Detecting Logic Vulnerabilities in E-Commerce Applications

Fangqi Sun Liang Xu Zhendong Su

University of California, Davis

{fqsun, leoxu, su}@ucdavis.edu

***Abstract*—E-commerce has become a thriving business model. With easy access to various tools and third-party cashiers, it is straightforward to create and launch e-commerce web applications. However, it remains difficult to create *secure* ones. While third-party cashiers help bridge the gap of trustiness between merchants and customers, the involvement of cashiers as a new party complicates logic flows of checkout processes. Even a small loophole in a checkout process may lead to financial loss of merchants, thus logic vulnerabilities pose serious threats to the security of e-commerce applications. Performing manual code reviews is challenging because of the diversity of logic flows and the sophistication of checkout processes. Consequently, it is important to develop automated detection techniques.**

This paper proposes the first *static detection* of logic vulnerabilities in e-commerce web applications. The main difficulty of automated detection is the lack of a general and precise notion of *correct payment logic*. Our key insight is that secure checkout processes share a *common invariant*: A checkout process is secure when it guarantees the *integrity* and *authenticity* of critical payment status (order ID, order total, merchant ID and currency). Our approach combines symbolic execution and taint analysis to detect violations of the invariant by tracking tainted payment status and analyzing critical logic flows among merchants, cashiers and users. We have implemented a symbolic execution framework for PHP. In our evaluation of 22 unique payment modules, our tool detected 12 logic vulnerabilities, 11 of which are new. We have also performed successful proof-of-concept experiments on live websites to confirm our findings.

I. Introduction

E-commerce web applications, a special type of web applications designed for online shopping, play an important role in the modern world. The U.S. Census Bureau of the Department of Commerce estimated that U.S. retail e-commerce sales for the second quarter of 2013 reached \$64.8 billion, an 18.4% increase from the previous year [28]. The prevalence of Internet and the rise of smart mobile devices contribute to the rapid growth of e-commerce web applications. Unfortunately, the complexity of e-commerce applications and the diversity of third-party cashier APIs make it difficult to implement perfectly secure checkout processes. Since logic attacks are tied directly to financial loss and merchant embarrassment, the impact of logic vulnerabilities in e-commerce applications is often severe.

NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23351

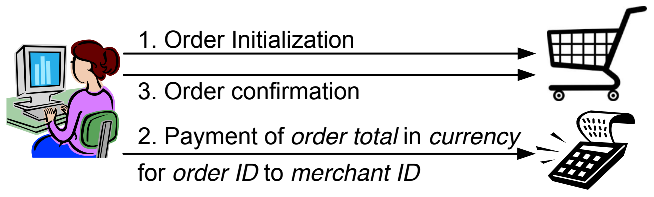
1. Order Initialization
3. Order confirmation
2. Payment of *order total* in *currency*
for *order ID* to *merchant ID*

Figure 1: Logic Flows in E-Commerce Web Applications.

Business or application logic refers to *application-specific* functionality and behavior. Besides general functionality (such as user authentication), each application has its unique handling of user inputs, user actions and communications with third-party components. Although logic vulnerability is not the most common type of web vulnerabilities, it often has serious impact and is easily exploitable. A logic vulnerability typically exists when an attacker abuses legitimate application-specific functionality against developers' intentions [10]. A report by WhiteHat Security lists seven examples of logic flaws [16]. When building an application, developers often have a clear picture of what the ideal application should be in their minds. Unfortunately, in practice, the implemented application often does more than what is intended. Put it another way, unexpected user inputs and logic flows can allow attackers to abuse insufficiently guarded application-specific functionality in dangerous ways. The uniqueness and complexity of logic flows complicate the establishment of a general line of defense against application-specific attacks.

Logic vulnerabilities in e-commerce applications, being a *subset* of general logic vulnerabilities, allow attackers to purchase products or services with incorrect or no payment at the expenses of merchants. Developers often make assumptions about what user inputs are and how users navigate web pages during checkout. However, when such assumptions do not hold and developers fail to implement proper security checks, attackers can exploit logic vulnerabilities in e-commerce applications for financial gains. CVE-2009-2039 [9] describes our motivating example where Luottokunta (version 1.2), a payment module in the osCommerce software [1], has a logic vulnerability that allowed attackers to tamper with order ID, order total and merchant ID. The latest version of Luottokunta (version 1.3) was released to patch this vulnerability by adding logic checks on some components of payment status. However, upon close examination, we were surprised to discover that it

Figure 2: Received Products from Vulnerable Websites.

is still vulnerable. The added check on order ID is insufficient, thus attackers can pay for one order and bypass payments for future orders. This is one of the new vulnerabilities that we detected.

The use of third-party cashiers in e-commerce applications introduces new security concerns even if the cashiers themselves are secure. For flexibility, a modern web application often presents several payment options during checkout by using one payment module for each third-party cashier. However, the integration of cashiers also increases the complexity of logic flows in checkout processes. Figure 1 illustrates three critical steps in a typical checkout process that involves a merchant, a cashier and a user: 1) order initiation on the merchant's server, 2) payment transaction on the cashier's server, and 3) order confirmation on the merchant's server. In the first step, the merchant initiates the basic payment information of an order. From then on, both the merchant and the cashier track the payment status of the order. Ideally, the merchant should either explicitly check *every* component of important payment status or directly communicate with the cashier. In practice, miscommunications between the merchant and the cashier may harm the integrity or authenticity of payment status. Insufficient or missing logic checks on payment status can allow an attacker to skip the second step or carry it out incorrectly. In a successful attack, the merchant is led to believe mistakenly that the order has been paid in full, while the cashier actually receives no payment or partial payment.

With the goal of confirming the real dangers that logic vulnerabilities in e-commerce applications pose, we designed responsible proof-of-concept experiments following the example set by Wang *et al.* [30]. Each experiment was performed on a live website that used a vulnerable payment module. Specifically, we received three products (Figure 2) from three websites which integrate vulnerable payment modules. First, for payment module RBS WorldPay, we received a Ubuntu notebook from the Ubuntu online shop by Canonical Ltd. We paid less by changing the currency from British pounds to U.S. dollars. Second, for payment module Authorize.net Credit Card SIM, we received a diaper game package from a baby products online shop. We paid nothing by replaying tokens from a previous order. Third, for payment module PayPal Standard, we received three chocolate pieces from a California chocolate online shop. We paid nothing to the merchant by changing the merchant ID from the chocolate shop owner's ID to our ID. After having received the products, we immediately compensated the three merchants for the respective correct full amounts. These experiments clearly demonstrate that insecure uses of third-party cashiers, such as the heavily vetted cashier PayPal, may give merchants a false sense of protection.

The detection of logic vulnerabilities in e-commerce applications is challenging for both manual and automated analyses since any weak link in a checkout chain can result in a logic vulnerability. On one hand, manual code review is time-consuming and error-prone. Security analysts often spend much time understanding different logic flows in an e-commerce application before examining security checks of payment status. In contrast, payment module developers are familiar with logic flows but not various attack vectors. In either case, a thorough manual code review of all possible logic flows in a checkout process is a nontrivial task. On the other hand, automatic code scanners cannot detect logic vulnerabilities without the knowledge of application-specific business context. E-commerce applications have various application-specific logic flows and each payment method has its unique APIs and security checks. Consequently, it is challenging to create general rules to automate the detection process.

Researchers have proposed various techniques to detect different logic vulnerabilities, including abnormal logic behavior [15], multi-module vulnerabilities [3] and single sign-on vulnerabilities [31, 33]. Each technique targets a particular domain of logic vulnerabilities and checks web applications against specifications in the given domain. Wang *et al.* [30, 33] are the first to perform security analysis on Cashier-as-a-Service based e-commerce applications. They found several serious logic vulnerabilities in a few popular e-commerce applications via manual code reviews [30] and proposed a proxy-based approach to dynamically secure third-party web service integrations which include the integration of cashiers [33].

In this paper, we propose the first *static detection* of logic vulnerabilities in e-commerce applications. Our key observation is that an invariant must be verified to secure a payment: A merchant M should accept an order O from a user if and only if the user has actually made a payment to the cashier in the correct amount and currency for that specific order O associated with merchant M. Based on this observation, we designed a symbolic execution framework that explores critical control flows exhaustively, tracking taint annotations for the critical components of payment status (order ID, order total, merchant ID and currency) and exposed signed tokens. Our main contributions are:

- We provide an application-independent invariant for detecting logic vulnerabilities in e-commerce web applications and discover a new attack vector: tampering with currency.
- We propose the first static analysis to detect logic vulnerabilities in e-commerce applications based on symbolic execution and taint tracking of payment status.
- We implement a scalable symbolic execution framework for PHP web applications. Our analyzer systematically explores control flows to examine logic flows in checkout processes.
- We evaluate our tool on 22 unique real-world payment

modules from various cashiers and detect logic vulnerabilities in 12 out of the 22 payment modules. We also perform responsible proof-of-concept experiments on live websites. Of the 12 detected vulnerabilities, 11 are new. The evaluation results demonstrate that our approach is effective and scalable.

The rest of the paper is organized as follows. We first give an example to illustrate the main steps of our approach (Section II). Section III describes our detailed algorithm and approach. Section IV presents the implementation of the automated analyzer we developed, and Section V shows the vulnerability report, the details of our experiments on live websites and the performance of our tool on real-world e-commerce payment modules. Finally, we survey related work (Section VI) and conclude (Section VII).

II. Illustrative Example

This section uses payment module Luottokunta (version 1.3) to illustrate the major steps of our approach. This module, which patched the vulnerability described in CVE-2009-2039 [9], is still vulnerable because of an insufficient check on untrusted order ID. During checkout, a user sends out the following four critical HTTP requests, the last two of which are redirections from HTTP responses with a status code of 302:

```
R1. User > Merchant(checkout_confirmation.php)
R2. User > Cashier(https://dmp2.luottokunta.fi)
R3. User > Merchant(checkout_process.php), 302
R4. User > Merchant(checkout_success.php), 302
```

With this payment module, a merchant can integrate the service of third-party cashier Luottokunta. Of the four requests, the second one is sent to the cashier and the rest are sent to the merchant. The first request (R1) initializes the checkout process for an order when the user navigates to page `checkout_confirmation.php`. The second request (R2) lets the user pass on the order information generated by the merchant to the cashier. After the user has completed the payment transaction on the cashier's server, the cashier sends the user a response that redirects the user to page `checkout_process.php` (R3) on the merchant's server to process the order. If the order is accepted, the merchant redirects the user to page `checkout_success.php` (R4).

Our symbolic execution starts from the first merchant page `checkout_confirmation.php` in the checkout process. To model the first request (R1), it symbolically executes the intermediate representation (IR) of this page and simultaneously parses its symbolic HTML output in search of critical HTML form elements. The analysis eventually finds an HTTP form that serves as a communication channel between the merchant and the cashier. Its elements record the order information and its action URL points to the cashier's URL (`https://dmp2.luottokunta.fi`). This form also contains a return URL (`checkout_process.php`), which will be used by the cashier to redirect the user back to the merchant's server once a payment transaction has been completed.

Since our analysis treats cashiers as black boxes that work correctly, we assume that the cashier would properly complete the payment transaction with the user (R2) and redirect the user back to the merchant (R3). To continue exploring logic flows, our analyzer symbolically executes page `checkout_process.php` which is a part of the return URL. A thorough examination requires the modeling of all possible responses from the cashier. Therefore, we use the symbolic *top* value ($\top$), *i.e.*, the most conservative value that denotes any possible value, for the request variables of R3. Our analyzer first propagates the end execution states from the previous page `checkout_confirmation.php` to the current page `checkout_process.php`, and then symbolically executes the IR of the current page. The execution eventually reaches function `before_process()` which has the following checks on payment status:

```
function before_process() {
  if (!isset($_GET['orderID'])) {
    tep_redirect(FILE_PAYMENT);
  } else {
    $orderID = $_GET['orderID'];
  }

  $price = $_SESSION['order']->info['total'];
  $tarkiste = SECRET_KEY
    . $price . $orderID . MERCHANT_ID;
  $mac = strtoupper(md5($tarkiste));

  if (($_POST['LKMAC'] != $mac)
      && ($_GET['LKMAC'] != $mac)) {
    tep_redirect(FILE_PAYMENT);
  }
}
```

Because request variable `$_GET['orderId']` has a symbolic top value, both branches of the first `if` statement are feasible. For the `true` branch, the user is redirected to merchant page `FILE_PAYMENT`. This redirection forms a backward flow, which does not contribute to the detection of logic vulnerabilities. Therefore, this backward logic flow is automatically discarded. For the `false` branch, an `MD5` value is calculated and stored in variable `$mac`. Note that the value of `$orderID` used in the calculation comes from an untrusted request variable `$_GET['orderId']` which is under attackers' control.

Our taint analysis tracks the components of critical payment status across logic flows in the checkout process. Initially, order ID, order total, merchant ID, currency and secret are all tainted. *Secret* refers to an unpredictable value that only the merchant and the cashier know. Therefore, the cashier can use it to sign messages. For taint manipulation, we have a set of rules. One rule removes a taint annotation when a conditional check verifies an untrusted value against a trusted component. For the last conditional in function `before_process()`, we have the following symbolic constraints for the `false` branch:

```
[ or
  ($_POST['LKMAC'] = strtoupper(md5(SECRET_KEY
    . $_SESSION['order']->info['total']
    . $_GET['orderID'] . MERCHANT_ID)));
  ($_GET['LKMAC'] = strtoupper(md5(SECRET_KEY
    . $_SESSION['order']->info['total']
    . $_GET['orderID'] . MERCHANT_ID)));
]
```

Among the symbolic values in the above constraints, `$_SESSION['order']->info['total']` is a trusted session value, while MERCHANT_ID and SECRET_KEY are trusted constants defined in the merchant's database. This conditional check guarantees that the cashier has received a payment in full on behalf of the merchant. Therefore, our analyzer removes the taint annotations of order total, merchant ID and secret. In contrast, $_GET['orderId'] is an untrusted request variable, and there is no check for currency.

After exploring before_process(), the symbolic execution eventually redirects the user to the final page checkout_success.php (R4). When the symbolic execution reaches this page, it means that the checkout process is complete and our analysis generates a final vulnerability report. In the report of this payment module, order ID and currency are still tainted, indicating that this module is vulnerable to two types of logic attacks. The first type of attacks allows an attacker to pay for one order and avoid payments of future orders by replaying the value of $_POST['LKMAC'] or $_GET['LKMAC'] of the paid order. Note that the attacker can easily intercept the value of $_POST['LKMAC'] or $_GET['LKMAC'] of any paid order by changing the return URL to her own choice in R2. The second type of attacks allows an attacker to pay less by paying the cashier the correct amount indicated by $_SESSION['order']->info['total'], but in a different currency. For example, the merchant might list product prices in European euros, but an attacker can pay in U.S. dollars instead. [1]

To illustrate the first type of logic attacks on order ID, suppose a user places two orders with IDs of 1001 and 1002 on a merchant's server. For either order, the designated cashier is assumed to generate a secret MD5 value and visit a URL of page checkout_process.php with the secret value only when a full payment has been made to the cashier. The secret values of orders 1001 and 1002 should be different and unpredictable. The URLs for the two orders are shown in the following.

```
URL1: http://merchant.com/checkout_process.php?
  orderID=1001&LKMAC=SecretMD5For1001

URL2: http://merchant.com/checkout_process.php?
  orderID=1002&LKMAC=SecretMD5For1002
```

The key problem for the detected logic vulnerability is that this e-commerce application does not check the request parameter values of orderID and LKMAC against trusted order ID and MD5 value. Suppose an attacker has paid for order 1001 and intercepted the secret value SecretMD5For1001 by changing the server in the return URL (URL1) from merchant.com to attacker.com. For order 1002, rather than making a payment and being redirected to URL2, the attacker can skip payment and jump directly to a forged merchant URL (URL1 shown above). The attacker can use the GET and POST request parameter values of order 1001 for order 1002 to avoid payment. This substitution leads the merchant to mistakenly believe that order 1002 has been paid, while the cashier actually has received nothing on behalf of the merchant for this order. Similarly, this vulnerability allows the attacker to bypass payment for future orders as long as the order total matches the total of order 1001.

[1]This attack can be launched when the cashier accepts multiple currencies for payments.

III. Approach

This section presents our high-level approach. We first define logic vulnerability in e-commerce applications, lay out our assumption, and then describe the core algorithm of our approach.

A. Definitions

Definition 1 (Merchant). A *merchant* accepts an order when it has been properly paid via a third-party cashier by a user. Merchant is the central role in e-commerce applications, coordinating communications between users and cashiers during checkout processes. Merchants are responsible for initializing orders, tracking payment status, recording order details, finalizing orders and shipping products (or providing services) to users.

Definition 2 (Cashier). A third-party *cashier* accepts the payment of an order from a user on behalf of a merchant. Cashiers bridge the gap between merchants and users when they lack mutual trust. Users trust cashiers with their private information, and merchants expect cashiers to correctly charge users.

Definition 3 (User). A *user* initiates a checkout process on a merchant's website, chooses a third-party cashier, makes a payment to the cashier and receives products (or services) from the merchant. User inputs and actions drive the logic flows of checkout processes. Some users are malicious, therefore merchants need to defend against untrusted user inputs and actions.

Definition 4 (Logic Flows in E-commerce Applications). *Logic flows in e-commerce applications* are communications between three possible parties: merchant nodes, cashier nodes and user. Any logic flow during checkout may influence payment status. Note that one merchant web page may be divided into multiple merchant nodes based on the runtime values of its HTTP request variables. For instance, one page may perform an "insert", "update" or "delete" operation depending on the value of $_GET['action']. Our analysis starts at the beginning merchant node n_0 of a checkout process and ends at the destination merchant node n_k where orders are accepted, tracking taint annotations of payment status and signed tokens across logic flows. Suppose for any valid node n_i in the checkout process, we start the analysis of n_i with execution state set Q_i. At the end of the analysis of n_i, we would have execution state set Q_j and a node to be visited next, namely n_j. Formally, logic flows in an e-commerce application can be represented as $\Pi = \{(n_i, Q_i) \rightarrow (n_j, Q_j) \mid 0 \leq i, j \leq k\}$.

Definition 5 (Logic State). A *logic state* consists of taint annotations and links to other valid nodes of a checkout process. The propagation of logic states reflects changes of payment status. Specifically, for any order that a user places on a merchant's website with the integration of a third-party cashier, a logic state stores taint annotations for the following payment status components and exposed signed tokens:

- ***Order ID***. The identifier of the order which should be paid for before it is accepted.
- ***Order total***. The total amount that the cashier should receive from the user on behalf of the merchant.
- ***Merchant ID***. The identifier used by the cashier for the merchant who is selling products or services. The cashier will ultimately transfer the received money from the user to the merchant.
- ***Currency***. The currency (system of money) in which the order payment should be made.
- ***Exposed signed token***. An encrypted value that is signed with a secret between the merchant and the cashier. It can act as a cashier's signature and is considered exposed when it is visible to users in the Document Object Model (DOM) tree of a merchant page.

Definition 6 (Logic Vulnerabilities in E-commerce Applications). A *logic vulnerability in an e-commerce application* exists when for any accepted order ID, the merchant cannot verify that the user has correctly paid the cashier the amount of order total in the expected currency to merchant ID. Our definition is inspired by and developed from an extensive study of cashier documentation, open-source e-commerce applications and related work [30, 33]. A payment is secure when both the integrity and authenticity of payment status are ensured. Tampering with currency is a new attack vector we discovered in our study.

Assumption *Third-party cashiers are secure.* We treat third-party cashiers as black boxes and assume that they are perfectly secure. Most third-party cashiers' source code is unavailable, but many cashiers have been vetted heavily. The security of a third-party cashier is orthogonal to the security of its integration in an e-commerce web application. Developers of payment modules are often less security-conscious than those of cashiers, thus payment modules are generally more prone to logic vulnerabilities.

Based on logic vulnerabilities in e-commerce web applications, it is easy to launch attacks on live websites. Simply using browser extensions, attackers can withhold HTTP requests, modify requests or completely forge requests. Moreover, attackers can exploit a signed token to pose as a cashier, reuse payment information from previous orders or intercept cashiers' responses by changing return URLs in HTTP forms.

In summary, logic vulnerabilities in e-commerce applications are caused by the following five types of taint annotations:

- ***Tainted order ID***. To bypass order payments, attackers can replay the payment information of a previous order from the same merchant. As long as the order total and currency of unpaid orders match the ones of the previously paid order, the unpaid orders would be accepted because order ID is not verified.
- ***Tainted order total***. Attackers can pay an arbitrary amount for an order by tampering with the order total sent to a third-party cashier if order total is not verified. A partial payment to the cashier is still necessary.
- ***Tainted merchant ID***. When merchant ID is tainted, an attacker can set up her own merchant account on the designated cashier's server where the original merchant ID was set up. This allows the attacker to send payments to herself instead of the merchant for orders placed on the merchant's website. Note that a check on the secret between the merchant and the cashier can replace the check on merchant ID because the secret is a unique, verifiable value set by the merchant.
- ***Tainted currency***. For cashiers that accept multiple currencies, it is possible to pay less for orders via the use of a different currency without changing the order total amounts.
- ***Exposed signed token***. An exposed signed token invalidates any security checks against trusted symbolic values. This is because such a signed request may be forged by an attacker rather than coming from a trusted cashier.

B. Automated Analysis

Section III-B1 presents our detection algorithm which explores critical logic flows in e-commerce applications among three parties (merchant, cashier and user). Section III-B2 describes taint manipulation rules which reflect changes to payment status.

1) Logic Vulnerability Detection Algorithm: Figure 3 presents our vulnerability detection algorithm which forms the core of our approach. It integrates symbolic execution of merchant nodes and taint analysis, and connects individual nodes to explore valid logic flows in e-commerce applications. We have four possible pairs of HTTP requests from the client side to the server side: (user, merchant), (user, cashier), (cashier, merchant) and (merchant, cashier). Attackers may skip user-to-cashier requests, but they need to send the same number of user-to-merchant requests to carry out all necessary steps of during checkout. Consequently, our algorithm analyzes merchant nodes that belong to a checkout navigation path in order.

There are three functions in Figure 3 and the first function DETECTVULS is the main function of our analysis algorithm. The second function ANALYZENODE analyzes each merchant node individually, and the third function GETNEXTNODE connects nodes together for valid logic flows. The analysis begins from start node n_s with a start execution state q_s. Both n_s and q_s are extracted from specifications $Spec$. An execution state q contains a logic state, memory maps for global and local variables, alias information, etc. Our algorithm analyzes logic flow $(user, \textsc{Merchant}(n_s))$ first, and continues until all valid logic flows are explored. In the end, for each execution state q_f in the final execution state set Q_f, function CHECKLOGICVULS checks the logic state in q_f and reports any detected logic vulnerabilities $Vuls$.

Function ANALYZENODE recursively analyzes merchant nodes of valid logic flows until the final node n_f is reached. The final execution state set Q_f is only updated when a new final execution state q_f has a uniquely new logic state. The reason behind this update strategy is that other data in an execution

```
DetectVuls($Spec$)
 1  $n_s \leftarrow$ GetStartNode($Spec$)
 2  $q_s \leftarrow$ InitState($Spec$)
 3  $q_s \leftarrow$ AddComm($user$, Merchant($n_s$), $q_s$)
 4  $Q_f \leftarrow$ AnalyzeNode($n_s, q_s, \emptyset, Spec$)
 5  $Vuls \leftarrow$ CheckLogicVuls($Q_f$)
 6  return $Vuls$

AnalyzeNode($n, q, Q_f, Spec$)
 1  $n_f \leftarrow$ GetFinalNode($Spec$)
 2  if $n = n_f$
 3     then $Q_f \leftarrow Q_f \cup \{q_f\}$
 4          return $Q_f$
 5  $q \leftarrow$ PropagateNodeState($n, q$)
 6  $Q \leftarrow$ SymbolicExecution($n, q, Spec$)
 7  for each $q_i$ in $Q$
 8  do $\langle n', q_i \rangle \leftarrow$ GetNextNode($q_i, Spec$)
 9      $Q_f \leftarrow$ AnalyzeNode($n', q_i, Q_f, Spec$)
10  return $Q_f$

GetNextNode($q, Spec$)
 1  $n \leftarrow$ ResetRedirection($q$)
 2  if $n = null$
 3     then $n \leftarrow$ ResetFormAction($q$)
 4  if IsCashier($n, Spec$)
 5     then $q \leftarrow$ AddComm($user$, Cashier($n$), $q$)
 6          $n \leftarrow$ ResetCallbackUrl($q$)
 7          if $n = null$
 8             then $n \leftarrow$ ResetReturnUrl($q$)
 9  $q \leftarrow$ AddComm($user$, Merchant($n$), $q$)
10  return $\langle n, q \rangle$
```

Figure 3: Algorithm for Vulnerability Detection.

state have no impact on the final vulnerability results. Function PropagateNodeState propagates an execution state q from a previous node (n_p) to the current merchant node (n), performing a few operations on q. Specifically, this function updates runtime constants such as `$_SERVER['PHP_SELF']`, updates array `$_GET` based on the query string of node n, updates array `$_POST` based on the form elements of node n_p and resets the memory map of local variables. By default, request variables have the symbolic top value, which represents all possible values including `null`. Next, merchant node n is symbolically executed via function SymbolicExecution, and Q is the end execution state set for n. During symbolic execution, HTML form action URLs, form elements and parameter values of merchant-to-cashier `cURL` [2] requests are monitored in search of links to other merchant nodes or cashier nodes.

To connect nodes of valid logic flows, function GetNextNode examines and resets four types of links: redirection URL, form action URL, callback URL and return URL. A redirection URL or form action URL can point to either a cashier node or a merchant node, while a callback URL or return URL can only point to a merchant node. To navigate only along valid logic flows, we discard URLs that form backward or self-cycle logic flows, and URLs that are irrelevant to checkout. Each reset function within function GetNextNode stores in n the extracted value of a particular type of URL, and resets the URL to `null`. The values of header redirection URL (obtained via function ResetRedirection) and form action URL (obtained via function ResetFormAction) are examined first. When URL n points to a cashier node, a logic flow (user, Cashier(n)) is added. To model cashiers' responses, this function examines the values of callback URL (obtained via function ResetCallbackUrl) and return URL (obtained via function ResetReturnUrl). Note that callback URL and return URL can only be set after a cashier has been visited. Callback URL is optional and can be visited first by a cashier to notify its merchant the completion of a payment transaction. Return URL is required and a user must relay a cashier's response to this URL to confirm a paid order on a merchant's server. The return value of function GetNextNode is a pair of merchant node n that should be visited next and the updated state q.

2) Taint Rules: To keep track of the integrity and authenticity of payment status, we designed a few taint manipulation rules. The integrity of payment status can thwart HTTP parameter tampering attacks, and the authenticity of payment status defends against forged payment status which is coined with predictable or exposed values of request variables. To be more specific, untainted order ID, order total, merchant ID and currency ensure the integrity, while no exposure of signed tokens ensures the authenticity. The underlying assumptions of the taint rules are: 1) requests from users are untrusted; 2) unsigned cashier requests sent via insecure channels are untrusted; and 3) cashier responses that are relayed by users to merchants via HTTP redirection (status code 302) are also untrusted. Initially, order ID, order total, merchant ID and currency are all tainted.

When a merchant correctly verifies a payment status component, the taint annotation of the checked component should be removed. Our approach uses taint removal rules for the following three cases:

- ***Conditional checks***. When an (in)equality conditional check verifies an untrusted value against a trusted symbolic value of a payment status component, remove taint from the checked payment status component.
- ***Writes to merchant database***. When a tainted value is written into a merchant's database with `INSERT` or `UPDATE` queries, conservatively remove taint from the component. Before a merchant employee ships a product or provides a service for an order, she needs to review order details retrieved from database tables. If a modified component is written to database, the merchant employee can easily spot the modified component and thus reject the order with the modified component.
- ***Secure communication channels***. For synchronous merchant-to-cashier `cURL` requests, remove taint for order total, merchant ID or currency when such component are included in `cURL` request parameters, and remove taint for order ID unconditionally. Synchronous requests are sent via secure communication channels,

[2]http://curl.haxx.se/

and thus can guarantee the authenticity of payment status changes that pass through such channels.

Our approach has one taint addition rule: When a conditional check for a cashier-to-merchant request relies on an exposed signed token, add taint to the exposed signed token. We keep track of all signed token values that are disclosed in DOM trees to users (typically in hidden HTTP form elements). Although hidden HTTP form elements are invisible in the presentation layer of HTML pages, attackers can obtain their values by simply viewing the source code of web pages. Note that not all exposed signed tokens are tainted. The taint addition rule only applies when an exposed signed token is used as an unpredictable value in a conditional check for a cashier-to-merchant request. Once a signed token is exposed, it is no longer unpredictable and therefore should not be used in a conditional check. For example, suppose we have a signed token in a hidden HTML form with symbolic value `md5($secret.$orderId.$orderTotal)`. If our analysis encounters equality check `$_GET['hash'] == md5($secret.$_GET['oId'].$_GET['oTotal'])`, it adds taint to the exposed signed token. This is because although `$secret` is unpredictable, the values of the three request variables are predictable. To pass the check, an attacker can use the exposed signed token for `$_GET['hash']`, the order ID and order total associated with the exposed signed token for `$_GET['oId']` and `$_GET['oTotal']` respectively.

IV. Implementation

We developed a symbolic execution framework that integrates taint analysis for PHP, one of the most prevalent languages for building web applications. We extended the PHP lexer and parser of a static string analyzer [23, 27, 32] written in OCaml. Our tool handles object-oriented features of PHP, including classes, objects and method calls. We wrote transfer functions for built-in PHP library functions, which include string functions, database functions, I/O functions, etc. Our tool consults Satisfiability Modulo Theories (SMT) solver Z3 [13] for branch feasibility, supporting arithmetic constraints, simple string constraints and some other types of constraints. Our implementation contains a total of $25,113$ lines of OCaml code. Although our implementation targets the PHP language, the high-level approach is general and applicable to e-commerce software written in other languages.

Figure 4 shows the architecture of our framework. Given the source code of an e-commerce web application and a specification for it, our analysis starts with a single execution state q_s at merchant node n_s, the first node in the checkout process. For each merchant node n_i, our PHP lexer and parser transform the corresponding merchant page into an Abstract Syntax Tree AST_i, which is then transformed into an Internal Representation IR_i by our IR constructor. After the symbolic execution engine explores all possible control flow paths of IR_i, we have a set of end execution states Q_i. Next, the navigator searches for valid logic flows, and continues symbolic execution for new merchant nodes until the final merchant node n_f is reached. Finally, the logic analyzer checks all the unique logic states of final execution state set Q_f, and then reports any detected logic vulnerabilities.

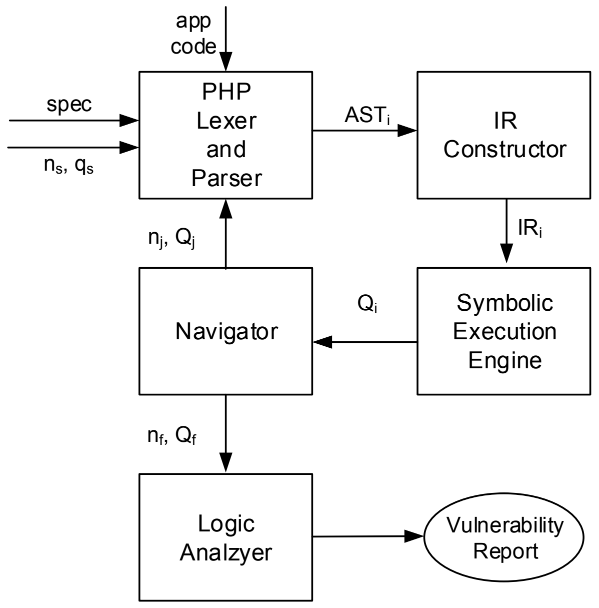

Figure 4: Symbolic Execution Framework.

To guide our automated analysis, we need developers to specify application-specific variable names of payment status components, critical merchant pages in the checkout process, cashier URLs, callback URL, return URL, configurable constants defined in the database and runtime values of a few variables that are used for the resolution of dynamic file inclusion and class construction. For instance, a payment class can be dynamically constructed based on a user's choice of payment methods. If the user chooses PayPal Standard as the payment method, we can specify the value of runtime variable `$_SESSION['payment']` to be "paypal_standard" to precisely resolve the target of class `$payment`.

A. Symbolic Execution

For each merchant page in the checkout process of an e-commerce application, our PHP lexer and parser transform its source code into an IR. We followed the PHP language reference and carefully wrote parsing rules to resolve reduce/reduce conflicts, assigned operator precedence to resolve shift/reduce conflicts and used associativity to resolve other types of conflicts. We observed that a PHP page can either statically or dynamically include other pages via PHP include or `iframe`, and the pages that are included can in turn include other pages. To fully expand a PHP page, our analyzer infers static targets of included pages when possible, and resorts to specification when targets can only be decided at run time. For example, static include `require(DIRS_CLASSES.'cart.php')` depends on the value of constant `DIRS_CLASSES`, while dynamic include `require($language.'.php')` depends on the runtime variable `$language`.

For heap modeling, our tool uses five variable maps: a variable-to-symbolic-value memory map, an instance-to-class-name map, an alias-to-variable map, an array-parent-to-array-elements map and an object-parent-to-object-properties map. First, the variable-to-symbolic-value map allows us to model a heap symbolically. A symbolic value is a recursive data structure composed of the following types: literal, basic symbolic PHP variable, library function call, concatenation of two symbolic values, arithmetic expression, comparison expression and symbolic PHP resource. For instance, symbolic value `md5("hello".$_GET['orderID'])` represents a call to library function `md5` with a symbolic argument of a concatenation of two symbolic values: a string literal "hello" and a basic symbolic variable `$orderID` of type integer. Second, given a class instance and a method name, the instance-to-class-name map enables us to quickly retrieve the corresponding class method definition. Third, the alias-to-variable map allows us to correctly update a symbolic heap. Aliases are created when: a method is called from within an object context (`$this` becomes available); a variable is assigned by reference; and a function/method has pass-by-reference arguments or returns a reference. Last, the two maps for array and object variables enable us to track the children of arrays and objects respectively. Our tool uses one memory map for global variables and one memory map for local variables.

To model arrays and objects in PHP, we adopt the McCarthy rule for list manipulations [13]. Given an array a, an array element e and array index i, let $a[i]$ represent an array select and $a\{i \leftarrow e\}$ represent an array store with the element at index i set to e. By the McCarthy rule, we have the following:

$$\begin{gathered}(\forall\ \texttt{array}\ a)(\forall\ \texttt{element}\ e)(\forall\ \texttt{index}\ i,j)\\ i=j \ \longrightarrow\ a\{i \leftarrow e\}[j]=e\\ \wedge\, i\neq j \ \longrightarrow\ a\{i \leftarrow e\}[j]=a[j]\end{gathered}$$

Our implementation precisely retrieves and updates array elements (or object properties) whenever possible. Otherwise, when an index of an array variable (or the field of an object property) is $\top$, all possible values of the array elements (or object properties) are merged. For example, suppose we have a simple array `$arr=array(1=>"x",2=>"y")`. If the value of array index `$i` is $\top$, the value of `$arr[$i]` is either "x" or "y". We also use the McCarthy rule to symbolically represent arrays and objects. As an example, the symbolic representation of `$arr` is:

$$array\_update(array\_update(array(),1,\text{“}x\text{”}),2,\text{“}y\text{”})$$

B. Path Exploration

Given a start execution state, our goal is to explore all possible intra-procedural and inter-procedural edges in the control-flow graph (CFG) of a merchant node. We use a work-list-based algorithm and explore CFG edges with a depth-first strategy. On one hand, to explore all possible control flows within a function/method body, a work list stores execution states for feasible branches that have not been explored yet. Each execution state includes a program counter (consists of a basic block number and a statement number within the basic block), a logic state, path condition, memory maps of global and local variables, etc. We set a configurable quota for the maximum number of similar execution states in a work list to avoid state explosion. When the quota for a work list is exhausted, we only add an execution state to the work list if it has either a new program counter or a new logic state. On the other hand, to explore all possible inter-procedural edges, our approach adopts a global call stack which stores snapshots of previous function environment before function calls. A function environment snapshot includes a parameter-argument map for the inter-procedural function call which is going to be explored, the work list and end execution states of the current function, etc.

Our tool consults the SMT solver Z3 for constraint solving. When a conditional is encountered during symbolic execution, our analyzer transforms the conditional into a formula of the `smtlib2` format, conjuncts the new formula with the current path condition, and feeds the merged path condition to Z3 to get an answer. When both branches are feasible, we select one branch to explore first, and add the other branch to the current work list. We support the following types in our constraints: boolean, integer, real, string, array, object, resource, `null` and $\top$. We try to infer the satisfiability of simple string constraints, which can contain literals, string variables, and operators such as $=, \neq, <, \leq, >$ and $\geq$. To symbolically represent PHP library function calls, we use `define-fun` in Z3 for function declarations.

```
$error = false;
if ($_POST['x_response_code'] == '1') {
  if (tep_not_null(AUTHORIZENET_MD5_HASH) &&
    ($_POST['x_MD5_Hash'] != strtoupper(
      md5(AUTHORIZENET_MD5_HASH .
      AUTHORIZENET_LOGIN_ID .
      $_POST['x_trans_id'] .
      $this->format_raw($order->info['total'])
    )))) {
    $error = 'verification';
  } elseif ($_POST['x_amount'] !=
      $this->format_raw($order->info['total'])) {
    $error = 'verification';
  }
} elseif ($_POST['x_response_code'] == '2') {
  $error = 'declined';
} else {
  $error = 'general';
}

if ($error != false) {
  tep_redirect(tep_href_link(
    FILENAME_CHECKOUT_PAYMENT,
    'payment_error=' . $this->code .
    '&error=' . $error, 'SSL', true, false));
}
```

Figure 5: Example for Path Exploration.

Consider the example in Figure 5 for path exploration. Since the default values of request variables are $\top$, all possible control-flow edges are explored. Only one exploration path in the example leads to a valid logic flow, while the other paths redirect users to a payment node with an error message in `$error`. For the second `if` conditional on the valid path, there is a method call `$this->format_raw($order->info['total'])`. To follow this inter-procedural edge, our analyzer first looks up the class name of object `$this` and then the definition of method `format_raw` in the corresponding class. Next, the analyzer updates aliases for pass-by-reference parameters which include `$this`, initializes parameter values based on the arguments of the method call, passes on the memory map of global variables and pushes a snapshot of the current function environment into the global call stack. At the end of the symbolic execution for `format_raw`, we have a set of execution states Q. After the method call returns, our analyzer pops the function environment from the global call stack and maps Q to Q' to update method arguments that have pass-by-reference parameters. To continue path exploration after the call, Q' is added to the current work list. When all possible paths are explored, merchant ID and order total are untainted in the execution state of the valid flow which keeps the value of `$error` unchanged.

C. Logic Flows

The focus of our analysis is critical logic flows of a successful checkout process. We discard backward flows, error flows or aborted flows since they are irrelevant to our security analysis. First, a *backward flow* happens when an error has occurred in a merchant node n, and the user is redirected to a previous merchant node or the same merchant node n. Second, an *error flow* refers to a redirection to a special error page or a visited page with an error message in a request variable. In the first case, the special error page does not belong to the critical checkout process and the flow to this page is discarded. In the second case, flows to pages with a symbolic error message variable are backward flows, which are automatically discarded. Last, an *aborted flow* happens when a serious error occurs and the rendering of a merchant page is stopped with an `exit` statement.

In search of links to other nodes, our analyzer parses symbolic values of HTTP forms and `cURL` parameters. Since string literal is not the only type that a symbolic value can represent, we cannot simply use regular expressions such as `<form \s*action\s*=\s*[^>]*>` to extract links. Consequently, our parser recursively examines each component of a symbolic value to correctly handle non-literals. In most cases, merchants embed URLs in HTTP requests to cashiers and our parser can find such URLs. However, a merchant may also store the configurations of callback URL and return URL on a cashier's server. For this case, we need to specify the pre-configured merchant URLs to continue exploring logic flows after a user-to-cashier request.

Requests from cashiers often store critical payment status in their parameters. Although the names of request parameters vary for different cashiers, it is not necessary to associate their names with payment status components unless their values are written to a database. On one hand, when an untrusted request parameter is compared against a trusted payment status component, our tool can infer which payment status component a request parameter is associated with, and apply taint rules for the involved payment status component. For instance, for `$_POST['x_amount'] ==$order->info['total']`, our analyzer removes taint from order total based on the trusted payment status component `$order->info['total']` rather than the untrusted `$_POST['x_amount']`. On the other hand, when untrusted request variables from cashiers are written to a database via `INSERT` or `UPDATE` queries, we need specifications of which payment status components the request variables are associated with. For example, suppose a specification associates `$_GET['v1']` with order ID and `$_GET['v2']` with order total. If these two request parameters are written to a merchant's database, they will be read from the database and displayed clearly to a merchant employee. Since she needs to review order details before accepting an order, she may reject any order with abnormal payment status. Consequently, the taint annotations of order ID and order total should be removed based on the specifications for `$_GET['v1']` and `$_GET['v2']`.

TABLE I: Payment Modules for Cashiers.

Cashier	**Modules**	**Unique**	**Callback**
2Checkout	1	1	N
Authorize.net	2	2	N
ChronoPay	1	1	Y
inpay	1	1	Y
iPayment	3	1	Y
Luottokunta	2	2	N
Moneybookers	23	1	Y
NOCHEX	1	1	N
PayPal	5	5	Y
PayPoint.net	1	1	N
PSiGate	1	1	N
RBS WorldPay	1	1	Y
Sage Pay	3	3	Y
Sofortüberweisung	1	1	Y
Sum	46	22	8

V. Empirical Evaluation

To evaluate the effectiveness and performance of our tool, we performed experiments on osCommerce [1], one of the most popular open-source e-commerce applications. It has a long history of 13 years, powering more than 14,000 registered sites [1]. The latest stable release (version 2.3) of osCommerce contains 987 files with 38,991 lines of PHP code. It supports various third-party cashiers and multiple currencies with different payment modules, which are integrated in the main framework as add-ons. Each payment module provides a payment method that a user can choose during checkout.

In total, We evaluated 46 payment modules, 22 of which have distinct CFGs. There are 928 payment modules for osCommerce, and new payment modules have been actively added since 2003. In addition, payment modules evolve over time. For example, module Luottokunta (version 1.2) was reported to be vulnerable [9], and Luottokunta (version 1.3) was released to patch the reported vulnerability. 46 payment modules are included in osCommerce by default, and 44 of them are developed to integrate third-party cashiers. The two remaining payment modules are irrelevant to our security analysis: One allows merchants to accept cash on delivery, and the other enables merchants to accept mailed money orders. The 44 payment modules that accept online payment have 20 unique CFGs. Modules that differ slightly from one another in terms of variable names and cashier URLs may have identical CFGs. Therefore, we evaluated 20 default payment modules that have unique CFGs as well as the two Luottokunta payment modules. All the experiments are run on a desktop PC with a quad-core CPU (2.40 GHz) and 4GB of RAM.

TABLE II: Logic Vulnerability Analysis Results.

Payment Module	**Tainted / Exposed**					**Safe**
	OrderId	**OrderTotal**	**MerchantId**	**Currency**	**SignedToken**	
2Checkout	✗	✗	✗	✗		✘
Authorize.net Credit Card AIM						✓
Authorize.net Credit Card SIM	✗			✗		✘
ChronoPay	✗	✗	✗	✗	✗	✘
inpay						✓
iPayment (Credit Card)	✗					✘
Luottokunta (v1.2)	✗	✗	✗	✗		✘
Luottokunta (v1.3)	✗			✗		✘
Moneybookers						✓
NOCHEX	✗	✗	✗	✗		✘
PayPal Express						✓
PayPal Pro - Direct Payments						✓
PayPal Pro (Payflow) - Direct Payments						✓
PayPal Pro (Payflow) - Express Checkout						✓
PayPal Standard			✗			✘
PayPoint.net SECPay	✗	✗		✗		✘
PSiGate	✗	✗	✗	✗		✘
RBS WorldPay Hosted				✗	✗	✘
Sage Pay Direct						✓
Sage Pay Form		✗		✗		✘
Sage Pay Server						✓
Sofortüberweisung Direkt				✗		✓*
Total	9	7	6	11	2	9 + 1*

Table I shows payment modules from 14 different cashiers. Column "Modules" shows the number of payment modules that a cashier has, and column "Unique" lists the number of payment modules that have unique CFGs. All the payment modules are in their latest versions except Luottokunta, for which we included two versions with different CFGs. Cashier Moneybookers provides 23 payment modules for various countries and currencies, but we observed that all of them share the same CFG. Therefore, it is sufficient to pick just one Moneybookers module for our security analysis. In contrast, PayPal has 5 payment modules and each of them has a unique CFG.

A. Analysis Results

Table II shows the analysis results for the 22 unique payment modules. Columns under "Tainted/Exposed" show the existence of tainted components of payment status and exposed signed tokens for each module. For these columns, a table cell marked with "✗" means that a payment status component is tainted or a signed token is exposed. The last column "Safe" summarizes the safety of a payment module. When a payment module verifies all the components of payment status and exposes no signed tokens, it is considered safe and marked with "✓"; otherwise, it is marked with "✘".

Table II shows that when a payment module is unsafe, it is often vulnerable to several types of logic attacks on different components of payment status. First, 9 modules fail to correctly verify order ID. This allows attackers to pay once for an order, and reuse the payment status values of the paid order to bypass payment for future orders. Second, 7 modules fail to verify order total, allowing attackers to pay arbitrary amounts. Third, 6 modules fail to verify merchant ID, allowing attackers to pay themselves instead. Note that the verification of secret can replace the verification of merchant ID. Fourth, 11 modules fail to verify currency, making it the most neglected component of payment status. When a cashier is configured to accept only one currency for a merchant, not verifying currency is safe and acceptable. However, we believe that the best practice is to always verify currency so that additional currencies can be easily added in the future. Last, 2 signed tokens are accidentally exposed in plain text, allowing attackers to pose as cashiers. We also tracked exposed secrets in our evaluation. When a secret is exposed, an attacker can arbitrarily forge values for order ID, order total, merchant ID and currency. Fortunately, none of the modules makes such a mistake.

In summary, as shown in the last column of Table II, 9 out of 22 modules are safe; module Sofortüberweisung Direkt is safe when only one currency is accepted; the remaining 12 modules are vulnerable. We expected the patched version of Luottokunta (v1.3) to be safe at first but were surprised to see that it is still vulnerable. This shows the difficulty of writing a perfectly secure payment module. We manually confirmed the detected vulnerabilities on a local deployment of osCommerce, successfully performed responsible experiments on live web stores powered by osCommerce and communicated with the developers of osCommerce about the detected vulnerabilities. We classified the detected logic vulnerabilities into the following categories.

1) Untrusted Request Variables: Payment module developers sometimes make the mistake of checking payment status based on untrusted request variables. Verifying untrusted request variables guarantees neither the integrity nor the authenticity of payment status, but may give developers a false sense of security. Four modules, namely, Authorize.net Credit Card AIM, iPayment (Credit Card), Luottokunta (v1.3) and PayPoint.net SECPay fall into this category. The values of untrusted request variables that pass such insufficient checks may be inconsistent with actual payment status components. For example, module iPayment (Credit Card) performs a check on order ID based on untrusted request variable `$_GET['ret_booknr']` in the following code.

```
$_GET['ret_param_checksum'] !=
md5(MODULE_PAYMENT_IPAYMENT_CC_USER_ID
  . ($this->format_raw($order->info['total'])
  * 100) . $currency
  . $_GET['ret_authcode'] . $_GET['ret_booknr']
  . IPAYMENT_CC_SECRET_HASH_PASSWORD)
```

An attacker could pay once for an order and intercept the cashier-to-merchant request of the paid order by modifying the return URL of the preceding merchant-to-cashier request. For the above example, the attacker needs to record the values of `$_GET['ret_param_checksum']`, `$_GET['ret_authcode']` and `$_GET['ret_booknr']`. For future orders, the attacker can purchase different products and bypass payments as long as the order total and currency are the same as the paid order. Note that `$_GET['ret_param_checksum']` is supposed to be an unpredictable and unique value signed with secret `IPAYMENT_CC_SECRET_HASH_PASSWORD`. However, simply replaying the intercepted values of the three GET variables would allow the attacker to pass the above payment status check. The check in the example is insufficient because the value of order ID in the conditional comes from untrusted `$_GET['ret_booknr']`.

2) Exposed Signed Tokens: An exposed signed token nullifies the verification of payment status. Two modules ChronoPay and RBS WorldPay Hosted expose their signed tokens. Verification based on exposed signed tokens fails to ensure the authenticity of payment status. An attacker could record the values of signed tokens hidden in HTML forms and forge a request to fake a completed payment. The following exposed signed token from the form element `M_hash`, for example, nullifies the verification on order ID, order total and merchant ID (secret `RBSWORLDPAY_HOSTED_MD5_PASSWORD` can also uniquely identify a merchant).

```
tep_draw_hidden_field('M_hash',
  md5(tep_session_id() . $customer_id
    . $order_id . $language
    . number_format($order->info['total'], 2)
    . RBSWORLDPAY_HOSTED_MD5_PASSWORD));
```

Fundamentally, exposed signed tokens are caused by using the same secret for both merchant signature and cashier signature. We observed that a signed token is often exposed when a merchant wishes to use it to authenticate herself to a cashier. A signed token can work both as a merchant signature and a cashier signature for non-cURL HTTP requests. When a signed token is used for both purposes, it is considered exposed if attackers can intercept cashier-to-merchant requests. There are two methods to fix the problem. The first one is to use just one secret but two ways of calculation to make the signed tokens different. For example, by simply changing the orders of the components of payment status in a calculation, we can generate different signed tokens with the same secret. A better method is to use two secrets to avoid exposing important signed tokens. We can use one secret to authenticate a merchant and the other to authenticate a cashier using the same calculation, without worrying about the security of signed tokens.

3) Incomplete Payment Verification: Payment modules sometimes only partially verify the components of payment status. In other words, checks of some components of payment status are missing rather than insufficient. Three modules, namely, Sage Pay Form, Sofortüberweisung Direkt and PayPal Standard belong to this category. Module Sage Pay Form writes partial payment status into the database, but misses checks on order total and currency. Module Sofortüberweisung Direkt does not verify currency and therefore is vulnerable to currency tampering attacks if cashiers are configured to support multiple currencies. Module PayPal Standard misses the check on merchant ID, allowing an attacker to pay herself instead.

4) Missing Payment Verification: Some payment modules are not designed to defend against logic attacks and have no security checks of payment status at all. They could easily become the playground for attackers. The following five payment modules unfortunately fall into this category: ChronoPay, Luottokunta (v1.2), NOCHEX, 2Checkout and PSiGate. Such payment modules should be patched as soon as possible.

B. Experiments on Live Websites

To show the feasibility and ease of attacks based on the detected logic vulnerabilities listed in Table II, we conducted experiments on three live websites in a responsible manner. We consulted lawyers at our university and followed the example of Wang *et al.* in setting up attacker anonymity, purchasing a VISA gift card at a supermarket with cash, and registering accounts on third-party cashiers [30]. The Google Chrome browser with no browser extensions suffices as our attack tool. Although we initially paid nothing or less to the merchants for the three orders we placed, we paid in full amounts to the merchants after we received the products shown in Figure 2. We reported the results of our experiments to osCommerce developers. The details of the experiments are elaborated in the following.

The Ubuntu online shop by Canonical Ltd. (RBS WorldPay Hosted). RBS WorldPay is a cashier mainly used in the U.K. and supports multiple currencies. The Ubuntu online shop is a featured osCommerce shop, and it uses the vulnerable module RBS WorldPay Hosted. As Table II shows, this payment module is vulnerable to currency attacks. We placed an order in U.K. pounds but paid cashier WorldPay in U.S. dollars of the same amount. About one week later, we received a Ubuntu notebook (shown in Figure 2) even though we did not pay the full amount at first.

A baby products online shop (Authorize.net Credit Card SIM). Module Authorize.net Credit Card SIM is vulnerable to order ID attacks. In our experiments on the baby products online shop, we placed two orders of the same order total but

TABLE III: Performance Results.

Payment Module	**Files**	**Nodes (%)**	**Edges (%)**	**Stmts (%)**	**States**	**Flows**	**Time (s)**
2Checkout	105	5,194 (19.09%)	6,176 (19.15%)	8,385 (25.01%)	40	4	16.04
Authorize.net Credit Card AIM	105	5,274 (19.95%)	6,284 (19.96%)	8,545 (25.97%)	43	4	17.65
Authorize.net Credit Card SIM	105	5,221 (19.66%)	6,221 (19.72%)	8,435 (25.52%)	46	4	16.89
ChronoPay	99	5,013 (15.67%)	5,969 (15.61%)	8,084 (20.75%)	69	5	31.51
inpay	100	5,118 (18.31%)	6,109 (18.42%)	8,408 (23.68%)	335	6	125.29
iPayment (Credit Card)	99	4,999 (16.09%)	5,932 (16.14%)	7,918 (21.62%)	38	5	21.86
Luottokunta (v1.2)	105	5,158 (18.94%)	6,127 (18.96%)	8,291 (24.72%)	34	4	15.33
Luottokunta (v1.3)	105	5,164 (18.99%)	6,135 (19.03%)	8,308 (24.80%)	35	4	15.33
Moneybookers	99	5,082 (15.90%)	6,059 (15.85%)	8,215 (21.08%)	66	4	80.85
NOCHEX	105	5,145 (18.90%)	6,111 (18.89%)	8,237 (24.67%)	33	4	15.03
PayPal Express	104	5,351 (12.63%)	6,379 (12.64%)	8,596 (17.95%)	62	11	42.15
PayPal Pro - Direct Payments	105	5,302 (19.85%)	6,339 (19.77%)	8,700 (25.61%)	65	4	20.76
PayPal Pro (Payflow) - Direct Payments	105	5,302 (19.92%)	6,339 (19.85%)	8,714 (25.71%)	63	4	20.85
PayPal Pro (Payflow) - Express Checkout	99	5,128 (14.41%)	6,107 (14.35%)	8,197 (20.08%)	31	10	31.95
PayPal Standard	99	5,040 (16.03%)	6,006 (16.01%)	8,170 (21.04%)	68	6	33.01
PayPoint.net SECPay	105	5,174 (19.09%)	6,152 (19.10%)	8,332 (24.97%)	40	4	15.80
PSiGate	106	5,231 (19.07%)	6,228 (19.04%)	8,436 (24.95%)	44	4	16.82
RBS WorldPay Hosted	99	5,019 (15.84%)	5,977 (15.92%)	8,121 (21.09%)	79	5	36.12
Sage Pay Direct	106	5,447 (20.71%)	6,515 (20.55%)	8,984 (25.97%)	95	4	26.20
Sage Pay Form	106	5,315 (19.52%)	6,329 (19.54%)	8,762 (24.55%)	55	4	19.96
Sage Pay Server	101	5,100 (14.72%)	6,067 (14.62%)	8,268 (19.78%)	42	6	28.26
Sofortüberweisung Direkt	98	5,038 (16.01%)	6,003 (15.96%)	8,160 (21.20%)	97	5	43.86
Average	102.73	5,173 (17.70%)	6,162 (17.69%)	8,376 (23.21%)	67.27	5.05	31.43

only paid for the first order. We set up a simple web page on our server to record the values of HTTP request variables. For the first order, we changed the value of return URL from the merchant URL to that of our web page. This change lets cashier Authorize.net send the payment notification request to us instead of the merchant. We replayed the recorded values of the request variables from the first order for the cashier-to-merchant request of the second order. We paid nothing for the second order at first but received a dirty diaper game package shipped from California.

A chocolate online shop (PayPal Standard). Module PayPal Standard is vulnerable to merchant ID attacks. PayPal is one of the most popular cashiers in the U.S., yet it is not used securely in this payment module. In our experiment on the chocolate online shop, we simply changed the merchant ID from the chocolate merchant's PayPal account to our own PayPal account for the user-to-cashier payment request. In this way, we received three pieces of chocolate although the payment was not made to the chocolate merchant at first.

C. Performance Evaluation

Table III shows some data that we collected during symbolic execution to demonstrate the performance of our tool. Column "Files" to column "States" show average numbers for all merchant nodes, while columns "Flows" and "Time (s)" show total numbers of merchant nodes. For the IR of each merchant node, we report the number of parsed files (column "Files"), the number of nodes and node coverage (column "Nodes (%)"), the number of edges and edge coverage (column "Edges (%)") and the number of statements and statement coverage (column "Stmts (%)"). Additionally, column "States" shows the total number of end execution states; column "Flows" shows the total number of logic flows among user, cashier and merchant during checkout; and column "Time (s)" shows the total analysis time in seconds for each payment module.

Merchant nodes are nontrivial to analyze. The number of files that each merchant node includes ranges from 98 to 106, with an average of 102.73. An IR has 5,173 basic blocks (nodes), 6,162 control flow edges and 8,376 statements on average. The coverage of nodes, edges and statements is calculated for the main function, function bodies and method bodies. Some defined functions, defined class methods and even some branches of the main function may not be executed at all in the checkout process. On average, the symbolic execution of each merchant node has a CFG node coverage of 17.70%, an edge coverage of 17.69% and a statement coverage of 23.21%.

To estimate the efforts of manual code review, we have also counted the lines of code that are related to the checkout process for payment modules. In general, the number of lines of code is slightly higher than the number of statements listed in Table III for each payment module. For example, for module PayPal Express, there are 8,727 lines of code in total to review while its IR has 8,596 statements. In addition to code, manual reviewers need to examine database tables and cashiers' documentation.

On average, it takes 31.43 seconds to explore 67.72 execution states in 5.05 logic flows for each payment module. In simple cases, it takes only 4 logic flows to initiate the checkout process, make a payment on a cashier's server, notify the merchant of the payment and complete the order. Module PayPal Express has the most complex logic flows. It uses 11 logic flows to obtain a `ppe_token` for each payment transaction, start an express checkout with function `setExpressCheckout`, make a payment on a PayPal server, get payer details with function `getExpressCheckoutDetails` and complete the sale with function `doExpressCheckoutPayment`. Module inpay has

TABLE IV: Coverage Results.

Payment Module	**Main**			**Func Stmts (%)**	**Class Stmts (%)**
	Nodes (%)	**Edges (%)**	**Stmts (%)**		
2Checkout	498 (39.60%)	693 (28.86%)	1,246 (58.89%)	2,249 (17.65%)	4,891 (19.76%)
Authorize.net Credit Card AIM	498 (40.20%)	693 (29.37%)	1,246 (59.94%)	2,249 (19.65%)	5,051 (20.40%)
Authorize.net Credit Card SIM	498 (39.60%)	693 (28.86%)	1,246 (58.89%)	2,249 (18.45%)	4,941 (20.32%)
ChronoPay	463 (36.04%)	647 (26.24%)	1,130 (54.34%)	2,249 (14.64%)	4,705 (15.61%)
inpay	510 (39.70%)	709 (30.22%)	1,218 (56.17%)	2,276 (17.27%)	4,915 (18.60%)
iPayment (Credit Card)	454 (38.25%)	632 (27.90%)	1,116 (59.10%)	2,249 (16.10%)	4,554 (15.15%)
Luottokunta (v1.2)	498 (39.60%)	693 (28.86%)	1,246 (58.89%)	2,249 (17.30%)	4,797 (19.32%)
Luottokunta (v1.3)	498 (39.60%)	693 (28.86%)	1,246 (58.89%)	2,249 (17.34%)	4,814 (19.46%)
Moneybookers	471 (36.12%)	656 (26.63%)	1,139 (54.32%)	2,249 (14.52%)	4,828 (16.29%)
NOCHEX	498 (39.60%)	693 (28.86%)	1,246 (58.89%)	2,249 (17.30%)	4,743 (19.19%)
PayPal Express	575 (28.91%)	797 (21.32%)	1,324 (44.76%)	2,249 (11.75%)	5,024 (13.66%)
PayPal Pro - Direct Payments	498 (40.20%)	693 (29.37%)	1,246 (59.94%)	2,249 (19.88%)	5,206 (19.87%)
PayPal Pro (Payflow) - Direct Payments	498 (40.20%)	693 (29.37%)	1,246 (59.94%)	2,249 (19.81%)	5,220 (20.09%)
PayPal Pro (Payflow) - Express Checkout	508 (34.70%)	706 (25.71%)	1,201 (52.96%)	2,249 (13.12%)	4,747 (15.07%)
PayPal Standard	477 (36.76%)	665 (27.08%)	1,151 (54.88%)	2,249 (15.09%)	4,770 (15.69%)
PayPoint.net SECPay	498 (39.60%)	693 (28.86%)	1,246 (58.89%)	2,249 (17.67%)	4,838 (19.64%)
PSiGate	498 (40.20%)	693 (29.37%)	1,246 (59.98%)	2,249 (17.74%)	4,942 (19.39%)
RBS WorldPay Hosted	461 (36.63%)	643 (27.02%)	1,132 (55.35%)	2,249 (14.97%)	4,740 (15.81%)
Sage Pay Direct	498 (40.20%)	693 (29.37%)	1,246 (59.94%)	2,249 (20.08%)	5,490 (20.67%)
Sage Pay Form	498 (39.70%)	693 (29.00%)	1,246 (58.97%)	2,251 (17.55%)	5,266 (19.41%)
Sage Pay Server	463 (36.07%)	645 (26.28%)	1,151 (55.42%)	2,249 (13.45%)	4,868 (14.27%)
Sofortüberweisung Direkt	470 (36.69%)	653 (26.94%)	1,136 (55.28%)	2,249 (15.41%)	4,776 (15.82%)
Average	492 (38.10%)	685 (27.92%)	1,211 (57.03%)	2,250 (16.67%)	4,915 (17.89%)

the longest analysis time (125.29 seconds) and also the largest number of execution states (335 states). The performance results show that our automated detection is more efficient and comprehensive than manual analysis. When we manually confirmed the detected logic vulnerabilities, we need around 30 minutes for each payment module. We spent about 15 minutes to read control flows of merchant pages and cashier documentation, and another 15 minutes to find valid inputs that lead to logic attacks.

We have adopted a few optimizations to speed up our analysis and two of them significantly reduced the analysis time. The first optimization sets the maximum number of similar execution states in a work list to one. This means that whenever the analysis stores a new execution state in a work list, it first checks if there already exists an execution state with the same program counter and logic state. If yes, the new execution state is discarded. Since such two execution states often differ only slightly, discarding the second state has no impact on the vulnerability analysis result. The analysis time for each payment module is limited to 10 minutes. When we increased the length of a work list to two, timeout events occurred before the analyses were completed. The second optimization sets some symbolic session variables to be not `null`, just like what they should be in a normal checkout process. For example, `$_SESSION['customer_id']` and `$_SESSION['cartId']` are specified as not `null`. The first few basic blocks in the IR of a merchant node often check whether some session variables are `null`. The second optimization rules out irrelevant branches at an early state of a symbolic execution process. This accelerates our analysis considering that the number of states usually grows at an exponential rate.

Table IV shows detailed coverage results. All the numbers in this table are average numbers of all merchant nodes. Columns under "Main" show the average numbers and coverage results (listed in parentheses) for the nodes, edges, and statements of the main functions in analyzed merchant nodes. Columns "Func Stmts (%)" and "Class Stmts (%)" show the average numbers and coverage results (listed in parentheses) for defined functions and classes of analyzed merchant nodes respectively.

On average, our symbolic execution covers 38.10% of 492 main-function nodes, 27.92% of 685 main-function edges and 57.03% of 1,211 main-function statements. Additionally, it covers 16.67% of 2,250 statements in defined function bodies and 17.89% of 4,915 statements in defined classes. Main function is the entry of each merchant node, and the average coverage for the statements of main functions is much higher than the coverage for defined functions and classes. It is obvious in Table IV that the deviation of class statement coverage is the highest. This is because different payment modules are integrated into the checkout process as plug-ins with dynamic class construction, and they have little influence on the statement numbers of the main functions and defined functions.

Our symbolic execution is developed for security analysis rather than achieving high coverage. The average coverage of all three types of statements (23.21% as shown in Table III) is lower than the coverage of main-function statements (57.03%), but higher than the coverage of defined function statements (16.67%) and the coverage of class statements (17.89%). There are three reasons for the low coverage. First, not all statements in defined functions and class methods are used in each merchant node. One merchant node may only need a few functionalities provided by defined functions and class methods. Second, our tool explores control flows of CFGs based on branch feasibility. Note that one merchant page is often divided into multiple merchant nodes based on different request parameter values. Our exploration is based on merchant nodes, but the coverage is calculated using merchant pages. This explains why some modules have low coverage. The callback page of PayPal Express for example, has a `switch` statement based on the value of request variable `$_GET['osC_Action']` near the beginning of the page. It has different branches to handle "cancel", "callbackSet", "retrieve" and default actions. For a merchant node of this page, only one `switch` branch is taken. Third, our specifications of some user inputs help us avoid the exploration of a few irrelevant control flows. Not all possible combinations of user inputs need to be examined for vulnerability detection, therefore our analysis focuses on user inputs that are related to the checkout process.

D. Discussions

The implementation of our detection tool is neither sound nor complete. For all the logic vulnerabilities detected by our tool, we carefully examined and tested each one to confirm that they are true positives. There is no observed false positives to the best of our knowledge. We cannot guarantee the absence of logic vulnerabilities because of the difficulty of exploring all possible logic flows in large real-world e-commerce applications. We hope our tool can help developers write secure payment modules and raise their security awareness.

Our static analysis still faces nontrivial challenges which include dynamic features of PHP, constraint solving and regular expressions. Typically, static analyses are limited in handling dynamic language features (e.g. dynamic includes, dynamic class, array and object construction), and the dynamic features of PHP also most significantly influence the scalability and precision of our analysis. For a precise resolution of dynamic features, specifications are incorporated for some critical code.

Our current implementation does not support JavaScript analysis yet. It is possible that some links to merchant nodes or cashier nodes are generated by JavaScript code on the client side. We did not encounter any JavaScript links in our experiments but our test subject may not be representative of other e-commerce applications. Detecting links in JavaScript code is a difficult task because of the various dynamic features of the JavaScript language. For example, its `eval` function, which executes statements provided as strings at run time, can be invoked in many different ways. For e-commerce applications that heavily use JavaScript, we may need to incorporate JavaScript analysis to detect critical URLs that are dynamically generated.

Automated analysis incurs significant engineering efforts and the amortized development cost can be kept low for e-commerce software with a large number of payment modules. Symbolic execution allows systematic exploration and is particularly useful to model HTTP requests/responses from cashiers and users as symbolic values can be used (rather than concrete values). In contrast, manual code review is error-prone, and it is difficult to cover all possible attack vectors and important control-flows (which may explain why many serious vulnerabilities still exist). The number of payment modules (928 for osCommerce) and the two vulnerable Luottokunta modules illustrate the difficulty of detecting missing/insufficient checks. However, for basic e-commerce software with only a few payment modules, manual code review may be a viable alternative.

It is possible that there exist multiple execution states with unique logic states when we reach the final merchant node during checkout. There is no universal criterion as to which logic state should be picked over another for valid logic flows, and we leave the selection of logic states to developers who have the best judgment. Our current tool includes all taint operations and flows in logic states as a reference, and uses heuristics based on our observations to rank logic states. The logic state that should be picked is often the one that has the least number of taint annotations, excluding exposed signed tokens. The reason is that our symbolic execution may conservatively explore a branch that will not be taken in practice, and only the opposite branch contains checks on payment status.

VI. Related Work

Logic vulnerabilities in e-commerce applications. The uniqueness of logic vulnerabilities, together with their great impact, has attracted the attention of researchers in recent years. Wang *et al.* [30] are the first to analyze logic vulnerabilities in Cashier-as-a-Service based web stores. Through manual security analysis, they found serious logic flaws that can lead to inconsistent payment status between a merchant's server and a cashier's server. Their follow-up work, InteGuard [33], offers dynamic protection of third-party web service integrations, including the integration of cashier service in merchants' websites. In contrast to their work, we seek to comprehensively examine various attack vectors on payment status and automatically detect logic vulnerabilities before the deployments of e-commerce applications. We discovered a new attack vector on currency which allows an attacker to modify the currency of a payment to her advantage, and designed a symbolic execution framework to systematically explore critical logic flows of checkout processes.

Parameter pollution vulnerabilities in web applications. Another active line of research is HTTP Parameter Pollution (HPP) in web applications. It is a common attack vector for various vulnerabilities which include logic vulnerabilities. WAPTEC [5] takes a white-box approach that combines symbolic execution and dynamic analysis to detect parameter tampering vulnerabilities in PHP applications, while NoTamper [4] and PAPAS [2] adopt black-box based approaches. NoTamper [4] detects insufficient server-side validations where a server fails to replicate the validations on the client side. PAPAS [2] aims at automated discovery of parameter pollution based on a black-box scanning technique for vulnerable parameters. Our approach also makes the assumption that user inputs are untrusted. However, in contrast to parameter pollution detection which examines parameters in isolation, our approach detects logic vulnerabilities in e-commerce applications by linking and analyzing the logic flows of a checkout process.

Other logic vulnerabilities in web applications. Besides attacks on e-commerce applications, logic vulnerabilities also open doors to other attacks which include access control attacks, single sign-on attacks and workflow violations in web applications. First, access control vulnerability exposes privileged functionality or resources to unauthenticated users. Nemesis [12] performs dynamic information flow tracking based on specified access control lists, while static approaches analyze source code to detect unprotected accesses [14, 26, 27]. Second, Wang *et al.* discovered new single sign-on attacks [31], and InteGuard moves a step forward [33] by using a proxy-based approach which checks a set of inferred invariants to let merchants safely integrate third-party web services. Third, to detect deviations of normal workflows, it is important to first establish a good guideline of correct workflows. Such a guideline can be specified by developers [19], inferred from client-side validations which should be replicated on the server side [4, 17], or obtained from dynamic analyses [3, 11, 15, 22]. An alternative way of thwarting logic attacks is secure-by-construction. Both Swift [8] and Ripley [29] aim to offload some computations to the client side while ensuring the consistency of logic states between servers and clients for modern web applications. Logic vulnerabilities in e-commerce applications are one important subtype of general logic vulnerabilities in web applications. Focusing on this particular domain, we are able to design an invariant of secure payments to detect logic vulnerabilities which are application-specific.

Symbolic execution and taint analysis. Symbolic execution and taint analysis are two widely used techniques in security research. Schwartz *et al.* [25] provide a high-level view of dynamic taint analysis and forward symbolic execution. Symbolic execution is a powerful technique that can be adopted for a diverse set of languages and problem settings ever since the seminal work by King [21]. For traditional programs, KLEE [6] is capable of automatically generating tests that achieve high coverage on even complex programs. For server-side languages, Halfond *et al.* [18] apply symbolic execution to precisely identify interfaces in the Java Enterprise Edition (JEE) framework, while Rubyx [7] detects security vulnerabilities based on specifications by symbolically executing Ruby-on-Rails web applications. For JavaScript, a client-side language widely used in web applications, Saxena *et al.* [24] designed and implemented a symbolic execution framework which can handle string constraints. Pixy [20] is a static taint analyzer built for PHP applications, and it detects injection vulnerabilities with taint analysis based on specifications of taint sources and sinks. Our approach combines symbolic execution with taint analysis in a novel way to detect potential logic attacks on payment status.

VII. Conclusion

Merchants should carefully verify each critical component of payment status to ensure the consistency of payment status between merchants' severs and cashiers' servers. This paper proposes the first static approach to automatically detect logic vulnerabilities in e-commerce web applications. Our key observation is that secure checks on payment status must verify the integrity and authenticity of order ID, order total, merchant ID and currency. Our framework integrates symbolic execution with taint analysis to track critical logic states, which include payment status, across checkout nodes. Our tool explored important logic flows, scaled to 22 unique real-world payment modules and detected 11 unknown vulnerabilities along with one known vulnerability. For future work, we plan to support additional path exploration strategies for our symbolic execution, add function summaries to improve performance and apply our analysis to a larger number of popular e-commerce web applications.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Davide Balzarotti, for useful feedback on earlier versions of this paper. We also gratefully acknowledge Ellen Auriti, Michael Sweeney, and Lynette Temple with the University of California for advice on our proof-of-concept experiments. This research was supported in part by NSF Grants 0917392, 1117603, 1319187, and 1349528. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] osCommerce Online Merchant. http://www.oscommerce.com/.

[2] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *Proceedings of Network and Distributed System Security*, 2011.

[3] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of Computer and Communications Security*, 2007.

[4] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. NoTamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of Computer and Communications Security*, 2010.

[5] Bisht, Prithvi and Hinrichs, Timothy and Skrupsky, Nazari and Venkatakrishnan, V. N. WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of Computer and Communications Security*, 2011.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of Operating Systems Design and Implementation*, 2008.

[7] A. Chaudhuri and J. S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of Computer and Communications Security*, 2010.

[8] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of Symposium on Operating Systems Principles*, 2007.

[9] Common Vulnerabilities and Exposures. CVE-2009-2039. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2039, 2009.

[10] Common Weakness Enumeration. CWE-840 business logic errors. http://cwe.mitre.org/data/definitions/840.html.

[11] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of Recent Advances in Intrusion Detection*, 2007.

[12] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication and access control vulnerabilities in web applications. In *Proceedings of the USENIX Security Symposium*, 2009.

[13] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[14] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the EAR: Discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of Computer and Communications Security*, 2011.

[15] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the USENIX Security Symposium*, 2010.

[16] J. Grossman. Seven business logic flaws that put your website at risk. http://www.whitehatsec.com/home/assets/WP_bizlogic092407.pdf, 2007.

[17] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proceedings of World Wide Web*, 2009.

[18] W. G. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of International Symposium on Software Testing and Analysis*, 2009.

[19] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of Automated Software Engineering*, 2010.

[20] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of Symposium on Security and Privacy*, 2006.

[21] J. C. King. Symbolic execution and program testing. In *Communications of ACM*, 1976.

[22] X. Li and Y. Xue. BLOCK: A black-box approach for detection of state violation attacks towards web applications. In *Proceedings of Annual Computer Security Applications Conference*, 2011.

[23] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of World Wide Web*, 2005.

[24] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of Symposium on Security and Privacy*, 2010.

[25] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of Symposium on Security and Privacy*, 2010.

[26] S. Son, K. S. McKinley, and V. Shmatikov. Fix Me Up: Repairing access-control bugs in web applications. In *Proceedings of Network and Distributed System Security*, 2013.

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

[28] U.S. Census Bureau. Quarterly retail e-commerce sales. http://www.census.gov/retail/mrts/www/data/pdf/ec_current.pdf, 2013.

[29] K. Vikram, A. Prateek, and B. Livshits. Ripley: Automatically securing Web 2.0 applications through replicated execution. In *Proceedings of Computer and Communications Security*, 2009.

[30] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online – security analysis of Cashier-as-a-Service based web stores. In *Proceedings of Symposium on Security and Privacy*, 2011.

[31] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed Single-Sign-On web services. In *Proceedings of Symposium on Security and Privacy*, 2012.

[32] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of Programming Language Design and Implementation*, 2007.

[33] L. Xing, Y. Chen, X. Wang, and S. Chen. InteGuard: Toward automatic protection of third-party web service integrations. In *Proceedings of Network and Distributed System Security*, 2013.