Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications

Oswaldo Olivo Isil Dillig Calvin Lin
{olivo, isil, lin}@cs.utexas.edu
Department of Computer Science
The University of Texas at Austin

ABSTRACT

This paper describes a new class of denial-of-service (DoS) attack, which we refer to as *Second Order DoS* attacks. These attacks consist of two phases, one that pollutes a database with junk entries and another that performs a costly operation on these entries to cause resource exhaustion. The main contribution of this paper is a static analysis for detecting second-order DoS vulnerabilities in web applications. We have implemented our analysis in a tool called TORPEDO, and we show that TORPEDO can successfully detect second-order DoS vulnerabilities in widely used web applications written in PHP. Once our tool discovers a vulnerability, it also performs symbolic execution to generate candidate attack vectors. We evaluate TORPEDO on six widely-used web applications and show that it uncovers 37 security vulnerabilities, while reporting 18 false positives.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis.

Keywords

Static analysis; Program Analysis; Denial-of-Service; Web Applications; Second-Order Vulnerabilities.

1. INTRODUCTION

Web applications form the backbone of the modern Internet and provide a plethora of useful services, including banking, commerce, education, blogging, and social networking. Since web applications do not require the user to install any software beyond a standard web browser, they are enormously popular. Unfortunately, this growing popularity has also made web applications a desirable target for many kinds of security exploits, including denial-of-service (DoS) attacks.

DoS attacks, which can render websites inaccessible and can cause significant financial damage to website owners, come in two flavors. *Network-based DoS attacks* require an attacker to flood a target server with many requests, thereby saturating server resources and rendering the target web service unavailable. In contrast, *application-level DoS attacks* take advantage of an underlying vulnerability in the web application and either cause the server to crash or exhaust its computational resources. These application-level DoS attacks are typically more dangerous and cannot be prevented using standard network-level defense mechanisms [19, 2].

In this paper, we address the issue of application-level DoS attacks that cause excessive CPU usage. In particular, we identify a new type of DoS attack, which we refer to as Second Order DoS attacks, because like recent work on second-order XSS and SQLI vulnerabilities [10], these attacks consist of two phases: In the first step, the attacker uses a bot to pollute the database with junk entries. In the second step, the attacker performs some expensive computation over the junk entries in the database, rendering the application unavailable for a considerable amount of time.

These second-order DoS attacks are made possible by the presence of lurking security vulnerabilities in web applications. Specifically, the first phase of the attack is feasible because the application does not employ an appropriate defense mechanism, such as a *CAPTCHA*, that prevents users from inserting database entries through a bot. Similarly, the second part of the attack is possible because the application does not sanitize the retrieved database entries (e.g., by bounding their size). Furthermore, such DoS attacks cannot be prevented using standard network-based defense mechanisms: Since the inserted database entries are typically small in size and temporally separated, second-order DoS attacks use low-bandwidth and can evade detection by techniques that monitor anomalous network traffic.

A key contribution of this paper is a static analysis for automatically identifying second-order DoS vulnerabilities. Our method consists of two consecutive taint analyses for detecting the vulnerability, followed by a symbolic execution phase for attack vector generation. The goal of our first static analysis is to infer *tainted database attributes:* In this context, a database attribute A is said to be tainted if a query on A can yield an unbounded number of database entries. The second phase of our analysis takes these tainted database attributes as input and checks for the existence of an $\Omega(n)$ computation over them. If these conditions are met, the application contains a second-order DoS vulnerability. In particular, the attacker can now taint a database attribute A by inserting many entries that share the same value for A, and he can then issue a carefully crafted query Q that

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813680.

triggers an expensive (at least linear-time) computation over the results of Q. Once our taint analysis detects a potential second-order DoS vulnerability, we also perform backwards symbolic execution to generate candidate attack vectors in order to help developers understand, confirm, and fix the vulnerability.

We have implemented our technique in a tool called TORPEDO, which analyzes PHP programs. Our evaluation on several widely-used PHP applications shows that TORPEDO can uncover serious security vulnerabilities. To summarize, this paper makes the following contributions:

- We argue that the notion of second-order vulnerabilities—well-known in the context of XSS and SQLI attacks—is also applicable in the context of denial-of-service attacks. We give a precise definition of *second-order DoS vulnerabilities*, and we explain how attackers can exploit them to cause CPU exhaustion while evading detection by network-based defense mechanisms.
- We present a novel static analysis for automatically detecting second-order DoS vulnerabilities. Unlike previous work that can assume that sanitizers exist as pre-defined functions, our technique is based on new expanded notions of *taint* and *sanitization* that take into account the state of the database and its impact on CPU usage.
- We describe a backwards symbolic execution algorithm for generating candidate attack vectors. This technique allows us to assess the impact of the uncovered security vulnerabilities without requiring a human to manually generate inputs.
- We empirically demonstrate that when applied to a set of widely-used PHP applications, TORPEDO can find 37 security vulnerabilities with a false detection rate of 33%.

2. BACKGROUND

In this section, we provide relevant background on PHP and relational databases.

2.1 Background on PHP Scripts

PHP is one of the most popular programming languages for web development, with approximately 82% of server-side scripts implemented using PHP [25]. Specifically, PHP makes it convenient to create *dynamic web pages* that interact with the user and customize page content based on user preferences. There are two key reasons why PHP is so popular for web application development: (1) HTML integration, and (2) native support for relational databases.

HTML integration. One of the most useful features of PHP is the way it allows programmers to handle HTML forms. Specifically, when a user fills out an HTML form, it is possible to invoke a PHP script with the user data stored in so-called *superglobals* that are available in every scope. Two of the most commonly used PHP superglobals are `$_GET` and `$_POST`, both of which are mappings from keys to values (called *arrays* in PHP terminology). If a web form contains an input field named x which is sent using the HTTP POST (resp. GET) method, then `$_POST["x"]` (resp. `$_GET["x"]`) holds the value of the user input for field x. For instance, consider the following HTML form:

```
<form action="example.php" method="get">
Name: <input type="text" name="name">
<input type="submit"> </form>
```

and the corresponding PHP script called "example.php":

```
echo $_GET["name"];
```

Here, when the user enters their name into the `name` field of the above HTML form and clicks submit, a PHP script called "example.php" gets executed. Furthermore, since the web form specifies the data submission method to be HTTP GET, the user input is stored in the superglobal variable `$_GET["name"]`. Thus, the net effect of the above script is to simply print out the name entered into the web form.

Database support. Another attractive feature of PHP is its native support for databases. The most popular database system used with PHP is MySQL, which is an open-source, cross-platform relational database. Certain built-in PHP commands allow scripts to connect to a MySQL database and request content that is typically displayed on a webpage. For the purposes of this paper, the most relevant command is the `mysqli_query` function, which takes as input a string corresponding to the database query. For instance, consider the following PHP code:

```
$name = $_GET["name"];
$query = "SELECT * FROM Customers WHERE Name=$name";
$result = $mysqli->query($query);
```

This code selects from the `Customers` database exactly those people whose name matches the user input. Insertions into the database are performed in a similar way, also using the `mysqli_query` function.

2.2 Relational Databases

In relational databases, such as MySQL, data is organized into *tables* (or *relations*) of rows and columns where there is a unique key associated with each row. In this model, each row is a *tuple* representing a single data item, and each column is a labeled *attribute* of the tuple. Given a relation R with set of attributes A and a formula φ such that $\text{vars}(\varphi) \subseteq A$, a selection operation $\sigma_\varphi(R)$ selects all tuples in R that satisfy condition φ. Similarly, given a set A' and a relation R with attributes A such that $A' \subseteq A$, a projection operation $\pi_{A'}(R)$ yields a new relation R' that includes all rows of R but only those columns whose name is in A'. Hence, the following MySQL query:

```
SELECT Author FROM Papers
WHERE title = "X" AND author="Y"
```

corresponds to the operation $\pi_{author}(\sigma_\varphi(papers))$ where φ represents the formula $title = \text{“}X\text{”} \wedge author = \text{“}Y\text{”}$.

In the remainder of this paper, we refer to any set of tuples retrieved from relation R as a *view* of R. Hence, the result of any MySQL query of the form:

```
SELECT ... FROM MyTable WHERE ...
```

corresponds to a view of `MyTable`. Note that every relation R is also a view of itself. Given some view R, we write $|R|$ to denote the number of tuples in R.

3. DEFINING SECOND ORDER DENIAL-OF-SERVICE VULNERABILITIES

In this section, we first give a precise definition of *second-order DoS vulnerabilities* and then discuss some common mechanisms that programmers employ for avoiding such security holes.

DEFINITION 1. **(Second-Order DoS Vulnerability)** *Let P be a program using a database table R, and let V_R be a view of R. We say that P contains a second-order DoS vulnerability if:*

1. *The quantity $|V_R|$ can be controlled by a bot*
2. *The worst-case resource usage of P is $\Omega(|V_R|)$*

As explained in this definition, there are two necessary conditions that enable a second-order DoS attack. First, the application must allow a bot to control the result size of some query on database table R. Second, the resource usage of the application must be at least linear in the size of this attacker-controlled view of R. If the application satisfies both of these conditions, then an attacker can insert junk entries into R in a way that drives $|V_R|$ to ∞ in the limit. Furthermore, since the worst-case resource usage of the application is proportional to $|V_R|$, the attacker can then craft inputs that trigger this excessive resource usage behavior.

In practice, there are several ways to avoid second-order DoS vulnerabilities in web applications. The most common way to protect against second-order DoS attacks is to perform some sort of Turing test before inserting any user input into the database. Hence, in this context, *CAPTCHA*s and other similar mechanisms (e.g., text message verification) provide a form of sanitization that prevents bots from polluting a database.

However, performing a Turing test is not the only way to defend web applications against second-order DoS attacks. For example, another form of sanitization is to require administrator credentials or to bound the size of a view V_R before performing computation whose resource usage behavior depends on $|V_R|$. For example, consider a web application that iterates over a collection obtained using the following database query:

```
$res = SELECT * FROM Papers WHERE Author=$author
```

Here, if the application disallows the insertion of more than 10 papers by the same author into the database, then the worst-case resource usage of the application will be bound by a constant and will therefore not be susceptible to a second-order DoS attack.

4. STATIC ANALYSIS

In this section, we describe our algorithm for statically detecting second-order DoS vulnerabilities. As shown schematically in Figure 1, our approach consists of two consecutive static taint analyses:

- ***Phase I:*** The goal of the first phase is to identify *tainted database attributes*. We say that an attribute $\mathcal{K}$ of some database table $\mathcal{R}$ is tainted if $|\sigma_{\varphi_\mathcal{K}}(\mathcal{R})|$ can be controlled by a bot, where $\varphi_\mathcal{K}$ is a condition involving attribute $\mathcal{K}$. Hence, our first static analysis determines which user inputs can reach which attributes of a database table without being sanitized. In this context, a sanitizer is a piece of code that prevents a bot from arbitrarily increasing the size of $\sigma_{\varphi_\mathcal{K}}(\mathcal{R})$.
- ***Phase II:*** In the second phase of our analysis, we start with tainted database attributes inferred in Phase I and then perform taint propagation to identify query results whose sizes may be arbitrarily large. Our analysis then issues a warning if the number of executions of a loop is controlled by such a *tainted query result*.

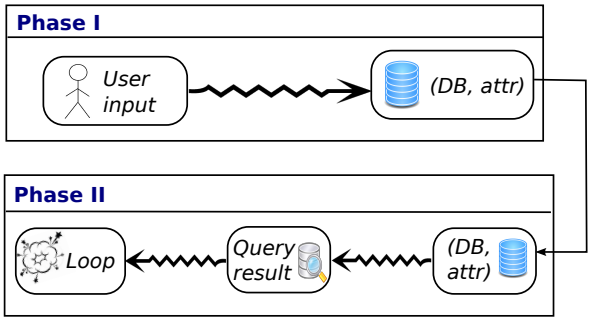

Figure 1: Schematic illustration of our approach. Here, squiggly arrows indicate flows between different resources.

$$
\begin{array}{lcl}
\text{Binop } \oplus & \in & \{+, -, ==, !=, <, >, ...\} \\
\text{Expr } e & := & c \mid v \mid e_1 \oplus e_2 \mid e_1[e_2] \mid \mathit{count}(e) \\
\text{Cond } \Phi & := & \mathcal{K} = v \mid \Phi_1 \text{ AND } \Phi_2 \mid \Phi_1 \text{ OR } \Phi_2 \\
\text{Stmt } S & := & v := e \mid S_1; S_2 \mid f(e_1, \ldots, e_k) \\
 & & \mid \text{if } (e) \text{ then } S_1 \text{ else } S_2 \\
 & & \mid \text{foreach } (v_1 \text{ as } v_2)\ S \\
 & & \mid \text{INSERT } (v_1, ..., v_n) \text{ INTO } \mathcal{R} \\
 & & \mid v := \text{SELECT } (\mathcal{K}_1, ..., \mathcal{K}_n) \\
 & & \quad \text{FROM } \mathcal{R} \text{ WHERE } \Phi
\end{array}
$$

Figure 2: Language used for describing our analysis

In the rest of this section, we will use the simplified language of Figure 2 to formally describe our static analyses. In particular, we consider a simple PHP-like language that has built-in support for database operations, such as INSERT and SELECT. Expressions in our simplified language include constants (1, "xyz", $\{1, 2, 3\}, \ldots$), variables, and binary operations $e_1 \oplus e_2$ where $\oplus \in \{+, -, *, =, \leq, ...\}$. In addition, we allow array reads $e_1[e_2]$ to model reading from PHP superglobals, such as `$_GET` and `$_POST`. Finally, the expression $\mathit{count}(e)$ yields the size of collection e.

Continuing with the grammar of Figure 2, statements include assignments $v := e$, composition $S_1; S_2$, if statements, and loops of the form foreach$(v_1$ as $v_2)$ S where v_1 is a collection (i.e., array or map), and v_2 is bound to each element v_1. In what follows, we will use function calls $f(e_1, \ldots, e_k)$ to model various kinds of sanitizers. To simplify our presentation, we only consider the two most important database operations, namely INSERT and SELECT[1]. Database insertion operations have the syntax

$$\text{INSERT } (v_1, \ldots, v_k) \text{ INTO } \mathcal{R}$$

[1]Our actual implementation handles most MYSQL commands, not just `INSERT` and `SELECT`.

where $\mathcal{R}$ is the name of a database table and $(v_1, \ldots, v_k)$ is a tuple to be inserted into $\mathcal{R}$. Selection operations have the syntax:

$$\text{SELECT } (\mathcal{K}_1, \ldots, \mathcal{K}_n) \text{ FROM } \mathcal{R} \text{ WHERE } \Phi$$

Here, each $\mathcal{K}_i$ is the name of an attribute of table $\mathcal{R}$ and Φ is a condition used for filtering relevant tuples in $\mathcal{R}$. Note that condition Φ is a boolean combination of atomic predicates of the form $\mathcal{K} = v$ where $\mathcal{K}$ is the name of an attribute and v is a variable.

4.1 Phase I Analysis

As mentioned earlier, the first phase of our algorithm performs static taint analysis to identify tainted database attributes. Here, taint sources correspond to user inputs, and sinks are database insertions. Our analysis distinguishes between two classes of sanitizers:

- ***Full sanitizers:*** Such sanitizers protect the application against bots. Examples of full sanitizers include Turing tests (e.g., *CAPTCHAs* or SMS verification) as well as administrative credential checks. We refer to these checks as full sanitizers because they sanitize *every* input received by the application henceforth.
- ***Conditional sanitizers:*** These checks sanitize a particular variable v for some attribute $\mathcal{K}$ of database table $\mathcal{R}$. Specifically, if v is *conditionally sanitized* for $(\mathcal{R}, \mathcal{K})$, this means that the value stored in variable v cannot occur an unbounded number of times in attribute $\mathcal{K}$ of table $\mathcal{R}$. The following PHP function exemplifies such a conditional sanitizer:

```
cond_sanitize(v) {
   $rows = SELECT * FROM Contacts where name = $v;
   if(count($rows) == 0) return true;
   return false;
}
```

  This function returns true iff attribute `name` of table `Contacts` does not already contain value v. Since this function is used to prevent insertion of duplicate names in the `Contacts` table, it sanitizes v for context (*Contacts*, *name*). These kinds of conditional sanitizers are ubiquitous in real code.

Since our analysis needs to differentiate between full and conditional sanitizers, our taint analysis is somewhat non-standard. We describe our Phase I static analysis in Figure 3 using judgments of the form:

$$b, \Gamma \vdash S : b', \Gamma', \Lambda$$

Here, b is a boolean value, referred to as a *bot checker*, indicating whether we have encountered a full sanitizer in the code analyzed so far. Environment Γ, called the *conditional taint environment*, maps each program variable to a propositional formula ϕ and is used to reason about conditional sanitization. Specifically, formula ϕ is used to represent all contexts under which a variable is tainted and is formed according to the following grammar:

$$\phi \;:=\; \mathit{true} \mid \mathcal{R}_{\mathcal{K}} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

Here, $\mathcal{R}_{\mathcal{K}}$ is a boolean variable representing context $(\mathcal{R}, \mathcal{K})$. Hence, if $\Gamma(v) = \mathcal{R}_{\mathcal{K}}$, this means that v is tainted under context $(\mathcal{R}, \mathcal{K})$. On the other hand, if $\Gamma(v) = \neg\mathcal{R}_{\mathcal{K}} \wedge \neg\mathcal{R}'_{\mathcal{K}'}$, then v is tainted in all contexts except $(\mathcal{R}, \mathcal{K})$ and $(\mathcal{R}', \mathcal{K}')$.

Going back to the judgment $b, \Gamma \vdash S : b', \Gamma', \Lambda$, we use a set Λ to keep track of tainted database attributes $(\mathcal{R}, \mathcal{K})$. Hence, the judgment $b, \Gamma \vdash S : b', \Gamma', \Lambda$ means the following: Assuming we analyze statement S in an environment where b indicates the presence/absence of a full sanitizer and Γ indicates conditional taint information, the analysis of statement S produces a new bot checker b', a new conditional taint environment Γ' and a set Λ of tainted database attributes. Thus, for a program P, if our analysis produces the judgment

$$\mathit{false}, [\,] \vdash P : b, \Gamma, \Lambda$$

then the set Λ gives us all the tainted database attributes inferred by phase I.

$$(\mathit{Input}) \quad \frac{\Gamma' = \Gamma[v \mapsto \mathit{true}]}{b, \Gamma \vdash v := \mathit{user\_input}() : b, \Gamma', \emptyset}$$

$$(\mathit{Asn.}\ 1) \quad \frac{e \notin \mathrm{dom}(\Gamma)}{b, \Gamma \vdash v := e : b, \Gamma, \emptyset}$$

$$(\mathit{Asn.}\ 2) \quad \frac{e \in \mathrm{dom}(\Gamma)}{b, \Gamma \vdash v := e : b, \Gamma[v \mapsto \Gamma(e)], \emptyset}$$

$$(\mathit{Stz.}\ 1) \quad \frac{}{b, \Gamma \vdash \mathit{full\_sanitizer}() : \mathit{true}, \Gamma, \emptyset}$$

$$(\mathit{Stz.}\ 2) \quad \frac{\Gamma' = \Gamma[v \mapsto (\Gamma(v) \wedge \neg\mathcal{R}_{\mathcal{K}})]}{b, \Gamma \vdash \mathit{cond\_sanitizer}(v, \mathcal{R}, \mathcal{K}) : b, \Gamma', \emptyset}$$

$$(\mathit{Insert}) \quad \frac{\mathit{Attributes}(\mathcal{R}) = (\mathcal{K}_1, \ldots, \mathcal{K}_n) \quad \Lambda = \{(\mathcal{R}, \mathcal{K}_i) \mid b = \mathit{false} \wedge \Gamma(v_i) \not\Rightarrow \neg\mathcal{R}_{\mathcal{K}_i}\}}{b, \Gamma \vdash \text{INSERT } (v_1, ..., v_n) \text{ INTO } \mathcal{R} : b, \Gamma', \Lambda}$$

$$(\mathit{Seq}) \quad \frac{b, \Gamma \vdash S_1 : b_1, \Gamma_1, \Lambda_1 \quad b_1, \Gamma_1 \vdash S_2 : b_2, \Gamma_2, \Lambda_2}{b, \Gamma \vdash S_1; S_2 : b_2, \Gamma_2, \Lambda_1 \cup \Lambda_2}$$

$$(\mathit{If}) \quad \frac{b, \Gamma \vdash S_1 : b_1, \Gamma_1, \Lambda_1 \quad b, \Gamma \vdash S_2 : b_2, \Gamma_2, \Lambda_2 \quad \Gamma' = \Gamma_1 \sqcup \Gamma_2 \quad \Lambda' = \Lambda_1 \cup \Lambda_2}{b, \Gamma \vdash \text{if}(e) \text{ then } S_1 \text{ else } S_2 :\ b_1 \wedge b_2, \Gamma', \Lambda'}$$

Figure 3: Phase I static analysis

Let us now consider the rules from Figure 3 in more detail. According to the first rule labeled *Input*, a call to function *user_input* is a taint source; hence variable v is unconditionally tainted after this statement. The next two rules describe taint propagation for assignments $v := e$ and cause variable v be to become tainted under the same contexts as e.

The next two rules, labeled *Stz 1* and *Stz 2*, describe the analysis of full and conditional sanitizers, respectively. According to *Stz 1*, the analysis of full sanitizers simply sets the bot checker to *true*.[2] The second rule labeled *Stz 2*

[2]The careful reader may wonder why we do not analyze full sanitizers by simply setting Γ to be the empty map. While this strategy can be made sound, it would be imprecise be-

analyzes conditional sanitizers that bound the number of occurrences of v that can appear in attribute $\mathcal{K}$ of table $\mathcal{R}$. Since variable v is now sanitized for context $(\mathcal{R}, \mathcal{K})$, we only propagate taint for v when $\neg\mathcal{R}_{\mathcal{K}}$ is true. Even though we model conditional sanitizers using the function call *cond_sanitizer*$(v, \mathcal{R}, \mathcal{K})$, our analysis automatically infers PHP/SQL expressions that perform such conditional sanitization. By contrast, our analysis requires manual annotations for full sanitizers (please see Section 6).

The rule labeled *Insert* describes the analysis of database insertions, which correspond to taint sinks. To understand this rule, suppose that we are performing an insertion into database table $\mathcal{R}$ which has attributes $\mathcal{K}_1, \ldots, \mathcal{K}_n$. Now, if we have previously performed full sanitization, we know that this insertion is not being performed by a bot. Hence, if b is *true*, then $|\sigma_{\varphi_{\mathcal{K}}}(\mathcal{R})|$ is not controlled by a bot, and $(\mathcal{R}, \mathcal{K})$ should not become tainted. Similarly, if we have performed conditional sanitization to restrict the number of occurrences of value v_i in the $\mathcal{K}_i$'th attribute of $\mathcal{R}$, then $|\sigma_{\varphi_{\mathcal{K}_i}}(\mathcal{R})|$ is bounded; hence, $(\mathcal{R}, \mathcal{K}_i)$ is not tainted. Thus, our analysis considers an insertion to be a sink for $(\mathcal{R}, \mathcal{K}_i)$ only when b is false and $\Gamma(v_i) \not\Rightarrow \neg\mathcal{R}_{\mathcal{K}_i}$ (i.e., it is possible that v_i is tainted under context $(\mathcal{R}, \mathcal{K}_i)$).

The last two rules of Figure 3 summarize taint propagation for sequences and if statements. In the *Seq* rule, observe that we take the union of Λ_1 and Λ_2 because a database attribute $(\mathcal{R}, \mathcal{K})$ is tainted if it is either tainted in S_1 *or* S_2.

Finally, let us consider taint propagation for a conditional statement of the form if(e) then S_1 else S_2. Here, we can only be sure that a full sanitization has been performed if it has been performed in both branches; thus, our analysis takes the *conjunction* of b_1 and b_2. On the other hand, since we want to overapproximate tainted database attributes, we propagate the union of Λ_1 and Λ_2. Similarly, since we also want to be conservative about taint information for variables, we compute the join ($\sqcup$) of Γ_1 and Γ_2, defined as follows:

$$(\Gamma_1 \sqcup \Gamma_2)(v) = \begin{cases} \Gamma_1(v) & \text{if } v \notin \text{dom}(\Gamma_2) \\ \Gamma_2(v) & \text{if } v \notin \text{dom}(\Gamma_1) \\ \Gamma_1(v) \vee \Gamma_2(v) & \text{otherwise} \end{cases}$$

In other words, this join operation ensures that variable v is tainted if it is tainted in either branch.

Observe that the rules from Figure 3 omit the analysis of loops and selection operations because (1) our analysis unrolls loops a fixed number of times, and (2) selection operations are effectively no-ops for the first analysis.

4.2 Phase II Analysis

The second phase of our algorithm performs a different kind of static taint analysis to infer $\Omega(n)$ computation over *tainted query results (views)*. For the Phase II analysis, taint sources are elements of set Λ (i.e., tainted database attributes) inferred by the Phase I analysis. On the other hand, taint sinks are loops that iterate over collections. Unlike the Phase I analysis where we need to differentiate between two classes of sanitizers, the notion of sanitization for the Phase II analysis is more straightforward: Specifically, we say that a collection v is sanitized if its size is bounded.

The taint propagation rules for Phase II are described in Figure 4 using judgments of the form:

$$\Lambda, \Upsilon \vdash S : \Upsilon', \mathcal{E}$$

Here, Λ is the output of the first static analysis (i.e., tainted database attributes), and $\mathcal{E}$ is a boolean variable indicating whether a vulnerability has been encountered. The set Υ is a set of variables that correspond to tainted query results. Essentially, our phase II analysis propagates taint arising from selection operations and issues a warning when the number of loop iterations depends on a tainted variable $v \in \Upsilon$.

Let us now consider the rules from Figure 4 in more detail. The first rule for assignments $v := e$ simply propagates taint to v if e is tainted. The second rule analyzes sanitizers that perform some bounds checking on the size of a collection v. In this case, since the size of collection of v is bounded, we simply remove v from the set of tainted variables Υ. While Figure 4 models sanitizers using a function call *sanitize(v)*, our implementation automatically infers sanitization expressions that perform bounds checking on the size of collections.

The most interesting aspect of Phase II is the analysis of selection operations. Here, given a set Λ of tainted database attributes, we need to infer whether the result of a selection query is also tainted. Unsurprisingly, this depends on the expression Φ used in the WHERE clause of the query. For this purpose, our analysis utilizes the helper rules shown in Figure 5. Given tainted attributes Λ and a database table $\mathcal{R}$, these rules decide whether to propagate taint or not. In particular, given a query whose WHERE clause is Φ, the judgment $\Lambda, \mathcal{R} \vdash \Phi$: *true* indicates that taint should be propagated to the result of the query.

The first two rules in Figure 5 deal with the base case where Φ is an atomic predicate of the form $\mathcal{K} = v$. In this case, if $(\mathcal{R}, \mathcal{K})$ is not tainted (i.e., $(\mathcal{R}, \mathcal{K}) \notin \Lambda$), then the result of the query cannot be unbounded; hence, $\Lambda, \mathcal{R} \vdash \mathcal{K} = v$: *true* if and only if $(\mathcal{R}, \mathcal{K}) \in \Lambda$. If the WHERE clause involves AND, then the result is tainted iff both Φ_1 and Φ_2 are tainted. In contrast, Φ_1 OR Φ_2 yields true iff $\Lambda, \mathcal{R} \vdash \Phi_i$: *true* for some $i \in \{1, 2\}$.

Based on this discussion, let us go back to the *Select* rule from Figure 4. As expected, the query result v becomes tainted if and only if $\Lambda, \mathcal{R} \vdash \Phi$: *true*; hence, we only add v to set Υ under this condition. Since the rules *Seq* and *If* are similar to taint propagation from the first phase, we do not describe them in detail.

In Figure 4, the last rule for loops describes the analysis of *sinks*. In particular, since the loop iterates over collection v_1, the CPU usage of the program is $\Omega(count(v_1))$.[3] Furthermore, if $v_1 \in \Upsilon$, we know that the size of v_1 may be unbounded, so the analysis issues a warning if $v_1 \in \Upsilon$. Note that this rule also propagates taint for the loop body.

5. ATTACK VECTOR GENERATION

So far, we have described a static analysis for identifying second-order DoS vulnerabilities. However, our static analysis has two main limitations: First, it can have false positives. Second, even when the tool reports a true positive, it can be hard to understand the conditions under which the detected vulnerability can be exploited. Hence, to help programmers understand and confirm the warnings

cause the program may apply full sanitization before asking the user for input.

[3]Unless there is a *break* or *return* statement, in which case the attacker should insert values in the first phase that don't trigger these statements. In our experience, we don't find this to be an obstacle for the attacker.

$$(Assign) \quad \frac{\Upsilon' = \begin{cases} \Upsilon \cup \{v\} & \text{if } e \in \Upsilon \\ \Upsilon & \text{if } e \notin \Upsilon \end{cases}}{\Lambda, \Upsilon \vdash v := e : \Upsilon', \mathit{false}}$$

$$(Sanitize) \quad \frac{}{\Lambda, \Upsilon \vdash \mathit{sanitize}(v) : \Upsilon \backslash \{v\}, \mathit{false}}$$

$$(Select) \quad \frac{\Lambda, \mathcal{R} \vdash \Phi : c \qquad \Upsilon' = \begin{cases} \Upsilon \cup \{v\} & \text{if } c = \mathit{true} \\ \Upsilon & \text{if } c = \mathit{false} \end{cases}}{\Lambda, \Upsilon \vdash v := \text{SELECT } (\mathcal{K}_1, ..., \mathcal{K}_n) \text{ FROM } \mathcal{R} \text{ WHERE } \Phi : \Upsilon', \mathit{false}}$$

$$(Seq) \quad \frac{\Lambda, \Upsilon \vdash S_1 : \Upsilon_1, \mathcal{E}_1 \qquad \Lambda, \Upsilon_1 \vdash S_2 : \Upsilon_2, \mathcal{E}_2}{\Lambda, \Upsilon \vdash S_1; S_2 : \Upsilon_2, \mathcal{E}_1 \vee \mathcal{E}_2}$$

$$(If) \quad \frac{\Lambda, \Upsilon \vdash S_1 : \Upsilon_1, \mathcal{E}_1 \qquad \Lambda, \Upsilon \vdash S_2 : \Upsilon_2, \mathcal{E}_2}{\Lambda, \Upsilon \vdash \text{if}(e) \text{ then } S_1 \text{ else } S_2 : \ \Upsilon_1 \cup \Upsilon_2, \mathcal{E}_1 \vee \mathcal{E}_2}$$

$$(Loop) \quad \frac{\Upsilon' \supseteq \Upsilon \qquad \Lambda, \Upsilon' \vdash S : \Upsilon', \mathcal{E}}{\Lambda, \Upsilon \vdash \text{foreach}(v_1 \text{ as } v_2) \ S : \ \Upsilon', (v_1 \in \Upsilon) \vee \mathcal{E}}$$

Figure 4: Inference rules describing the second phase of static analysis

$$(Base\ 1) \quad \frac{(\mathcal{R}, \mathcal{K}) \in \Lambda}{\Lambda, \mathcal{R} \vdash \mathcal{K} = v : \mathit{true}}$$

$$(Base\ 2) \quad \frac{(\mathcal{R}, \mathcal{K}) \notin \Lambda}{\Lambda, \mathcal{R} \vdash \mathcal{K} = v : \mathit{false}}$$

$$(AND) \quad \frac{\Lambda, \mathcal{R} \vdash \Phi_1 : c_1 \qquad \Lambda, \mathcal{R} \vdash \Phi_2 : c_2}{\Lambda, \mathcal{R} \vdash \Phi_1 \text{ AND } \Phi_2 : c_1 \wedge c_2}$$

$$(OR) \quad \frac{\Lambda, \mathcal{R} \vdash \Phi_1 : c_1 \qquad \Lambda, \mathcal{R} \vdash \Phi_2 : c_2}{\Lambda, \mathcal{R} \vdash \Phi_1 \text{ OR } \Phi_2 : c_1 \vee c_2}$$

Figure 5: Helper rules for SELECT

generated by the tool, we have also developed an *attack vector generation engine*. In the context of second-order DoS vulnerabilities, an *attack vector* consists of the following:

- **Component 1:** A set S of tuples inserted into the database, together with other user inputs that are needed to trigger these insertion operations
- **Component 2:** A particular database query that causes the application to perform some expensive computation over S (again, along with other user inputs that are necessary for triggering this behavior)

As shown schematically in Figure 6, our approach to automated attack generation is based on backwards symbolic execution and constraint solving. Specifically, given a pair of program locations π_0, π_1 associated with sources and sinks,

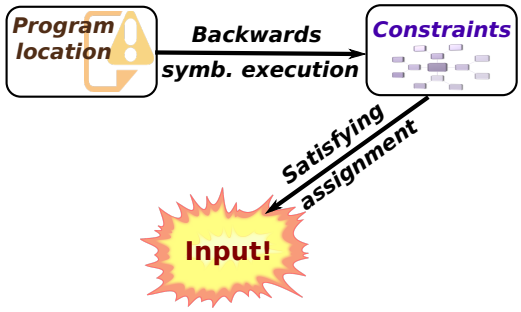

Figure 6: Illustration of attack vector generation

we perform backwards symbolic execution to collect a set of constraints ϕ describing the conditions under which execution will reach from π_0 to π_1. We then use an SMT solver to find a satisfying assignment σ to ϕ and use σ to generate a candidate attack vector.

5.1 Backwards Symbolic Execution

Given source and sink locations π_0, π_1, the goal of our symbolic execution is to generate a constraint ϕ that describes the conditions under which control will reach from π_0 to π_1. This analysis is described in Figure 7 using judgments of the following shape:

$$\phi, b, \pi_0, \pi_1 \vdash S : \phi', b'$$

Here, π_0 and π_1 are the source and sink locations respectively and b is a boolean value indicating whether we have reached the source. Given condition ϕ, formula ϕ' represents the constraint under which control will reach π_1 starting from the current statement S. Hence, if our analysis

produces the judgment:

$$false, false, \pi_0, \pi_1 \vdash P : \phi, \_$$

then ϕ yields the condition under which π_1 is reachable from π_0 in program P.

Let us now consider the rules from Figure 6 in more detail. According to rule 1, if we encounter a sink statement at program point π_1, we know that π_1 is unconditionally reachable from this statement; hence, we propagate *true*. On the other hand, if we encounter a source at program point π_0, we have reached the desired source; hence, we set the boolean variable b to *true* to indicate that we should stop computing weakest preconditions.

Continuing with rule (3) for assignment $v := e$, we use the standard rule for weakest precondition computation by substituting e for v in constraint ϕ. However, if we have already reached the source (i.e, b is true), we do not need to compute weakest preconditions; hence, the generated constraint is $(b \rightarrow \phi) \wedge (\neg b \rightarrow \phi[e/v])$. Rule (4) for composition $(S_1; S_2)$ also computes weakest preconditions in the standard way.

Rule (5) for conditionals is a bit more involved. First, we compute the weakest precondition of ϕ with respect to the then and else branches. Now, if we have encountered the source π_0 in either branch, we need to stop propagating the weakest precondition; hence the new value of boolean b' is $b_1 \vee b_2$. Assuming we have not yet encountered the desired source, the new precondition is computed as $(e \wedge \phi_1) \vee (\neg e \wedge \phi_2)$. To see why this is correct, consider two cases: If there is sink π_1 is in the then branch, then condition e needs to be true for control to reach π_1; hence, we conjoin e with ϕ_1. On the other hand, if the desired sink is in the else branch, $\neg e$ needs to hold for control to reach π_1. Note that at least one of ϕ_1 and ϕ_2 is guaranteed to be false because the sink location cannot be in both branches simultaneously.

The last rule describes the analysis of `SELECT` statements. This rule is similar to the assignment rule, except that we now substitute v with a more complex term t that involves set comprehensions. In particular, term t is a set that contains all tuples in $\mathcal{R}$ satisfying condition Φ.

5.2 Using Constraints to Generate Inputs

We now describe how to use the constraints from Section 5.1 to generate attack vectors. As in Section 4, the attack vector generation consists of two phases: In the first phase, the input to the analysis is a (user input, database insertion) pair discovered to be a feasible source-sink flow in the Phase I static analysis of Section 4.1. Similarly, for the second phase of attack vector generation, the input is a (database selection, loop) pair that is deemed to be a feasible source-sink flow according to the Phase 2 static analysis (Section 4.2).

The second phase of attack vector generation is simpler than the first phase because we only need to compute a single input. For this purpose, we get a satisfying assignment σ to the constraint generated using backwards symbolic execution. The input values given by σ, together with the database query for the sink, are now used to generate the second component of the attack vector.

The first phase of attack vector generation is a bit more involved since we need to generate a *sequence* of tuples, rather than a single input. A key challenge here is that the constraint ϕ generated using symbolic execution can refer to database $\mathcal{R}$, which evolves as we insert new tuples into the database. To address this challenge, we employ an iterative algorithm that repeatedly instantiates constraint ϕ with respect to the actual tuples inserted into the database. In particular, given a set variable $\mathcal{R}$ used in constraint ϕ, we first initialize it to the empty set and get a satisfying assignment σ_0 for $\phi[\emptyset/\mathcal{R}]$. This assignment σ_0 is now used to generate a concrete tuple t to insert into the database. In the next iteration, we instantiate $\mathcal{R}$ with the set $\{t\}$ and ask for a new satisfying assignment σ_1 to $\phi[\{t\}/\mathcal{R}]$. This process continues until we have inserted a sufficiently large number of tuples into the database.

EXAMPLE 1. *We illustrate attack vector generation using a realistic code pattern commonly found in PHP programs:*

```
L1: x := $_POST["a"]; y := $_POST["b"]; z := $_POST["c"];
    if (z == "1") {
       v := SELECT (k1, k2) from R where k2 = y;
       if(count(v)==0)
L2:       INSERT (x, y) INTO R;
    }
```

Our goal is to generate a sequence of tuples that can reach the database insertion at program point `L2` *starting from program location* `L1`*. Using backwards symbolic execution, we first generate the following constraint ϕ:*

$$\$\_\texttt{POST}[\text{“}\texttt{c}\text{”}] = \texttt{1} \wedge$$
$$count(\{(e.\mathcal{K}_1, e.\mathcal{K}_2) \mid e \in \mathcal{R} \wedge e.\mathcal{K}_2 = \$\_\texttt{POST}[\text{“}\texttt{b}\text{”}]\}) = 0$$

To generate the first tuple, we instantiate $\mathcal{R}$ with $\emptyset$, which yields `$_POST["c"] = "1"`*. Hence, the values of* `$_POST["a"]` *and* `$_POST["b"]` *are unconstrained, but* `$_POST["c"]` *must be* `"1"`*; so suppose we generate the input* `"aa", "bb", "1"`*. In the next iteration, we instantiate $\mathcal{R}$ with* `{ ("aa", "bb") }` *in ϕ. This new constraint now tells us that, in addition to* `$_POST["c"] = "1"`*, we need* `$_POST["b"]` $\neq$ `"bb"`*. Hence, we now pick the next input to be* `"aa", "bc", "1"`*. In the next iteration, we instantiate $\mathcal{R}$ with the set* `{ ("aa", "bb"), ("aa", "bc") }`*, which tells us that* `$_POST["c"] = "1"` $\wedge$ `$_POST["b"]` $\neq$ `"bb"` $\wedge$ `$_POST["b"]` $\neq$ `"bc"`*. This process continues until we have inserted sufficiently many tuples into the database.*

5.3 Constraint Solving

As described in the previous subsections, the formulas we need to solve for attack vector generation belong to the following constraint language :

$$\begin{array}{lll} \text{Term } t & := & c \mid x \mid f(t) \mid \{x \mid x \in S \wedge \phi\} \\ \text{Formula } \phi & := & \text{true} \mid \text{false} \mid \ t_1 \text{ op } t_2 \\ & & \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \phi_1 \rightarrow \phi_2 \end{array}$$

Here, c is a constant, x is a variable, f is an uninterpreted function, and S is a set. If we exclude *set comprehension terms* of the form $\{x \mid x \in S \wedge \phi\}$ from this constraint language, then our formulas would belong to the standard first order theory of equality with uninterpreted functions (and linear arithmetic) and could be directly solved using off-the-shelf SMT solvers. Unfortunately, to allow the computation of weakest preconditions in the presence of database queries, our analysis also needs to introduce set comprehension terms which are not supported by standard SMT solvers. For this purpose, we employ a customized decision procedure that overapproximates satisfiability by performing a transformation and using off-the-shelf SMT solvers. In particular, our algorithm consists of the following steps:

$$(1)\quad \frac{\pi = \pi_1}{\phi, b, \pi_0, \pi_1 \vdash sink_{@\pi} : true, b}$$

$$(2)\quad \frac{\pi = \pi_0}{\phi, b, \pi_0, \pi_1 \vdash source_{@\pi} : \phi, true}$$

$$(3)\quad \frac{\phi' = (b \rightarrow \phi) \wedge (\neg b \rightarrow \phi[e/v])}{\phi, b, \pi_0, \pi_1 \vdash v := e : \phi', b}$$

$$(4)\quad \frac{\begin{array}{c}\phi', b', \pi_0, \pi_1 \vdash S_1 : \phi'', b'' \\ \phi, b, \pi_0, \pi_1 \vdash S_2 : \phi', b'\end{array}}{\phi, b, \pi_0, \pi_1 \vdash S_1; S_2 : \phi'', b''}$$

$$(5)\quad \frac{\begin{array}{c}\phi, b, \pi_0, \pi_1 \vdash S_1 : \phi_1, b_1 \\ \phi, b, \pi_0, \pi_1 \vdash S_2 : \phi_2, b_2 \\ b' = b_1 \vee b_2 \\ \phi' = (e \wedge \phi_1) \vee (\neg e \wedge \phi_2)\end{array}}{\phi, b, \pi_0, \pi_1 \vdash \text{if}(e) \text{ then } S_1 \text{ else } S_2 : (b' \rightarrow \phi) \wedge (\neg b' \rightarrow \phi'), b'}$$

$$(6)\quad \frac{\begin{array}{c}t = \{(x.\mathcal{K}_1, \ldots, x.\mathcal{K}_n) \mid x \in \mathcal{R} \wedge \Phi[x.\mathcal{K}_i/\mathcal{K}_i]\} \\ \phi' = (b \rightarrow \phi) \wedge (\neg b \rightarrow \phi[t/x])\end{array}}{\phi, b, \pi_0, \pi_1 \vdash v := \text{SELECT } (\mathcal{K}_1, ..., \mathcal{K}_n) \text{ FROM } \mathcal{R} \text{ WHERE } \Phi : \phi', b}$$

Figure 7: Backward symbolic execution rules for generating attack vector constraints. Statements whose preconditions are not met (or are not shown in the figure) are assumed to be no-ops.

1. Replace each set comprehension term $t_i \in \phi$ with a fresh variable x_i and generate the formula $\phi' = \phi[x_i/t_i]$ and the mapping $\Gamma : \{x_i \mapsto t_i\}$
2. Let $\Gamma(x_i)$ be $\{x \mid x \in S_i \wedge \phi_i\}$. Now, generate the constraint:

$$\phi'' = (\bigwedge_{x_i \in dom(\Gamma)} (count(x_i) = 0 \rightarrow \bigwedge_{e_{ij} \in S_i} \neg\phi_i[e_{ij}/x]))$$

3. Use an off-the-shelf SMT solver to get a satisfying assignment to $\phi' \wedge \phi''$

It is easy to see that our procedure overapproximates satisfiability: In particular, we construct ϕ' by replacing set comprehension terms with fresh variables. Second, observe that $count(x_i) = 0$ implies that x_i must be the empty set; hence if x_i represents a set comprehension term $\{x \mid x \in S_i \wedge \phi_i\}$, we know that ϕ_i must evaluate to false for *all* elements in set S_i, which is expressed using ϕ''.

Since our procedure overapproximates satisfiability, it is possible that the inputs generated using our technique do not trigger the desired source-sink flow in reality. However, we have not observed this overapproximation to be a problem in practice. Specifically, the overwhelming majority of the set comprehension terms t_i appear in the form $count(t) = 0$; and, under this restriction, it can be shown that $\phi' \wedge \phi''$ is *equisatisfiable* to the original constraint ϕ.

6. IMPLEMENTATION

TORPEDO consists of approximately 4,000 lines of PHP code. We use Nikic's PHP parser [22] to obtain an initial AST and then construct our own CFG representation for each function. Since an SMT solver (recall Section 5) is needed to automatically generate attack vectors, TORPEDO uses the Z3 SMT solver [33] and its string solving plug-in [34] for finding satisfying assignments to constraints. For interprocedural analysis, TORPEDO conceptually performs function inlining by "gluing together" the CFGs of callees at method invocation sites.

Internally, TORPEDO consists of four different modules, namely, (1) static taint analysis, (2) symbolic execution engine, (3) sanitization inference, and (4) an engine for inferring database schemas. Since we have already described the taint analyzer and symbolic execution engine in detail, we now briefly outline the design of modules (3) and (4).

Sanitizer Inference Engine. To infer sanitizers, TORPEDO uses a combination of manual annotations and automated static analysis. For the Phase I static analysis from Section 4.1, TORPEDO requires manual annotations of full sanitizers, since automated inference of Turing tests is beyond the scope of static analysis. Hence, we have manually annotated *CAPTCHA* routines, as well as methods that check for website administrator credentials. However, TORPEDO does perform static analysis to infer conditional sanitizers that impose a bound on the size of data structures. Specifically, TORPEDO first statically analyzes the source code to generate constraints on collection sizes (for Phase II) and on the number of occurrences of keys in database tables (for Phase I). TORPEDO then uses an SMT solver to check if the generated constraints imply an upper bound on the size of some data structure of interest. Hence, a statement S is considered to be a sanitizer if the size of a data structure is unbounded before S but becomes bounded after S.

Database Schema Inference. Recall from Section 4 that our static analysis needs to know the attributes for a given database table, so TORPEDO performs a pre-analysis that infers the schema for each database table. Specifically, after parsing all files in the application, TORPEDO looks for

Application	Files	LOC	# TP	# FP
SCARF	19	1686	11	0
OpenConf	130	22,889	7	0
HotCRP	103	48,144	8	11
Gallery	505	62,699	1	0
osCommerce	702	86,693	5	3
Wordpress	479	261,564	5	4
Total	1938	483,675	37	18

Table 1: Summary of our experimental results. "TP" indicates true positives (i.e., real vulnerabilities), and "FP" denotes false positives.

Application	n=25,000	n=50,000	n=75,000	n=100,000
SCARF	6m32s	22m53s	*TO*	*TO*
OpenConf	2m26s	17m49s	*TO*	*TO*
HotCRP	7m46s	26m33s	*TO*	*TO*
Gallery	1m57s	6m12s	14m8s	29m47s
osCommerce	7m16s	15m35s	33m17s	*TO*
Wordpress	46s	2m11s	8m37s	21m53s

Table 2: Amount of time the server is unresponsive for different numbers (n) of entries inserted into the database during the first phase. *TO* means that the application becomes unresponsive for more than one hour. This data only includes a subset of the uncovered vulnerabilities.

`CREATE TABLE` instructions in the source code, and it records the ordered set of attributes associated with each table name. Since database queries in PHP can be generated dynamically through the use of string variables, Torpedo can only over-approximate the set of string values provided to the queries. As we discuss later, this source of imprecision is one of the main causes of false positives.

7. EVALUATION

We evaluate Torpedo by applying it to six server-side web applications written in PHP. Specifically, our benchmarks include HotCRP (a widely-used conference management software), WordPress (a popular blogging application), Gallery (a photo management application), SCARF (a research discussion forum), osCommerce (an online store management software), and OpenConf (another conference management system). In total, we use Torpedo to analyze 483,675 lines of PHP code. We perform our experiments on a MacBook Air laptop with Mac OSX 10.9.3, a 2 GHz Intel Core i7 processor, and 8 GB of RAM.

Table 7 summarizes our experimental results, showing that Torpedo finds a total of 37 vulnerabilities across six applications and only reports 18 false positives. On average, we see that Torpedo has a 33% false detection rate.

Discussion of true positives. For the true vulnerabilities detected by our tool, we use Torpedo's attack vector generation capability to confirm that the uncovered vulnerability can be exploited to launch a DoS attack. Specifically, we devise a bot to insert a large number of junk entries into a given database and then issue a query that triggers an $\Omega(n)$ computation over these entries. As expected, the severity of the DoS attack is proportional to the number of entries inserted during the first phase. Table 2 shows the number of database entries inserted during the first phase against the amount of time the server is unresponsive during the second phase. For example, for a vulnerability in HotCRP, when we insert 75,000 entries into the database in the first phase, we can bring down the server for more than an hour by issuing a *single* query in the second phase.

Upon manual inspection of the true vulnerabilities, we find that the second phase of the attack does not necessarily need to be carried out by a bot. In fact, all of the running times reported in Table 2 are caused by a single database query, so automation of the second phase is not a prerequisite for the DoS attack. By contrast, since the uncovered DoS attacks typically involve the insertion of thousands of entries into the database, automation is crucial to the first phase of the attack.

Another interesting aspect of the vulnerabilities is that a few tainted database attributes typically lead to several security vulnerabilities in the same application. In other words, many of the source-sink flows identified by Torpedo's Phase II taint analysis share the same source. For example, the 8 different vulnerabilities found in HotCRP involve two distinct tainted database attributes. This observation suggests that several vulnerabilities within the same application can be prevented by a single fix that blocks the first phase of the attack (e.g., by employing some kind of Turing test).

Finally, we observe that the severity of the uncovered vulnerability is proportional not only to the number of database entries inserted in the first phase of the attack but also to the *kind* of sink encountered in the second phase. In particular, $\Omega(n)$ computations that involve network operations, file I/O or database manipulation typically lead to more serious vulnerabilities. For example, one of the vulnerabilities in osCommerce involves sending emails to all registered users, which can lead to the collapse of the server's network for hours.

Discussion of false positives. We now discuss the root causes of the false positives reported by Torpedo. Manual inspection of all false alarms reveals that there are only two root causes: (1) incomplete sanitizer inference, and (2) incomplete database query resolution. In particular, all 11 false positives in HotCRP are due to missed detection of sanitizers, and the remaining 7 false alarms in osCommerce and WordPress are caused by imprecise string analysis used for database schema detection. Torpedo fails to identify some of the sanitizers in HotCRP because the constraints generated for sanitizer inference are overapproximate: Since Torpedo heuristically drops some interprocedural path conditions for scalability reasons, the SMT solver may decide that the overapproximate constraint is satisfiable even though the exact constraint is in fact unsatisfiable. We note that all false positives in HotCRP can be eliminated using a few simple annotations that explicitly identify sanitizers missed by Torpedo. Incomplete database query resolution occurs when Torpedois unable to identify the string values of the database name and/or attributes in a dynamic database query, and this over-approximation results in false positives. We believe the remaining 8 false positives in osCommerce and Wordpress can be eliminated by employing a more precise string analysis for inferring the tables and attributes of database queries.

Lasting Damage to Applications. While the results in Table 2 focus on the damage of a specific second-phase attack, the impact of the first phase can often be much more significant: Once the database has been polluted, the application is often primed for multiple possible second-phase attacks, becoming virtually unusable. For example, imagine a conference submission site whose database has been populated with an enormous number n of spurious papers. This database is unlikely to be useful for the review process because the conference chair would have to be careful to not trigger the high-complexity behavior in the application. Alternatively, the conference chair could try to cleanse the database. In OpenConf, the program chair might try to flag for withdrawal all submissions with no uploaded files. This natural reaction to the attack unfortunately triggers the second phase of the attack. In general, a careful attacker can perform a dictionary attack that makes it difficult to distinguish malicious from benign entries, complicating the task of automatically cleansing the tables without removing legitimate entries.

In the remainder of this section, we describe two representative vulnerabilities uncovered by TORPEDO and outline how an attacker can exploit these vulnerabilities to launch a second-order DoS attack.

7.1 Exploit in HotCRP

One of the vulnerabilities uncovered by TORPEDO is in the HotCRP conference management application. This vulnerability arises due to an interaction between three HotCRP features, namely account creation, paper registration, and merging of accounts.

To understand the vulnerability and how it can be exploited, we first observe that HotCRP allows a registered user to add an *unrestricted* number of papers to an underlying database called `Paper`. Furthermore, it is possible, although not trivial, to automatically pollute this `Paper` database by ensuring that certain conditions are met (for example, `$_REQUEST`["p"] is set to "new", `$_POST`["submitfinal"] and `$_POST`["submitpaper"] are both set to "1", the hidden *formid* value, which is actually leaked from the cookie, is valid, and so on).

Second, let us consider the HotCRP functionality that allows users to merge different accounts, shown in Figure 9. This merge operation first retrieves the set S of all papers associated with one account, and then, for each paper in S, it performs an update operation on the `Paper` database to modify the corresponding author information. Thus, when merging an account A with another account B, the amount of work that is performed is directly proportional to the number of papers associated with A's account.

Thus, to launch a DoS attack on HotCRP, an attacker can implement a bot that performs the following steps:

1. It registers a large number m of bogus user accounts.[4]

2. For each user account, it then registers a large number n of papers by providing inputs that pass the paper registration filters.

[4]Even though HotCRP disallows the registration of multiple accounts associated with the same email address, the attacker can still generate arbitrarily many HotCRP accounts because some email services, including Yahoo, do not prevent bots from creating accounts.

```
...
if ($Me->is_empty())
    $Me->escape();
...
if (isset($_REQUEST["merge"]) && check_post()) {
    if (!$_REQUEST["email"])
        ...
    else if (!$_REQUEST["password"])
        ...
    else {
     $MiniMe = Contact::find_by_email($_REQUEST["email"]);
       if (!$MiniMe)
          ...
     else if
       (!$MiniMe->check_password($_REQUEST["password"]))
          ...
        else if ($MiniMe->contactId == $Me->contactId) {
          ...
        } else if (!$MiniMe->contactId || !$Me->contactId)
          ...
        else {
            ...
            $result = $Conf->qe("select paperId,
            authorInformation
            from Paper where authorInformation like '%\t" .
            sqlq_for_like($MiniMe->email) . "\t%'");
            $qs = array();
            while (($row = edb_row($result))) {
             $row[1] = str_ireplace("\t"
                        . $MiniMe->email .
                       "\t", "\t" .
                       $Me->email . "\t", $row[1]);
             $qs[] ="update Paper set authorInformation='"
               .sqlq($row[1]) . "' where paperId=$row[0]";
            }
           ...
```

Figure 8: Code snippet showing merging of user accounts functionality in HotCRP.

3. Finally, it merges account i with account $i+1$ for each $i \in [1, m-1]$, again by generating inputs that pass HotCRP's checks that (unsuccessfully) attempt to prevent illicit account merging.

Observe that this attack causes the server to perform work whose complexity is $\Theta(m \times n)$, where m is the number of accounts and n is the number of papers per account. Furthermore, since m and n can both be made arbitrarily large, this attack can feasibly cause HotCRP to become unavailable for significant periods of time. For example, when $m = 5$ and $n = 25,000$, the bot we created was able to bring down HotCRP for more than an hour on our local server.[5]

7.2 Exploit in osCommerce

We now describe a vulnerability uncovered by TORPEDO in the osCommerce application. Since osCommerce provides infrastructure for online stores, DoS attacks involving osCommerce directly cost money to businesses that use this application. An interesting aspect of the vulnerability found in osCommerce is that the second phase of the attack is only *indirectly* triggered by the attacker. Thus, the point of this example is to illustrate that second-order DoS attacks can be serious even if the second phase is not under the direct control of the attacker.

[5]For the HotCRP results in Table 2, we use $m = 1$.

```
if (isset($HTTP_POST_VARS['action'])
    && ($HTTP_POST_VARS['action'] == 'process')
    && isset($HTTP_POST_VARS['formid'])
 && ($HTTP_POST_VARS['formid'] == $sessiontoken)) {

 ...

  if (strlen($firstname)
    < ENTRY_FIRST_NAME_MIN_LENGTH) {
    $error = true;

    $messageStack->add('create_account',
         ENTRY_FIRST_NAME_ERROR);
  }
  ...

  if (strlen($email_address) <
      ENTRY_EMAIL_ADDRESS_MIN_LENGTH) {
    $error = true;

    $messageStack->add('create_account',
       ENTRY_EMAIL_ADDRESS_ERROR);
  }
  ...
  } else {
    $check_email_query = tep_db_query(
        "select count(*) as total from "
           . TABLE_CUSTOMERS .
          " where customers_email_address = '" .
         tep_db_input($email_address) . "'");
    $check_email = tep_db_fetch_array
         ($check_email_query);
    if ($check_email['total'] > 0) {
      $error = true;
      $messageStack->add('create_account',
           ENTRY_EMAIL_ADDRESS_ERROR_EXISTS);
    }
  }
  ...
  if ($error == false) {
    ...
    tep_db_perform(TABLE_CUSTOMERS, $sql_data_array);

  ...
```

Figure 9: Code snippet showing user registration functionality in osCommerce

The vulnerability in osCommerce arises from an interaction between two features, namely account creation and email subscription. Let us first consider account creation. The key point here is that osCommerce does not employ a mechanism that prevents bots from registering spurious user accounts. As shown in the code snippet of Figure 9, osCommerce uses a database relation called `TABLE_CUSTOMERS` that stores information about all registered users. When a user fills out an HTML form to create a new account, the code performs checks to ensure that certain conditions are met, for instance, that the customer's first name and email address are a certain minimum length, that there are no other users with the same email address, etc. However, these checks are not sufficient to rule out the possibility that the web form is being filled out by a bot. In fact, the application does not even check that the user has entered a valid (i.e., existing) email address. As a result, it is fairly straightforward to create a bot that registers many spurious users and pollutes the `TABLE_CUSTOMERS` database.

The next feature relevant to the attack is an email subscription feature that notifies users when certain events occur. For example, a user can subscribe to a *product category*, which notifies the user when a new product is added to that category (e.g., electronics or groceries). Similarly, users can subscribe to individual products so that osComerce can notify them of changes to the inventory (e.g., when a certain product is re-stocked). Since the code implementing this subscription feature does not protect itself against bots, it is possible for an attacker to subscribe a huge number of spurious accounts to all possible categories and products. Hence, each time there is a minor change in the inventory, the application will send an enormous number of email messages to all of the spurious users registered by the attacker.

At first sight, this exploit may seem insignificant because the amount of work performed upon each inventory change is only *linear* in the number of subscribed users. However, when we perform experiments to evaluate the impact of this kind of attack, we find that we can bring down our own server for over 10 minutes by registering only 40,000 users and subscribing them to a product. If the website becomes unavailable for over 10 minutes every time there is a minor change to the inventory, customers are unlikely to enjoy their online shopping experience. Thus, even this innocuous-looking attack makes websites based on osCommerce quite unusable for all practical purposes.

Moreover, we find that after polluting the database, the application's administrator becomes essentially helpless, as many of the application's administrative features become unusable due to their long setup or execution times. The panel for displaying user profiles becomes too slow to use. Sending a general newsletter or new product announcements can take hours or even collapse the network, preventing the legitimate users from receiving the message. Personalized emails to specific users can become almost impossible to send due to the need to navigate a vast amount of junk entries. Of course, the administrator likely has no idea which operations have been affected by the attack, further adding to his despair. Unfortunately, as mentioned earlier, it can be extremely difficult to automatically cleanse the database in the presence of a sophisticated first-phase attack.

8. RELATED WORK

We now place our work in the context of prior work, starting with DoS attacks and then considering static analyses for other security purposes.

Defending Against Network-Based DoS Attacks.

Most mechanisms for defending against DoS attacks are deployed at the network level to monitor network behavior, identify anomalous traffic, and set up firewalls that block the attack [11, 1, 27, 21, 20, 29, 17, 8, 15, 32, 28, 23]. Although these techniques are capable of preventing some DoS (and DDoS) attacks, they are limited to a specific underlying model of anomalous traffic and can suffer from scalability problems. Furthermore, they sometimes raise false alarms that prevent legitimate users from accessing the application. In general, distinguishing between DoS attacks and sudden high-volume user traffic (flash crowds) is an open problem [13]. A more detailed survey of network-layer DoS prevention mechanisms can be found elsewhere [19, 2].

More importantly, network-based defense measures are only activated while the attack is in progress, and they are

incapable of diagnosing application-specific performance issues that can be exploited by low-bandwidth attacks, including the second-order DoS attacks considered in this paper.

Defending Against Application-Level DoS Attacks.

Typical application-level DoS vulnerabilities cause the software to crash or become unresponsive. Particularly dangerous are the cases where a small amount of data sent by an attacker can cause the application to shut down (*ping of death* [5, 14]), enter an infinite loop, or trigger a super-linear recursion call-stack (*inputs of coma* [6, 26]).

Dynamic analyses for the detection of application-level DoS vulnerabilities try to generate inputs that either exhibit worst-case execution times or enter non-terminating loops [4, 3, 12]. Dynamic analyses can be difficult to scale to large programs and cannot always generate inputs that uncover such worst-case behaviors.

Static analyses have been used to identify high-complexity code whose behavior is dependent on user input and can thus be manipulated by an attacker [6, 26]. Such analyses have been able to uncover some simple classes of algorithmic complexity attacks, but they cannot automatically identify more subtle cases of algorithmic complexity vulnerabilities, such as improperly implemented hash functions [9]. In particular, detection of such vulnerabilities requires knowledge about input distributions, which is hard to reason about statically.

Recent work on static analysis has focused on extreme cases of DoS vulnerabilities: detection of super-linear recursion call-stacks [6] and infinite loops [26]. Our work detects DoS vulnerabilities that stem from polynomial loops that can be manipulated by the attacker, instead of the extreme cases of superlinear recursion and infinite loops. More importantly, our work focuses on high resource usage behavior that is triggered by certain kinds of database queries and where the query result is controlled by the attacker.

Other Static Analysis-Based Defenses.

Second-order vulnerabilities involving SQL injections (SQLI) and cross-site scripting (XSS) have been known for some time, but a static analysis for detecting these attacks has only recently been proposed [10]. Our work is inspired in part by this recent work and shares a similar overall framework that consists of two connected taint analyses. However, since we target DoS vulnerabilities rather than XSS and SQLI, our notions of taint and sanitization are different; thus, the details of the detection algorithms also differ substantially. First, our analysis must detect multiple potential insertions into database tables and analyze their performance impact. Second, our analysis needs to differentiate between full and conditional sanitizers, whereas conditional sanitization is not relevant in the context of XSS and SQLI vulnerabilities. Third, unlike the method of Dahse and Holz, our technique must perform automated inference to identify conditional sanitizers. Finally, another contribution of this paper over previous work is a novel symbolic execution algorithm for generating candidate attack vectors.

Xie and Aiken [31] implement a symbolic execution algorithm for SQL injections. The idea is to detect unsanitized variables that reach SQL queries using semantic analysis of path constraints. Their symbolic execution engine is similar to ours, but we must solve the additional problem of relating insertions to extractions. In addition, our engine also generates the attack vectors.

Livshits and Lam describe a flow-analysis for detecting XSS and SQL injection vulnerabilities in Java [18]. Wassermann and Su use static analysis to identify XSS vulnerabilities on code using weak sanitization [30]. These approaches tackle a different security vulnerability and do not reason about database interactions.

Automatic Generation of Attack Vectors.

Kiezun et al. [16] describe a method of generating attack vectors for XSS attacks and SQL injections. Specifically, they use dynamic symbolic execution to generate concrete inputs that trigger execution paths involving sensitive nodes. Sen et al. [24] and Chaudhuri & Foster [7] use similar dynamic symbolic execution methods for Javascript and Ruby-on-Rails respectively. Since our approach is static, it has the potential to report more false positives, but a key advantage is that it does not depend on an input-generation mechanism or a mutation library.

9. CONCLUSIONS

In this paper, we have introduced the notion of second-order DoS attacks and presented static analyses (1) for detecting their underlying application-level vulnerabilities and (2) for generating attack vectors to exploit these vulnerabilities. We have used these techniques to detect 37 security vulnerabilities across six open-source web applications, while admitting 18 false positives. Our experiments show that these vulnerabilities can be exploited by performing a tractable number of insertions and can render the application unresponsive for hours.

More broadly, this work expands the threat model: Whereas taintedness traditionally is concerned with specific user input values, second-order DoS attacks are also concerned with tainted database entries whose presence might lead to some expensive future operation. In general, as threat models continue to expand, so does the motivation for more sophisticated static analyses that can defend against new attacks.

Acknowledgments.

This work was funded in part by AFRL Award FA8750-15-2-0096 and NSF grants CNS-1138506 and DRL-1441009. We thank the anonymous referees, Will Robertson, and Tom Dillig for helpful comments on earlier drafts.

10. REFERENCES

[1] S. Abdelsayed, D. Glismsholt, C. Leckie, S. Ryan, and S. Shami. An efficient filter for denial-of-service bandwidth attacks. In *Global Telecommunications Conference (GLOBECOM)*, pages 1353–1357. IEEE, 2003.

[2] M. Abliz. Internet denial of service attacks and defense mechanisms. In *Technical Report No. TR-11-178*, pages 1–50. University of Pittsburgh, 2011.

[3] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE*, 2009.

[4] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *ICSE*, 2009.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[6] R. Chang, G. Jiang, F. Ivančić, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *22nd Computer Security Foundations Symposium (CSF)*, pages 186–199. IEEE, July 2009.

[7] A. Chaudhuri and J. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 585–594. ACM, 2010.

[8] S. cookies. `http://cr.yp.to/syncookies.html`.

[9] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security*, 2003.

[10] D. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. In *23rd USENIX Security Symposium*, pages 989–1003, Aug. 2014.

[11] T. M. Gil and M. Poletto. Multops: A data-structure for bandwidth attack detection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 3–3, 2001.

[12] A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Proving non-termination. In *POPL*, 2008.

[13] M. Handley, E. Rescorla, and IAB. Internet denial-of-service considerations. In *RFC 4732*, 2006.

[14] http://insecure.org/sploits/ping-o death.html. Ping of death.

[15] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *NSDI*, 2005.

[16] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *31st International Conference on Software Engineering (ICSE)*, pages 199–209, May 2009.

[17] R. Kompella, S. Singh, and G. Varghese. On scalable attack detection in the network. In *Proceedings of Internet Measurement Conference (SIGCOMM)*. ACM, 2004.

[18] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, 2005.

[19] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher. *Internet of Service: Attack and Defense Mechanisms*. Prentice Hall, 2005.

[20] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In *ICNP*, pages 312–321, 2002.

[21] T. Peng, C. Lecking, and K. Ramamohanarao. Proactively detecting distributed denial of service attacks using source ip address monitoring. In *Networking*, pages 771–782. Springer-Verlag, 2004.

[22] PHP-Parser. `https://github.com/nikic/PHP-Parser`.

[23] V. Sekar, N. Duffield, K. van der Merwe, O. Spatscheck, and H. Zhang. Lads: Large-scale automated DDoS detection system. In *USENIX*, 2006.

[24] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498. ACM, 2013.

[25] M. share of PHP-based websites. `http://w3techs.com/technologies/details/pl-php/all/all`.

[26] S. Song and V. Shmatikov. Saferphp: Finding semantic vulnerabilities in php applications. In *6th Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, November 2011.

[27] R. Talpade, G. Kim, and S. Khurana. Nomad: Traffic-based network monitoring framework for anomaly detection. In *International Symposium on Computers and Communications*, pages 442–451. IEEE, 1999.

[28] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *SIGCOMM*, 2006.

[29] H. Wang, D. Zhang, and K. Shin. Detecting syn flooding attacks. In *21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1530–1539. IEEE, 2002.

[30] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[31] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, 2006.

[32] X. Yang, D. Wetherall, and T. T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, 2005.

[33] Z3. `https://github.com/Z3Prover/z3`.

[34] Z3-str2. `https://sites.google.com/site/z3strsolver/`.