that vulnerabilities are true by generating concrete exploits [7, 25, 27, 32]. However, these approaches use largely static analysis methods. While static analysis methods can provide good coverage of an application, they often sacrifice precision due to technical challenges related to handling complex program artifacts, which is one of the main reasons for generating false positives. In particular, static analysis is challenging in the context of the dynamic features of web applications, where content (e.g., forms, links, JavaScript code) is often generated on the fly, and the code is executed at different tiers, whose effects are difficult to model statically.

我们通过动态执行配合静态的方式，动态能够减少复杂性，静态能够扩大代码覆盖范围

In this paper, our main contribution is a precise approach for vulnerability analysis of multi-tier web applications with dynamic features. Rather than following a strictly static analysis strategy, our approach combines dynamic analysis of web applications with static analysis to automatically identify vulnerabilities and generate concrete exploits as proof of those vulnerabilities. The combination of dynamic and static analysis provides several benefits. First, the dynamic execution component greatly reduces the complexity faced by the static analysis by revealing run-time artifacts, which do not need to be modeled statically. On the other hand, the static analysis component guides its dynamic counterpart in maximizing the coverage of the application by analyzing application paths and providing inputs to exercise those paths. Second, our approach scales to very large applications (e.g., 965K LOC), surpassing significantly the state of the art. The main reason for the increased scalability is the ability of the dynamic execution component to reduce the complexity faced by the static analysis component.

我们设计了几个分析模板以及一个攻击字典来实例化每一个模板

An additional goal of our approach is that of enabling automatic exploit generation for different classes of vulnerabilities with minimal analysis setup overhead. To achieve this goal, our approach was designed with several analysis templates and an attack dictionary that is used to instantiate each template. There exist other *static* approaches that try to achieve such generality for identifying vulnerabilities [9, 15]. However, our approach extends [9] by (a) applying precise dynamic analysis techniques and (b) automatically generating exploits for the identified vulnerabilities.

第一步，我们用符号执行构建各个模块的行为模型

Our approach is implemented in a tool called NAVEX. NAVEX's operations are divided into two steps. In the first step, we create a model of the behavior of individual modules of a web application using symbolic execution. To address the scalability challenge, we prioritize only those modules that contain potentially vulnerable sinks where an attacker 'may' be successful in injecting malicious values or in exploiting other types of vulnerabilities, and analyze them further in the successive search.

第二不，我们构建应用并且使用爬虫获取网站路径，同时使用动态符号执行最大化代码覆盖范围

In the second step, we construct the actual exploits. This requires modeling the whole application and discovering a sequence of HTTP requests that take an application to execute a vulnerable sink. To address the scalability challenge in this phase, we perform dynamic analysis of a deployed application and use a web crawler and a concolic executioner on the server-side to uncover possible HTTP navigation paths that may lead the attacker to the vulnerable sink. To maximize the coverage of the code during dynamic analysis, the crawler and concolic executioner are aided by a constraint solver, which generates the (exploit) sequence of HTTP inputs.

我们构造了一个可拓展的web漏洞生成框架

Our contributions in NAVEX include an exploit generation framework that can easily scale to large applications and many classes of vulnerabilities and a novel method that combines dynamic execution and static analysis to address scalability issues affecting previous works, mainly due to the dynamic features of web applications.

We evaluate NAVEX on 26 applications having a total of 3.2M SLOC and 22.7*K* PHP files. NAVEX was able to analyze the applications and generated 204 exploits, in little under 6.5 hours. Of these exploits, 195 are related to SQLI and XSS, while 9 are related to logic vulnerabilities, such as Execution After Redirect (EAR) vulnerabilities. We note that NAVEX is the first reported work in the literature to construct exploits for EAR vulnerabilities.

This paper is organized as follows. Section 2 discusses a running example to highlight challenges and provides an overview of NAVEX, Architectural and algorithmic details of NAVEX are discussed in Section 3. Section 4 contains details about the implementation, Section 5 describes the evaluation of NAVEX, and Section 6 discusses the related work. Finally, Section 7 contains the conclusions.

2 Challenges and Approach Overview

In this section, we use a running example to highlight the challenges addressed in this paper. We then present an overview of NAVEX.

2.1 Running Example

Listings 1-3 present a simple book borrowing web application, which will be used throughout this paper to illustrate our approach. Books can be selected through the web form in `selectBooks.php` module (lines 23-38 in Listing 1). `SelectBooks.php` validates some of the user input using JavaScript (lines 31-36). The user input is further validated and sanitized by server-side code (lines 4-12). Next, the module queries the database to check the book availability (line 17). Based on the query results, `$_SESSION['ISBN']` is initialized and an HTTP link to `hold.php` is printed on the browser.

```
1  <?php
2  if(!isset($_SESSION['username']))
3    header( "Location: index.php" );
4  if (isset($_POST['book_name']))
5   $book_name =
         mysql_real_escap_string($_POST['book_name']);
         //sanitization
6  else
7   $book_name ="";
8  if (isset($_POST['edition']))
9    $edition = (int)$_POST['edition']; //user input is
          sanitized
10 else
11   error();
12 if (isset($_POST['publisher']) &&
       strlen($_POST['publisher'])<=35)
13   $publisher = str_replace(""", "\"", $_POST['publisher']);
14 else
15   error();
16 $action = $_GET['action'])
17 $isbn= mysql_query( "SELECT isbn FROM BOOK_TABLE WHERE
       book_name='$book_name' AND edition = '$edition' AND
       publisher='$publisher'"); //vulnerable sink to SQLI
18 if (mysql_num_rows( $isbn ) == 1 ){
19     $_SESSION['ISBN'] = $isbn;
20     echo "<a href='".BASE_URL."hold.php'> Hold the
           Book</a>";
21     }
22 ?>//client-side code starts
23 <html><body><form method="post" action="<?php echo
       $_SERVER['PHP_SELF']."?action=borrow"?>"
       onsubmit="validate()">
24  <select name='book_name'> //drop-down list
25     <option value="Intro to CS by author1">Intro to
           CS</option>
26     <option value="Intro to Math by author2">Intro to
           Math</option>..
27   </select>
28   <input type='text' name='publisher'>
29   <input type='text' name='edition'>
30 </form>
31 <script type="text/javascript">
32 function validate() { //validates form upon submission
33   var edition = document.getElementsByName("edition");
34   if(edition.value <= 0)
35       return false; // do not submit the form
36   return true; //submit the form
37  }
38 </script></body></html>
```

Listing 1: `selectBooks.php`, find books to borrow.

`Hold.php` (Listing 2) performs additional checks and, if they are satisfied, an HTTP link guides the user to the next step (line 7). When the link is clicked the superglobal $_GET['step'] is set and the module `checkout.php` is therefore included by `hold.php` and executed. `Checkout.php` completes the borrowing process by providing a link (line 19) to the user for confirmation. The link sets two superglobals (`$_GET['step']` and `$_GET['msg']`), which will be checked by the module (line 6). Finally, a confirmation function (line 13) is called to notify the user that the book was successfully reserved.

```
<?php
if(!isset($_SESSION['username'])) {
  header( "Location: index.php" );
  exit();
  }
if (isset($_SESSION['ISBN'])){
  echo "<a href='".BASE_URL."hold.php?step=checkout'>
       Checkout</a>";
  if (isset($_GET['step']) && $_GET['step'] == "checkout")
     include_once( "checkout.php");
  }
?>
```

Listing 2: `hold.php`, hold books for pickup.

```
<?php
if(!isset($_SESSION['username'])) {
 header( "Location: index.php" );
 exit();
 }
if (isset($_GET['msg']) && isset($_SESSION['ISBN'])){
 $sql = "SELECT name FROM USERS WHERE
      username='$_SESSION['username']'" ;
 $result = mysql_query($sql);
 $name = $db->sql_fetchrow($result);
 $msg = $_GET['msg'];
 confirm($name, $msg);
 }
function confirm($name, $msg){
 if (isset($name) && isset($msg) )
   echo $name. " you are ".$msg; // XSS vulnerability
 }
?> //client-side code starts
<html><body>
<a href="hold.php?step=checkout&msg=done">DONE</a>
</body></html>
```

Listing 3: `checkout.php`, checkout functionality.

The example contains sensitive sinks that are vulnerable to injection and logic attacks. For example, the query in listing 1 (line 17) is vulnerable to SQLI through the variable `$publisher`, which is not properly sanitized before reaching the sink. In particular, the `str_replace` function (line 13) does a poor job of sanitizing `$publisher`, since an SQLI attack not involving double quotes may still be used. Additionally, the `echo` call in Listing 3 is vulnerable to XSS as the user input `$msg` is not sanitized. Finally, the sink at Listing 1 line 3 is vulnerable to an Execution After Redirect (EAR) logic attack because the execution after the `header` call (redirects the execution to another PHP module) does not halt since there is no call to an execution termination function afterward. Consequently, the following statements will be executed regardless of the check at line 2. The problem is further exacerbated by the fact that those statements contain a vulnerable SQL query. An attacker may thus be able to run a SQLI exploit without needing to log in first.

2.2 Challenges

As illustrated by the example, typical web applications have client-side logic that consists of forms, links, and JavaScript code, which may be dynamically generated by the server-side code, as well as a complex server-side logic that frequently interacts with the client-side and with the database backend. Therefore, building an exploit generation framework that uncovers a wide range of different types of exploits for dynamic web applications is non-trivial. Specifically, we identify the following challenges:

Sink reachability. In web applications, some tasks/-functionalities require a series of steps, and there are dependencies that exist between these tasks. These steps are usually accomplished using different modules where the state of the application, maintained through the use of global constructs (e.g., $_GET[] in PHP), is updated to reflect the completion/failure of a step. If a sensitive sink is located deep in these interrelated modules, the challenge is to automatically generate an exploit that navigates through the complex dependencies among application modules while satisfying constraints required at each junction in the navigation. For instance, a successful exploit for the vulnerable `echo` in Listing 3, must consider navigation and constraint satisfaction through the modules `selectBooks.php`, `hold.php`, `index.php` (not shown in the example), and `checkout.php`.

More broadly, we must take into account several factors. First, data flow paths from sources to sensitive sinks must be identified. Next, possible data sanitizations along those paths must be analyzed. However, sanitizations are available in many flavors, including built-in sanitizations (e.g., `htmlspecialchars()`), implicit sanitizations (e.g., cast operators as shown in the running example), custom sanitizations (e.g., custom use of `str_replace()`), and sanitizations induced by database constraints (e.g., `NOT NULL` constraints). The practical challenge here is to precisely identify when such sanitizations are sufficiently robust to eliminate all possible risks.

Dynamic features. An automatic exploit generation approach that is entirely based on static aspects of a web application is prone to miss certain real exploits. As mentioned before, modern web applications often contain features that are revealed only when the application is executed. These features often include dynamically generated forms and links that may drive the navigation of the application to vulnerable sinks. Unless the application is deployed and executed, it is challenging for a static analysis approach to infer such artifacts, which may contain useful constraints for exploit paths. For instance, line 23 of Listing 1, where the `action` of the form is set by the result of running the embedded PHP code. To precisely infer the value of that action, a static analyzer has to be able to handle the PHP semantics of that code portion. Other situations (not shown in the example) include dynamically generated content including JavaScript generated content. It is, therefore, necessary to incorporate dynamic analysis as part of the exploit generation framework to make these runtime artifacts explicit. An additional challenge with dynamic execution is maximizing the coverage of an application.

Scalability. Generating executable exploits that span multiple modules and traverse execution paths inside each module for large and complex modern web applications is challenging. Constructing exploits requires analyzing the application as a whole, including its client-side, server-side and database backend. To deal with this challenge, the exploit generation approach must be designed with careful considerations for pruning unfeasible exploit paths. To demonstrate the need for a scalable approach, let's consider our running example. For this simple application, to construct an exploit for the vulnerable sink in Listing 3, we have to process a total of 44 execution paths in the 3 modules (i.e., 32 paths in `selectBooks.php`, 4 in `hold.php`, and 8 in `checkout.php`) to find candidate exploitable paths to the sink.

Another scalability challenge we need to tackle is related to the goal of generating exploits for multiple classes of vulnerabilities. To address this challenge, we need to support abstraction and analysis of multiple classes of vulnerabilities efficiently, as to generate as many different types of exploits as possible.

2.3 Approach Overview

Our goal is to build a precise, scalable, and efficient exploit generation framework that takes into account the dynamic features of web applications and the navigational complexities that stem from dependencies among the client-side, server-side and database backend.

Our approach is implemented in a system called NAVEX, as shown in Figure 1. To address the *scalability* challenges, our approach is divided into two steps: (I) vulnerable sink identification and (II) concrete exploit generation.

Given the application source code, the first step identifies vulnerable sinks in the application and the corresponding modules. This phase analyzes each module separately and is crucial for prioritizing only those modules that have vulnerabilities; thus significantly reducing the search space and contributing to *scalability*. To address the *sink reachability challenge*, NAVEX builds a precise representation of the semantics of built-in sanitization routines. In addition, for custom sanitizations, it builds a model using symbolic constraints. These constraints are used by a constraint solver, which determines if the sanitizations are sufficiently robust.

The second step is responsible for generating concrete exploits. The main problem in automatically generating concrete exploits is that of identifying application-wide navigation paths that, starting from public-facing pages, drive the execution to the vulnerable sinks identified in

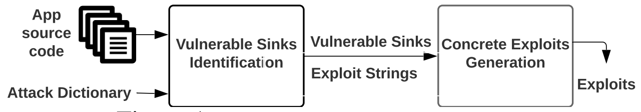

Figure 1: The architecture of NAVEX.

the first step through a series of HTTP requests. The output of the dynamic execution is a *Navigation Graph* that represents the navigation structure of the web application. Finally, for every module containing a vulnerable sink, as identified in the first step, NAVEX uses this navigation graph to find the paths from public modules to that module along which the exploit can be executed. The dynamic features challenge is addressed in NAVEX by combining dynamic analysis and symbolic execution of applications. To maximize the coverage of an application, NAVEX repeats the dynamic execution many times, each time with different inputs generated by a constraint solver in a way that maximizes path coverage in the application. At each execution, NAVEX collects the information necessary to derive the application's navigation structure.

3 Architecture and Algorithms

3.1 Vulnerable Sink Identification

To identify the vulnerable sinks, NAVEX analyses each module separately. An implicit goal of this step is to exclude from the following step those modules that do not contain vulnerable sinks. In particular, as depicted in Figure 2, NAVEX first builds a graph model of each module's code, then it discovers the paths that contain data flows between sources and sinks. Finally, it uses symbolic execution to generate a model of the execution as a formula and constraint solving to determine which of those paths are potentially exploitable. Each of these components is described next.

3.1.1 Attack Dictionary

To address the challenge of *discovering multiple classes of vulnerabilities*, NAVEX was designed to be easily extensible to a wide range of vulnerabilities, such as SQLI, XSS as well as logic vulnerabilities such as EAR [18] and command injection. A key observation is that several types of vulnerabilities are essentially similar. For instance, SQLI and XSS both depend on the flow of malicious data from sources to sinks and injection of malicious data in those sinks. The main difference is the nature of the sink and the attack payload. This similarity, in turn, can be leveraged to build analysis templates that can be instantiated with minimal changes to discover different classes of vulnerabilities. To this end, NAVEX builds an *Attack Dictionary*, which is used to instantiate analysis templates targeting each class of vulnerability. In particular, it contains attack specifications, as follows:

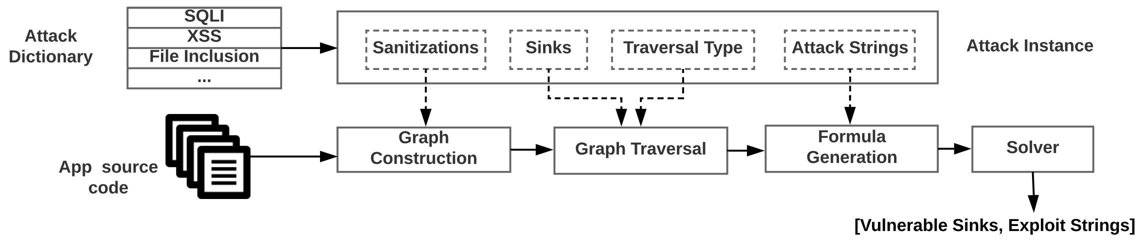

Figure 2: Vulnerable Sinks Identification (Step I) Components.

Sinks. These are instructions that execute the malicious content of an attack. For instance, `echo` and `print` PHP functions are sinks for XSS attacks.

Sanitizations. These include an extensive list of PHP sanitizations, including built-in sanitization functions and operators, which may implicitly sanitize an input (e.g., cast operators). While extensive, this list is not exhaustive, and therefore it may miss functions. However, the semantics of known custom sanitization functions (e.g., `str_replace`) are captured by NAVEX using constraint solving.

Traversal Type. It specifies the type of traversal that is needed on the graph (the graph representation will be described shortly). We currently support forward and backward traversals between sources and sinks. Specifically, injection vulnerabilities typically need a backward traversal, while vulnerabilities such as EAR need a forward one.

Attack Strings. The attack strings are specifications of the possible (malicious) values that can appear at a sink. While not exhaustive, the list of attack strings used by NAVEX is very extensive. It contains 45 attack patterns collected from cheat sheets (e.g., [6]), and security reports.

Currently, the attack dictionary contains entries for SQLI, XSS, file inclusion, command injection, code execution, and EAR.

3.1.2 Graph Construction

This step builds a graph model to represent the possible execution paths, which are later symbolically executed, in a PHP module. Specifically, our graph model is based on Code Property Graphs (CPGs) [9, 33], which combine abstract syntax trees (AST), control flow graphs (CFG), call graph, and data dependence graphs (DDG) under a unique representation to discover vulnerabilities, which are modeled as graph queries. In particular, given a source and a sink instruction, CPGs can be used to find data dependency paths between their variables.

However, our final goal is not merely that of finding vulnerable paths but also that of generating *concrete exploits*. To this end, we extend CPGs with *sanitization* and *database constraint* tags. These tags are attributes added to the CPGs and are used to prune out a large number of potentially unexploitable paths and indirectly addressing the challenge of path explosion.

Sanitization Tags. A sanitization tag stores information about the sanitization status of each variable in a node, if any. The possible values of the tag are `unsan-X`, `san-X` where X represents the specific vulnerability. For instance, `san-sql` and `unsan-sql` represent presence (or non-presence) of SQLI sanitization, respectively.

The values of the sanitization tags are inferred and added to the graph during its construction. In particular, as a node is added to the CPG, the corresponding node's AST is analyzed to detect eventual sanitizations. This analysis is guided by the *sanitizations* patterns contained in the attack dictionary for each type of vulnerability. When a match among the sanitization patterns is found for a variable in a node, the corresponding `san-X` value is set for that variable. Note, we add sanitization tags that resolve the sanitization status of different types of PHP statements such as assignment, cast, binary, unary statements, built-in functions, etc.

To demonstrate how NAVEX assigns sanitization tags, let us consider the statement at line 9 in Listing 1. NAVEX starts by inspecting the AST of `$edition = (int)$_POST['edition']` to assign an appropriate tag to $_POST['edition'] first. Then, it propagates the sanitization status to `$edition`. In this case, the assigned tag to $_POST['edition'] is `san-all` because the cast to integer operator sanitizes it for all vulnerabilities in our attack dictionary. Consequently, the variable `$edition` will have the same value in its sanitization tag.

Database Constraint Tags. Databases may often enforce additional constraints on the data that flow to the database tables. For instance, the columns of a database table may implicitly sanitize certain inputs, based on the column's data type (e.g., `enum` or `integer`). We enhance code property graphs to capture database constraints. In particular, for each web application, NAVEX parses its schema to collect table names, their columns names, data types, and value constraints (e.g., NOT NULL).

During the CPG construction, NAVEX adds a tag called *DB* to the root node of each application. This tag contains the collected information from the schema, and it is utilized later during the graph traversal and exploit generation (Sections 3.1.3 and 3.1.4).

3.1.3 Graph Traversal

The goal of this step is to discover vulnerable paths from sources to sensitive sinks by inspecting the enhanced CPG.

Backward Traversal. An example of a *backward* traversal for discovering vulnerable paths for injection vulnerabilities is shown in Algorithm 1.

The algorithm starts by searching the graph for calls to sensitive sinks specified in the attack dictionary (line 4). For each node representing a sink, it follows backward the data dependency edges for all variables used in that sink using the function `AnalyzeNode` (line 8). This function calls `FollowBackwardDDEdge` (line 18) to find all data dependency paths from a sink node to either a source or a function argument (if the sink is inside a function). If a path ends at a function argument, `AnalyzeNode` is called recursively over the nodes representing the call sites of that function (line 15). The function `FollowBackwardDDEdge` identifies intra-procedural paths between sources and sinks and uses the *sanitization* and *DB* tags to eliminate sanitized paths. Finally, `getPathsTo` (line 24) finds all traversed and unsanitized paths in the graph leading to source nodes.

As an example, consider the vulnerable sink `echo` to XSS (line 15) in Listing 3. Starting from this sink, the algorithm follows all data dependency edges backwards while checking the sanitization tags of `$name` and `$msg`. Since they are both unsanitized, NAVEX stores the intra-procedural paths of the variables and follows the data dependency edges in the caller function until it reaches the source of `$msg` (line 10). Note, `$name` is not a user input (holds values from the database) and therefore the algorithm only returns the inter-paths of `$msg` as vulnerable paths to XSS.

The `FilterSanNodes` function uses the sanitization and DB tags to prune out unpromising paths for exploit generation. In particular, DB tags are utilized during the search for SQLI vulnerability. For each write query, NAVEX parses the query using a SQL parser to find necessary information such as table and columns names. Then, it matches the extracted information with the DB tag to derive constraints from the columns data types and value constraints (F_{db}). These constraints are used in conjunction with the path constraints (F_{path}) in the next step (Section 3.1.4).

Forward Traversal. As another example, to detect EAR vulnerabilities, NAVEX performs a *forward* graph traversal from *sources* to *sinks* where the sources are redirection instructions (e.g., `header`) and the sinks are termination instructions (e.g., `die`). In particular, we distinguish between two types of EAR vulnerabilities, namely *benign* where the code between sources and sinks does not contain sensitive operations (e.g., SQL queries) and *malicious* EAR where that code contains them [18].

Algorithm 1 Injection Vulnerability Path Discovery

```
1:  Input: $sources, sinks$
2:  output: $VulnerablePaths$
3:
4:  $sinkNodes = \text{FINDSINKNODE}(sinks)$
5:  for all $sn \in sinkNodes$ do
6:      $VulnerablePaths = \text{ANALYZENODE}(sn)$
7:  return $VulnerablePaths$
8:  function ANALYZENODE(node)
9:      $VulnerablePaths \leftarrow []$
10:     $paths = \text{FOLLOWBACKWARDDDEDGE}(sn)$
11:     for all $path \in paths$ do
12:         if path has a source then
13:             $VulnerablePaths \leftarrow path$
14:         else
15:             $callPaths = \text{ANALYZENODE}(callNode)$
16:             $VulnerablePaths \leftarrow path + callPaths$
17:     return $VulnerablePaths$
18: function FOLLOWBACKWARDDDEDGE(node)
19:     $Intra\_Paths \leftarrow []$
20:     while $node$ is not a source $\wedge$ $node$ is not a func. argument do
21:         $IncNodes = \text{GETINCOMINGDDNODE}(node)$
22:         $UnsanNodes = \text{FILTERSANNODES}(IncNodes)$
23:         $node \leftarrow unsanNodes$
24:     $Intra\_Paths = \text{GETPATHSTO}(node)$
25:     return $Intra\_Paths$
```

The output of this step is a set of paths that are potentially vulnerable. This set of paths is sent in input to the next step.

3.1.4 Exploit String Generation

The last step of the static analysis is the generation of exploit strings over the vulnerable paths discovered during graph traversal. In this step, each vulnerable path is modeled as a logical formula F_{path}. In addition, the constraints derived from the *DB* tags F_{db} are added to the formula. It is next augmented with additional constraints over the variables at the sinks F_{attack}, which represent values that can lead to an attack. These values are retrieved from the *Attack Dictionary* based on the type of vulnerability under consideration.

The augmented formula (i.e., $F_{path} \wedge F_{db} \wedge F_{attack}$) is next sent to a solver, which provides a solution (if it exists) over the values of the input variables, that is an *exploit string*. This solution contains the values of the input variables, which, after the path and sanitizations executions, cause the attack string to appear at the sink. However, even if a solution exists, the related exploit is not necessarily feasible. To determine its feasibility, NAVEX needs to uncover the sequence of HTTP requests that must be sent to the application to execute the attack described by the exploit strings. This step is exposed in the rest of this section.

3.2 Concrete Exploit Generation

To generate the concrete exploits, NAVEX executes several steps as depicted in Figure 3. First, a *dynamic execution* step creates a navigation graph that captures the possible sequences in which application modules can be executed. Next, the *navigation graph* is used to discover execution paths to only those modules that contain the vulnerable sinks uncovered by the vulnerable sink identification step. Finally, the final exploits are generated. We describe each of these steps next.

3.2.1 Dynamic Execution

This step is responsible for building an application-wide navigation graph, which represents possible sequences of module executions together with associated constraints.

Previous research [7] has recognized the importance of building such a graph. However, a key difference with that work is the approach in which the graph is generated. In particular, the approach of [7] uses static analysis to discover links and forms and does not deal with the dynamic features of web applications, whose semantics are challenging to be captured statically.

In contrast, NAVEX uses a dynamic execution approach. It executes the web application through a crawler so that a significant portion of those dynamic features become concrete and do not need to be symbolically evaluated. However, a common challenge when performing the dynamic analysis is maximizing the coverage of the application. To address this challenge, NAVEX uses constraint solving and concolic execution to generate a large number of form inputs that aid the crawler in maximizing the coverage of the application.

Crawler. The crawler is responsible for uncovering the navigation structure of the applications. For each application, the crawler is initiated with a *seed URL* and whenever necessary, valid login credentials. While most applications have two types of roles (administrator and regular user), to maximize the crawling coverage, the crawler does the authentication for each role-type in the application. Starting from the *seed URL*, the crawler extracts HTML links, forms, and JavaScript code. The links are stored and used as the next URLs to crawl. For form submissions, the crawler needs to construct values that comply with the form restrictions (e.g., length of input) and satisfy eventual JavaScript validations. Having a mechanism that automatically generates valid form inputs greatly improves the crawling coverage of web applications since web forms are common constructs that influence the navigation structure.

To address this problem, our crawler extracts the forms' input fields, buttons, and action and method attributes (i.e., GET or POST) using an HTML parser and generates a set of constraints over the form values implied by the form attributes. In addition, to deal with JavaScript code that validates form inputs, the crawler leverages the techniques used in [12]. Specifically, the JavaScript code is extracted and analyzed using concrete-symbolic execution. The code is first executed concretely and when the execution reaches a conditional statement that has symbolic variables, the execution forks. Then, the execution resumes concretely. After the execution stops for all the forks, a set of constraints that represent each execution path that returns true is generated. NAVEX combines the form HTML constraints F_{html} and the JavaScript constraints F_{js} to produce the final form constraints F_{form}. As an example, the constraints for the form in our running example (Listing 1) are:

F_{html}: `(book_name=="Intro to CS by author1"` $\vee$ `book_name=="Intro to Math by author2")`
F_{js}: `edition` $>$ `0`
F_{form}: $F_{html} \wedge F_{js}$

Finally, the formula f_{form} is sent to the solver to find a solution. NAVEX uses the solver solution, form *method*, and *action* fields to issue a new HTTP request to the application (i.e., `http:.../selectBooks.php?action=borrow POST[book_name=Intro to CS by author1, edition=2]`).

Addressing Server-side Constraints. Server-side code often introduces additional constraints on the values of the input variables, which can influence the navigation structure of an application. Most commonly, these include constraints over the values submitted via forms. For instance, in Listing 1, the server-side code introduces an additional check over the string length of `$publisher`, which is not present in the JavaScript validation.

Typically, when the server constraints are satisfied, the execution proceeds and the state of the application is changed, while in the opposite case, the application rejects the form inputs and the state of the application does not change. Therefore, to maximize the coverage of the application, the crawler must be able to generate form inputs that are accepted by the application.

While automatically generating form inputs that are rejected is easier, generating inputs that are accepted is more challenging. To deal with this challenge, we utilize an execution-tracing engine on the server-side code. NAVEX uses the produced trace information to determine whether a request is successful by checking if the application is (*i*) changing its state (i.e., creating a new session, setting a new variable and superglobal values, etc.) and (*ii*) performing sensitive operations such as querying the database.

When a request is not successful, NAVEX utilizes the trace information to perform a concolic execution. In particular, it first retrieves the executed statements including the conditional statements. Then, the collected

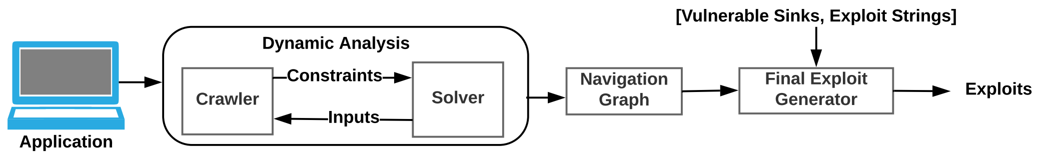

Figure 3: Concrete Exploit Generation (Step II) Components.

conditional statements are transformed automatically to solver specifications and negated to uncover new execution paths. The newly created specifications are then sent to the solver to generate new form inputs. This process is continuously repeated until the form submission is successful. As an example, the above inference constructs the following constraints that yield to a successful form submission

`(book_name=="intro to CS by author1"` $\vee$
`book_name=="intro to Math by author2")` $\wedge$
`length(publisher)<=35` $\wedge$ `edition >0`

Finally, for each accepted form, NAVEX stores the full HTTP request that led to the successful submission.

3.2.2 Navigation Graph

The *Navigation Graph* produced by the dynamic execution step represents the applications' navigation behavior. It is a directed graph $G = (N, E)$ where each node $n \in N$ represents an HTTP request and each edge $e = (n_i, n_j) \in E$ represents a navigation from n_i to n_j, which can be of type *link* or *form*. In particular, for every edge $e = (n_i, n_j) \in E$ n_i represents the page from which the request was originated. Each node in the graph has the following properties `id, URL, role`, and `form_params` for nodes representing an HTTP request generated by a form submission. The `id` property stores a unique identifier of the node, the `URL` property is the URL in the HTTP request, which is composed of the module name and HTTP parameters of the request, and the `role` property holds the login credentials used as input to the crawler as illustrated in Figure 4. It is important to note that the navigation graph can contain multiple nodes associated with the same PHP module. In particular, if a PHP module can accept different combinations of input variables, each such combination is represented by a corresponding node in the NG.

A partial instance of an NG, related to our running example is shown in Figure 4. As an example, one possible form submission, with form input values generated by the solver, is represented by the edge between nodes 2 and 3, while the other edges represent *link* navigation. Note that `hold.php` is associated with two different nodes (id-s 5 and 6), each having a different combination of input variables (i.e., HTTP parameters). This representation will be crucial in the next step when exploring paths to the exploitable modules.

3.2.3 Final Exploit Generation

To generate the final concrete exploits, NAVEX utilizes the NG along with the vulnerable sinks identified by the techniques introduced in Section 3.1. One challenge that NAVEX must solve in this step is that of combining the results produced by the step of vulnerable sink identification with the Navigation Graph. In particular, when modules containing vulnerable sinks are included by other modules using PHP inclusion, the former does not appear in the NG, because there is no explicit navigation to them. For instance, the module `checkout.php` does not appear in the NG in Figure 4. To execute these vulnerable modules, the execution must invoke the including modules.

To address this issue, NAVEX executes a preprocessing *inclusion resolution* step, which creates an *inclusion map* that stores the file inclusion relationships. The map is constructed by performing a traversal that searches the enhanced CPG for nodes that represent calls to file inclusion PHP functions (e.g., `require, include, etc`).

Once the inclusion resolution step is completed, NAVEX uses the NG and the produced inclusion map to search paths on the NG from public modules to the exploitable modules (or their including parents). It is important to note that the previous identification of vulnerable sinks that 'may' be exploitable greatly reduces the cost of such search and increases the likelihood of finding executable exploits.

The search method is summarized in Algorithm 2. The first input to the search is the set of pairs $\{(module, exploit)\}$ from Step I of NAVEX. *Module* represents the vulnerable module, and *exploit* represents the assignments of malicious values to inputs generated by the solver. The next input is the `InclusionMap` and the `SeedURLs`, which represent the publicly accessible modules. For each vulnerable module, using the inclusion map and the parameters in the exploit, the algorithm first finds possible destination nodes, which will be the targets of the graph search (line 5). These nodes (`DestURLs`) represent either the vulnerable module or its parents (if a parent PHP module includes the vulnerable module). `GetDestURLs` returns only those nodes of the NG, whose parameter names match the parameter names appearing in the corresponding `exploit`. The function `ExpSearch` first identifies the nodes whose URL matches one of the `SeedURLs` (i.e., matches the `URL`

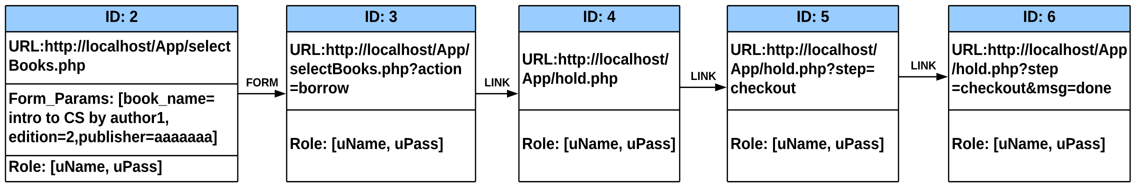
ID: 2
URL:http://localhost/App/selectBooks.php
Form_Params: [book_name=intro to CS by author1, edition=2,publisher=aaaaaaa]
Role: [uName, uPass]
FORM
ID: 3
URL:http://localhost/App/selectBooks.php?action=borrow
Role: [uName, uPass]
LINK
ID: 4
URL:http://localhost/App/hold.php
Role: [uName, uPass]
LINK
ID: 5
URL:http://localhost/App/hold.php?step=checkout
Role: [uName, uPass]
LINK
ID: 6
URL:http://localhost/App/hold.php?step=checkout&msg=done
Role: [uName, uPass]

Figure 4: The navigation graph (NG) of our running example.

Algorithm 2 Generating Concrete Exploits

```
1: Input: $ModulesAndExploits = \{(module, exploit)\}, InclusionMap, SeedURLs$
2: output: Concrete exploits for VulnModule
3:
4: for all $vm \in ModulesAndExploits$ do
5:     $DestURLs = \text{GETDESTURLS}(vm, InclusionMap)$
6:     $Exploit = \text{EXPSEARCH}(SeedURLs, DestURLs, vm)$
7:     $AllExploits \leftarrow Exploit$
8: return $AllExploits$
9:
10: function EXPSEARCH(SeedURLs, DestURLs, vm)
11:     $SrcNodes = \text{FINDSRCNODES}(SeedURLs)$
12:     for all $sn \in SrcNodes$ do
13:         $paths = \text{GETPATHSTO}(sn, DestURLs)$
14:         for all $path \in paths$ do
15:             $exploit = \text{REPLACEVULNPARAMS}(path, vm)$
16:             $ConcreteExploits \leftarrow exploit$
17:     return $ConcreteExploits$
```

property) (line 11). The traversal then explores the NG for each of the retrieved `SrcNodes` to find paths between the source node and the `DestURLs` (line 13). Finally, for each found path, it replaces the values of the HTTP parameters in the last edge with the malicious values generated by the solver.

Applying the algorithm to our running example, yields to considering `http://localhost/App/selectBooks.php` as a `SeedURL`, and the node with id 6 in Figure 4 as `DestURL`, because that node matches the vulnerable module, whose corresponding (XSS) exploit contains an assignment of a malicious value to the HTTP variable `msg`. Since the exploit string for `msg` is `<script> alert("XSS");</script>` (generated by the solver and stored in `exploit`), `GetPathsTo` explores the following navigation paths between the `SeedURL` and `DestURL`: (1) nodes of [id=2, id=3, id=4, id=5] and (2) nodes of [id=2, id=3, id=4, id=5, id=6]. However, it returns only the first navigation path because the URL of node 5 does not contain the HTTP parameter `msg`. Finally, `ReplaceVulnParams` function replaces the value of the `msg` with the malicious value of the exploit. As a result, NAVEX generates the following set of HTTP requests as a *concrete exploit* for the vulnerable sink (line 15) at Listing 3:

```
1. http://localhost/App/index.php
2. http://localhost/App/selectBooks.php with
   POST params:[book_name=intro to CS by
   author1, edition=2,publisher=aaaaaaa]
3. http://localhost/App/selectBooks.php?action
   =borrow
4. http://localhost/App/hold.php
5. http://localhost/App/hold.php?step=checkout
6. http://localhost/App/hold.php?step=checkout
   &msg=<script>alert("XSS");</script>
```

As can be noted, as a result of our dynamic execution and of the navigation graph design where nodes represent HTTP requests, the challenging problem of finding sequences of HTTP requests that execute an exploit is transformed into a simple graph search problem, which is efficient.

4 Implementation

The implementation of NAVEX is based on several existing tools, most of which were extended to deal with our problem. For Step I of our approach, the PHP extension [9] of code property graphs [33] was enhanced with additional tags to enable precise taint tracking and database constraints reasoning. The enhanced CPG is then imported to the Neo4j [4] graph database, an open source graph platform to create and query graph databases. The graph traversals, such as algorithm 1, are written in Gremlin [1]. Neo4j and Gremlin are also used in Step II to build and search the navigation graph.

For constraints solving, we leveraged Z3 solver [17] and its extension Z3-str [35]. In particular, when graph traversals report a vulnerable path to a sink, NAVEX analyzes the returned path and its nodes. Based on each node type, a Three-Address Code (TAC) formula that represents the node is created automatically. The TAC Formula consists of right operand (*rightOp*), *operator*, and left operand (*leftOp*), node type, and unique node id. Then, NAVEX starts analyzing each TAC formula according to its *type*. Based of the *operator, leftOp*, and *rightOp*, NAVEX generates: (1) appropriate Z3 variable declarations, (2) a set of assertions that replicate the semantics of the PHP *operator* in Z3 specification, and (3) an assertion that assigns appropriate attack strings from our attack dictionary to each sink variable in the formula. NAVEX supports assignment, unary, binary, conditional, built-in function, and cast statement types. The TAC formula analysis and Z3 translation engine code are approximately 3600 Java LOC.

For Step II, we extended crawler4j [2] by adding support for collecting forms and JavaScript code, extracting constraints from the forms, and generating Z3 assertions. To deal with JavaScript, we used an extension of the Narcissus JavaScript engine [3], which adds the ability to evaluate JavaScript code symbolically. Then, constraints extracted from JavaScript related to form inputs are transformed and combined with the form constraints and solved by Z3.

To generate server-side execution traces, we leveraged Xdebug [5], an open source debugger for PHP code. Note that Xdebug, like any debugging tool, imposes performance issues such as HTTP responses delays due to trace generation. Therefore, to maintain acceptable performance, NAVEX invokes Xdebug and analyzes traces on demand.

Broadly, the techniques implemented in NAVEX can be used to generate exploits for non-PHP web applications. At an implementation level, NAVEX is applicable to other server-side languages if the target source code is represented as CPGs, models of the target language features (i.e., built-in functions, operators, etc.) as solver specifications are available, and suitable server-side execution tracing tool is used.

NAVEX is an open-source software available at https://github.com/aalhuz/navex

5 Evaluation

Dataset. We evaluated NAVEX on 26 real-world PHP applications with a combined codebase of 3.2M SLOC and 22.7K PHP files as shown in Table 1. Our criteria for selecting the applications include: (*i*) evaluation on the latest versions of popular, complex and large PHP applications such as `Joomla, HotCRP, and WordPress`, and (*ii*) comparison of NAVEX on the same test applications used by state-of-the-art work in exploit generation (e.g., Chainsaw [7]) and vulnerability analysis (e.g., RIPS [15], [16]).

Setup. NAVEX was deployed on Ubuntu 12.04 LTS VM with 2-cores of 2.4GHz each and 40GB RAM. We first generated the enhanced CPG and used it to find exploitable paths for all the 26 applications. Then, we deployed the applications that have exploitable paths. The deployment process includes: installing each application on a server, creating login credentials for each role, and populating the application database with initial data by navigating the application and submitting forms when necessary. We take a snapshot of each application's database and use it after each crawling to restore the original state of the database. Note that due to specific deployment instructions for each application, we could not leverage automation to include more applications to evaluate. Given ample time for manual deployment, NAVEX can be used to analyze and generate exploits for hundreds or thousands of applications.

Application (version)	PHP Files	PHP SLOC
myBloggie (2.1.4)	56	9090
Scarf Beta	19	978
DNscript	60	1322
WeBid (0.5.4)	300	65302
Eve (1.0)	8	905
SchoolMate (1.5.4)	63	15375
geccbblite (0.1)	11	323
FAQforge (1.3.2)	17	1676
WebChess (0.9)	29	5219
WordPress (4.7.4)	699	181257
HotCRP (2.100)	145	57717
HotCRP (2.60)	43	14870
Zen-Cart (1.5.5)	1010	109896
OpenConf (6.71)	134	21108
osCommerce (2.3.4)	684	63613
osCommerce (2.3.3)	541	49378
Drupal (8.3.2)	8626	585094
Gallery (3.0.9)	510	39218
Joomla (3.7.0)	2764	302701
LimeSurvey (3.1.1)	3217	965164
Collabtive (3.1)	836	172564
Elgg (2.3.5)	3201	215870
CPG (1.5.46)	359	305245
MediaWiki (1.30.0)	3680	537913
phpBB (2.0.23)	74	29164
phpBB (3.0.11)	387	158756

Table 1: Subject applications of our evaluation.

AST, CFG, PDG, and sanitization and DB tags generation	1hr 25m
Graph database size	4.15 GiB
Total # nodes	24,418,552
Total # edges	56,060,195

Table 2: Statistics on the enhanced CPG generation.

Summary of results. NAVEX constructed a total of 204 exploits, of which 195 are on injection, and 9 are on logic vulnerabilities. The sanitization-tags-enhanced CPG reduced false positives (FPs) by 87% on average. The inclusion of client-side code analysis for building the navigation graph enhanced the precision of exploit generation by 54% on average. On the evaluation set, NAVEX was able to drill down as deep as 6 HTTP requests to stitch together exploits.

Enhanced code property graph statistics. For all the applications under test, Table 2 shows the enhanced CPG construction time and size. Note, the enhanced graph represents the source code of all the 26 applications under test, indicating the low runtime overhead of NAVEX.

Navigation graph statistics. Table 3 summarizes the total time to generate concrete exploits in Step II of NAVEX. The application list in the table represents the applications for which NAVEX found exploitable paths. Therefore, if an application did not have any exploitable path, NAVEX will not model its navigation behavior. The number of roles reflects the number of all account types (privileges) for each application. The NG has approximately 59*K* nodes and 1*M* edges.

5.1 Exploits

SQLI Exploits. NAVEX examined calls to

Application	Total Crawling, Forms Spec. Generation, Solving Time & NG Building Time	# of Roles
myBloggie	2m	2
SchoolMate	0	5
WebChess	1m 36sec	2
Eve	1m 5sec	1
geccbblite	57sec	1
Scarf	1m 44sec	2
FAQforge	47sec	1
WeBid	9m 29sec	2
DNscript	51sec	1
phpBB2	2m 14sec	2
HotCRP (2.60)	30m 13sec	4
osCommerce (2.3.3)	2hr 6m 32sec	2
CPG	24m 40sec	2
MediaWiki	15m 30sec	1
LimeSurvey	46sec	2
osCommerce (2.3.4)	2hr 19m 1sec	2
OpenConf	2m 1sec	2
Gallery3	5m 51sec	2
Collabtive	24m 2sec	3
Total time	6hr 27m 18sec	
Graph database size	104.44 MiB	

Table 3: Statistics on the Navigation graph generation.

`mssql_query`, `mysql_query`, `mysqli_query`, and `sqlite_query` as sinks for SQLI vulnerability. It reported a total of 155 SQLI exploitable sinks with a running time of 37m and 45sec. From these, it generated 105 concrete SQLI exploits in 7m and 76sec as summarized in Table 4.

NAVEX generated SQLI exploits for all applications that have SQLI exploitable sinks (seeds) except for `SchoolMate`. In `SchoolMate`, the crawler recovered only three HTTP requests. This application has 5 different roles, and for each role, our crawler was able to log in successfully. However, each time the crawler sends an HTTP request after the login, the application redirects the execution to the login page, which means that the application does not properly maintain user sessions. Therefore, the crawler did not proceed, and the coverage was low. This faulty application was chosen in our evaluation mainly to compare the results of NAVEX with other related work that included it in their test applications. The reported exploitable sinks, nevertheless, are confirmed to be true positives (TPs).

Selected SQLI Exploit. One of the applications for which NAVEX generated a large number of SQLI exploits is `WeBid`. Listing 4 shows an exploitable sink located in the user interface. An authenticated user can check other users' messages (line 3), consequently, the messages will be flagged as read (line 6). The generated exploit for both sinks is in Listing 5.

```
1 $messageid = $_GET['id']; //no sanitization
2 //1st vul. query
3 $sql = "SELECT * FROM '".$DBPrefix."messages' WHERE
      'id'='$messageid'";
4 ....
5 //2nd vul. query
6 $sql = "UPDATE '".$DBPrefix."messages' SET 'read'='1' WHERE
      'id'='$messageid'";
```

Listing 4: Simplified code for SQLI vulnerability in *WeBid*.

Application	SQLI Exp. Sinks	TPs	FPs	SQLI Exploits
myBloggie	22	22	0	22
Scarf	0	0	0	0
DNscript	1	1	0	1
WeBid	40	40	0	40
Eve	5	5	0	5
SchoolMate	50	50	0	0
geccbblite	4	4	0	4
FAQforge	14	14	0	14
WebChess	13	13	0	13
osCommerce (2.3.3)	1	1	0	1
phpBB (2.0.23)	5	5	0	5
Total	**155**	**155**	**0**	**105**

Table 4: Summary of the generated SQLI exploitable sinks and exploits.

Application	XSS Exp. Sinks	TPs	FPs	XSS exploits
myBloggie	2	2	0	2
Scarf	1	1	0	1
DNscript	1	1	0	1
WeBid	12	8	4	8
Eve	2	2	0	2
SchoolMate	11	11	0	0
FAQforge	7	7	0	7
WebChess	14	14	0	14
HotCRP (2.60)	5	5	0	5
osCommerce (2.3.4)	5	5	0	5
osCommerce (2.3.3)	46	45	1	42
CPG	11	11	0	0
MediaWiki	1	1	0	1
phpBB (2.0.23)	15	15	0	2
Total	**133**	**128**	**5**	**90**

Table 5: Summary of the generated XSS seeds and exploits.

```
http://localhost/WeBid/user_login.php
    POST[username=user,password=pass,action=login]
http://localhost/WeBid/index.php
http://localhost/WeBid/user_menu.php
http://localhost/WeBid/yourmessages.php?id=1' OR '1'='1
```

Listing 5: SQLI exploit generated for the sinks in Listing 4.

XSS Exploits. NAVEX examined calls to `echo` and `print` PHP functions as sinks for XSS vulnerability. It found a total of 133 XSS exploitable sinks, 5 of which are false positives, in 1h and 49m. It successfully generated 90 XSS exploits for the 133 sinks in 40m and 12sec as shown in Table 5. For all exploitable sinks, NAVEX generated XSS exploits except for `SchoolMate`, due to the reported problem.

Note, we consider an exploit a zero-day if the exploit in an active application was not reported before and has a significant effect, which is not the case for the vulnerability in `MediaWiki` for instance.

Selected XSS Exploit. For `osCommerce2.3.4`, NAVEX generated 5 XSS exploits. In the following, we demonstrate one of these exploits, which illustrates the precision of our analysis in capturing the effect of custom and built-in sanitization functions along different paths to sinks.

Listing 6 shows the vulnerable sink (`echo`) where user input `$HTTP_GET_VARS['page']` passes through 3 different functions and it is finally processed by either `htmlspecialchars` or `strtr` PHP functions. NAVEX did not report the paths going through `htmlspecialchars` as exploitable because it is a sufficient XSS sanitization function. On the other hand, it reported the paths that include `strtr`, which is not a typical sanitization function for XSS, as vulnerable. In this example, `strtr` replaces double quotes with `&quot;` which is not sufficient to prevent XSS. NAVEX inferred the semantics of this function (through its modeling of many PHP functions as solver specifications) and used the solver to find an XSS attack string that does not include double quotes from our XSS attack dictionary. Additionally, to break out the outer single quotes, the attack string should have a single quote (&#39; HTML entity) encoded (%26%2339%3B).

As a result, the solver selected `%26%2339%3B-alert(1)-%26%2339%3B` as a malicious user input that satisfies the path constraints. Listing 7 shows the exploit constructed automatically for this vulnerability.

```
echo '<tr .. onclick="document.location.href=\'' .
      tep_href_link(FILENAME, 'page=' .
      $HTTP_GET_VARS['page']) . '\'">';
//1st function
function tep_href_link($page = '', $parameters = '') {
 if (tep_not_null($parameters))
  $link .= $page . '?' . tep_output_string($parameters);
        ...}
//2nd function
function tep_output_string($string, $translate = false,
      $protected = false) {
if ($protected == true)
 return htmlspecialchars($string);
else
  if ($translate == false)
     return tep_parse_input_field_data($string, array('"' =>
           '&quot;'));
...}
//3rd function
function tep_parse_input_field_data($data, $parse) {
  return strtr(trim($data), $parse);}
```

Listing 6: Simplified code for XSS vulnerability in *osCommerce 2.3.4*.

```
http://localhost/oscommerce-2.3.4/catalog/admin/login.php
      ?action=process
      POST[username=admin@test.com,password=pass]
http://localhost/oscommerce-2.3.4/catalog/admin/index.php
http://localhost/oscommerce-2.3.4/catalog/admin/reviews.php
http://localhost/oscommerce-2.3.4/catalog/admin/reviews.php
      ?page=%26%2339%3B-alert(1)-%26%2339%3B
```

Listing 7: An XSS exploit generated for Listing 6.

EAR Exploits. NAVEX examined a total of 246 calls to `header` function (EAR source) in 17m and 17sec. It found 19 benign EAR and 3 malicious EAR vulnerabilities. It successfully generated 9 exploits for the 22 EAR vulnerabilities combined as summarized in Table 6. Note that in the case of EAR, an exploit is a sequence of HTTP requests causes the code after the redirection function to execute.

Code Execution Exploits. NAVEX examined all calls to

Application	Benign EAR Sinks	Malicious EAR Sinks	FPs	EAR Exploits
myBloggie	7	0	0	0
WeBid	0	1	0	1
Eve	1	0	0	1
HotCRP (2.100)	1	0	0	1
HotCRP (2.60)	1	0	0	1
OpenConf	4	0	1	1
osCommerce (2.3.4)	0	1	0	1
osCommerce (2.3.3)	0	1	0	1
Gallery	2	0	0	0
Joomla	0	0	1	0
LimeSurvey	1	0	0	0
Collabtive	1	0	0	1
MediaWiki	1	0	1	1
Total	**19**	**3**	**3**	**9**

Table 6: Summary of the generated EAR seeds and exploits.

the PHP function `eval`, a total of 98 calls in our data set, in 21m and 20sec. All the calls are not vulnerable, and therefore, NAVEX did report any exploitable code execution sinks, and no exploits were generated.

Command Injection Exploits. NAVEX examined all calls to `exec`, `expect_popen`, `passthru`, `pcntl_exec`, `popen`, `proc_open`, `shell_exec`, `system`, `mail`, and `backtick operator`, a total of 350 calls, in 22m and 32sec. NAVEX did not find any vulnerable sinks.

File Inclusion Exploits. NAVEX examined a total of 8063 calls to `include`, `include_once`, `require`, and `require_once` in 27m and 58sec. It marked 1 sink as exploitable in `WeBid`. However, an exploit could not be generated because the unsanitized file name (user input) is prefixed and postfixed with some constant strings, which cannot be overwritten by a malicious input.

5.2 Measurements

Performance and scalability. Figure 5 shows the performance of NAVEX measured by the total time to find exploitable sinks and to generate exploits per vulnerability type. Note, for each vulnerability type, the blue bar shows the total time of the analysis of Step I, for *all* applications under test. The orange bar, on the other hand, records the total time spent by Step II, for the applications that have exploitable sinks.

Dynamic analysis coverage. We consider the number of statically identified vulnerabilities by Step I as a baseline to assess the coverage of Step II. NAVEX successfully constructed 105 exploits for 155 SQLI sinks, 90 exploits for 128 XSS sinks, and 9 exploits for 19 EAR vulnerabilities. Overall, the total coverage of Step II is 68% in comparison with the total vulnerable sinks for all applications.

Effect of sanitization tags on code property graphs. Figure 6 shows the effect of enhancing the CPG with sanitization and DB tags on the total number of vulnerable sinks. The orange bar shows the total number of vulnerable sinks with the enhancements, showing reductions in false positives. Overall, the number of reported vulner-

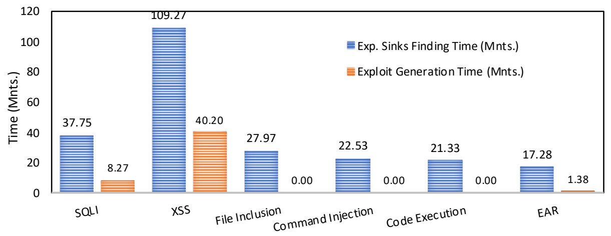

Figure 5: Performance of NAVEX for each vulnerability type. Note, zero values refer to the absence of exploits.

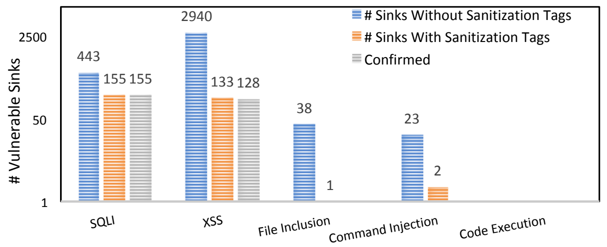

Figure 6: The effect of sanitization-tag-enhanced CPG in reducing false positives in vulnerable sink finding. For SQLI, the numbers show the # of sinks using sanitization and DB properties.

able sinks for each vulnerability type is reduced, on an average, by 87% due to enhancements implemented on CPGs to significantly cut-down false positives.

Effect of client-side code analysis. One of the contributions of our work is the precise handling of client-side code during the NG construction. Forms are common artifacts in modern web applications. In our dataset, we counted the frequency of using forms to receive data from users. We found out that the number of unique forms in all applications ranges from 3 (as in `geccbblite`) to 186 (as in `WeBid`) with an average of 45 form/application. Additionally, Figure 7 validates our claim that in order to improve the coverage and consequently generate more exploits in deployed applications, we must support input generation and constraints extraction from forms and JavaScript code. It can be seen from Figure 7 that NAVEX's precision significantly increases.

Additionally, we measured the maximum length of all navigation paths leading to all exploitable sinks. For SQLI and EAR exploits, we found that the maximum exploit length is 5 whereas for XSS is 6.

5.3 Comparison with Related Work

We compare the results of NAVEX with other related works based on the following: (1) common subject applications (and same version numbers), (2) common vulnerability types, and (3) knowledge of how the results of the related work are counted. Several related work met those criteria such as `CRAXweb` [22], `RIPS` [15], [16], [31], `Ardilla` [25], and `Chainsaw` [7]. However, since `Chainsaw` [7], the most recent related work, provided a detailed comparison between their work and [22], [31],

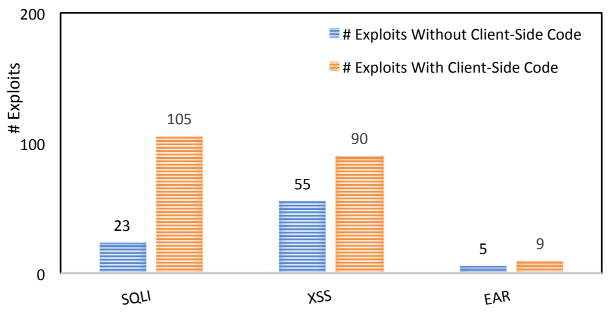

Figure 7: The enhancement on exploit generation precision due to client-side code analysis.

and [25], we compare NAVEX with `Chainsaw`, `RIPS`, and [16].

Vulnerability detection. In Table 7, we compare `RIPS`, `Chainsaw`, and [16] with NAVEX in terms of the total number of the reported SQLI and XSS vulnerabilities. Compared to `Chainsaw`, NAVEX found the same number of XSS and SQLI vulnerabilities in `scarf` and `Eve`, nevertheless, it reported more vulnerable sinks for `myBloggie`. In addition, NAVEX found 71 vulnerable sinks in `HotCRP`, `osCommerce`, and `phpBB` because it can handle object-oriented PHP code, which is not available in `Chainsaw`. Compared to `RIPS`, NAVEX found 19 more vulnerable sinks for `phpBB`, `osCommerce`, and `myBloggie`. It missed 2 vulnerable sinks in `HotCRP` due to missing edges in the code property graph that represent dynamic function calls.

Exploit generation. Since `Chainsaw` supports generating exploits for XSS and SQLI, we compare it to NAVEX with respect to the total number of the generated SQLI and XSS exploits as well as some performance measurements (see Table 8). NAVEX constructed 19 more exploits in `WeBid`, `myBloggie`, `geccbblite`, `WebChess`, and `FAQforge`, and achieved the same for `Eve`, `scarf`, and `DNscript`. For `SchoolMate`, NAVEX did not generate exploits due to issues related to maintaining users sessions (as discussed earlier). Since in `Chainsaw` the exploit generation is done statically, it was able to generate exploits for this application.

NAVEX significantly outperformed `Chainsaw` in terms of efficiency. `Chainsaw` generated the exploits in 112min while NAVEX took 25min and 2sec. In addition, we contrast the total time to build and search the navigation graph in NAVEX (18m 26sec) with the total time to construct and search the Refined Workflow Graph (RWFG) (1day 13h 21m) in `Chainsaw`. This indicates that the techniques used in NAVEX improved the exploit generation efficiency without losing precision.

5.4 Limitations and Discussion

Unsupported features. Certain features of web applications are not yet supported and therefore limit our coverage. For example, forms that have `inputs` of type `file` require the user to select and upload an actual file from

Application	RIPS [15]	[16]	Chainsaw [7]	NAVEX
myBloggie	21	SQLI(5)	22	24
Scarf	-	SQLI(1)	1	1
Eve	-	-	7	7
HotCRP (2.60)	7	-	-	5
osCommerce (2.3.3)	42	-	-	46
phpBB (2.0.23)	8(SQLI)	-	-	20

Table 7: Comparison on the number of identified (SQLI+XSS) vulnerable sinks.

Application	Chainsaw [7]	NAVEX
Eve	7	7
SchoolMate	54	0
WebChess	25	27
FAQforge	8	21
geccbblite	3	4
myBloggie	22	24
Scarf	1	1
DNscript	2	2
WeBid	47	48
Total exploit generation time	112m	25m 2sec
Total NG construction & solving time	1day 13h 21m	18m 26sec

Table 8: Comparison on the number of generated (SQLI+XSS) exploits.

the local system. In a given test setting, this can be made to work with our solver, but to make this work across all platforms requires more engineering effort. Another issue is of deriving TAC formulas from graph nodes automatically. It is a challenging process that involves analyzing each AST node and supporting different node structures for each node type. For example, the left-hand side of an assignment statement in PHP can be a simple variable, a constant, a function call, nested function calls, etc. We have carefully considered these cases, and NAVEX has the support for most such node types and structures, yet there are a few instances still under development. In our data set, NAVEX incorrectly flagged only 5 sinks as XSS exploitable in `osCommerce2.3.3` and `WeBid`. In PHP, statically handling dynamic calls to functions is challenging. NAVEX utilizes CPGs, which do not have full support for resolving dynamic function calls. However, this did not have a big impact on the results reported by NAVEX. For instance, there were 3 false positives reported for EAR vulnerability in `Joomla`, `OpenConf`, and `MediaWiki`.

6 Related Work

Exploit generation for web applications. Exploit generation has seen a lot of interest in binary application [8, 14, 21]. For web applications, the closest work to NAVEX is Chainsaw [7], a system that uses purely static analysis to build concrete exploits. NAVEX differs from Chainsaw in 2 aspects: (*i*) it performs a combination of dynamic and static analyses, which enables it to better scale to large applications and to find more exploits, (*ii*) it supports finding exploits for multiple classes of vulnerabilities. Additional related works include Ardilla [25], which uses concolic execution and taint tracking to construct SQLI and XSS attack vectors; CRAXweb [22], which employs concrete and symbolic execution supported by a constraint solver to generate SQLI and XSS exploits. QED [27] generates first-order SQLI and XSS attacks using static analysis and model checking for Java web applications. [32] generates inputs that expose SQLI vulnerabilities using concolic execution of PHP applications. EKHunter [19] combines static analysis and constraint solving to find exploits in for-crime web applications. WAPTEC [13] and NoTamper [12] generate exploits for parameter-tampering vulnerabilities. These works, however, are limited to single PHP modules and do not consider whole-application paths.

Modeling with code property graphs. Yamaguchi et al. [33] introduced the notion of CPGs for vulnerability modeling and discovery in C programs. In a follow-up work [9], they applied CPGs for vulnerability discovery on PHP applications. While our work uses the flexibility and efficiency that CPGs offer, our problem goes a step further to generate actual executable exploits. As a consequence, we enhance CPGs with additional attributes.

Vulnerability analysis. There is a large body of research that studied server-side vulnerability detection. Broadly, there are static analysis approaches (such as [11, 15, 16, 18, 23, 24, 26, 29–31, 34]), dynamic analysis approaches (e.g., [20, 28]), and hybrid approaches (such as [10]). Although NAVEX employs some of these analysis techniques to find vulnerabilities, the aim of NAVEX is different from these works as it constructs exploits for the identified vulnerabilities. Our navigation modeling is inspired by MiMoSA [11], which is a system that finds data and workflow vulnerabilities by analyzing modules of web applications. NAVEX advances the analysis by combining static and dynamic analyses to construct concrete exploits for large web applications.

7 Conclusions

In this paper, we present NAVEX, an automatic exploit generation system that takes into account the dynamic features and the navigational complexities of modern web applications. On our dataset, NAVEX constructed a total of 204 exploits, of which 195 are on taint-style vulnerabilities, and 9 are on logic vulnerabilities. We demonstrated that NAVEX significantly outperforms prior work on the precision, efficiency, and scalability of exploit generation.

Acknowledgments

We thank Curt Thieme for his support with the applications' deployment. We also thank 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 FA8650-15-C-7561.

References

[1] Apache tinkerpop. https://tinkerpop.apache.org/gremlin.html, 2018. Accessed: 2018-05-1.

[2] crawler4j. https://github.com/yasserg/crawler4j, 2018. Accessed: 2018-05-1.

[3] Narcissus. https://github.com/mozilla/narcissus/, 2018. Accessed: 2018-05-1.

[4] The neo4j graph platform the #1 platform for connected data. https://neo4j.com/, 2018. Accessed: 2018-05-1.

[5] Xdebug - debugger and profiler tool for php. https://xdebug.org/, 2018. Accessed: 2018-05-1.

[6] Xss filter evasion cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, 2018. Accessed: 2018-05-1.

[7] ALHUZALI, A., ESHETE, B., GJOMEMO, R., AND VENKATAKRISHNAN, V. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), ACM, pp. 641–652.

[8] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. AEG: Automatic Exploit Generation. In *NDSS* (2011), vol. 11, pp. 59–66.

[9] BACKES, M., RIECK, K., SKORUPPA, M., STOCK, B., AND YAMAGUCHI, F. Efficient and flexible discovery of php application vulnerabilities. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on* (2017), IEEE, pp. 334–349.

[10] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)* (2008), pp. 387–401.

[11] BALZAROTTI, D., COVA, M., FELMETSGER, V. V., AND VIGNA, G. Multi-module Vulnerability Analysis of Web-based Applications. In *the 14th ACM Conference on Computer and Communications Security (CCS)* (2007), pp. 25–35.

[12] BISHT, P., HINRICHS, T., SKRUPSKY, N., BOBROWICZ, R., AND VENKATAKRISHNAN, V. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 607–618.

[13] BISHT, P., HINRICHS, T., SKRUPSKY, N., AND VENKATAKRISHNAN, V. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *the 18th ACM conference on Computer and communications security* (2011), pp. 575–586.

[14] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), pp. 143–157.

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

[16] DAHSE, J., AND HOLZ, T. Static Detection of Second-Order Vulnerabilities in Web Applications. In *23rd USENIX Security Symposium (USENIX Security)* (2014), pp. 989–1003.

[17] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[18] DOUPÉ, A., BOE, B., KRUEGEL, C., AND VIGNA, G. Fear the ear: discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 251–262.

[19] ESHETE, B., ALHUZALI, A., MONSHIZADEH, M., PORRAS, P. A., VENKATAKRISHNAN, V. N., AND YEGNESWARAN, V. EKHunter: A Counter-Offensive Toolkit for Exploit Kit Infiltration. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015).

[20] HALDAR, V., CHANDRA, D., AND FRANZ, M. Dynamic taint propagation for java. In *21st Annual Computer Security Applications Conference (ACSAC)* (2005), pp. 9–pp.

[21] HU, H., CHUA, Z. L., ADRIAN, S., SAXENA, P., AND LIANG, Z. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), USENIX Association, pp. 177–192.

[22] HUANG, S., LU, H., LEONG, W., AND LIU, H. CRAXweb: Automatic Web Application Testing and Attack Generation. In *IEEE 7th International Conference on Software Security and Reliability, SERE* (2013), pp. 208–217.

[23] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web* (2004), ACM, pp. 40–52.

[24] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A Static Analysis tool for Detecting Web Application Vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on* (2006), pp. 6–pp.

[25] KIEYZUN, A., GUO, P. J., JAYARAMAN, K., AND ERNST, M. D. Automatic Creation of SQL Injection and Cross-Site Scripting Attacks. In *IEEE 31st International Conference on Software Engineering (ICSE)* (2009), pp. 199–209.

[26] LIVSHITS, V. B., AND LAM, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *14th USENIX Security Symposium* (Baltimore, Maryland, USA, 2005).

[27] MARTIN, M., AND LAM, M. S. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium* (2008), pp. 31–43.

[28] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J., AND EVANS, D. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference* (2005), Springer, pp. 295–307.

[29] SAMUEL, M., SAXENA, P., AND SONG, D. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), pp. 587–600.

[30] SAXENA, P., MOLNAR, D., AND LIVSHITS, B. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), pp. 601–614.

[31] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices* (2007), vol. 42, ACM, pp. 32–41.

[32] WASSERMANN, G., YU, D., CHANDER, A., DHURJATI, D., INAMURA, H., AND SU, Z. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis* (2008), pp. 249–260.

[33] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 590–604.

[34] YU, F., ALKHALAF, M., AND BULTAN, T. Stranger: An automata-based string analysis tool for php. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2010), pp. 154–157.

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