Toward Black-Box Detection of Logic Flaws in Web Applications

Giancarlo Pellegrino
EURECOM, France
SAP Product Security Research, France
giancarlo.pellegrino@eurecom.fr

Davide Balzarotti
EURECOM, France
davide.balzarotti@eurecom.fr

***Abstract*—Web applications play a very important role in many critical areas, including online banking, health care, and personal communication. This, combined with the limited security training of many web developers, makes web applications one of the most common targets for attackers.**

In the past, researchers have proposed a large number of white- and black-box techniques to test web applications for the presence of several classes of vulnerabilities. However, traditional approaches focus mostly on the detection of input validation flaws, such as SQL injection and cross-site scripting. Unfortunately, logic vulnerabilities specific to particular applications remain outside the scope of most of the existing tools and still need to be discovered by manual inspection.

In this paper we propose a novel black-box technique to detect logic vulnerabilities in web applications. Our approach is based on the automatic identification of a number of behavioral patterns starting from few network traces in which users interact with a certain application. Based on the extracted model, we then generate targeted test cases following a number of common attack scenarios.

We applied our prototype to seven real world E-commerce web applications, discovering ten very severe and previously-unknown logic vulnerabilities.

I. Introduction

Web applications play a very important role in many critical areas, and are currently trusted by billions of users to perform financial transactions, store personal information, and communicate with their friends. Unfortunately, this makes web applications one of the primary targets for attackers interested in a wide range of malicious activities.

To mitigate the existing threats, researchers have proposed a large number of techniques to automatically test web applications for the presence of several classes of vulnerabilities. Existing solutions span from black-box fuzzers and pentesting tools to static analysis systems that parse the source code of an application looking for well-defined vulnerability patterns. However, traditional approaches focus mostly on the detection of input validation flaws, such as SQL injection and cross-site scripting. To date, more subtle vulnerabilities specific to the logic of a particular application are still discovered by manual inspection [33].

Logic vulnerabilities still lack a formal definition, but, in general, they are often the consequence of an insufficient validation of the business process of a web application. The resulting violations may involve both the control plane (i.e., the navigation between different pages) and the data plane (i.e., the data flow that links together parameters of different pages). In the first case, the root cause is the fact that the application fails to properly enforce the sequence of actions performed by the user. For example, an application may not require a user to log in as administrator to change the database settings (authentication bypass), or it may not check that all the steps in the checkout process of a shopping cart are executed in the right order. Logic errors involving the data flow of the application are caused instead by failing to enforce that the user cannot tamper with certain values that propagate between different HTTP requests. As a result, an attacker can try to replay expired authentication tokens, or mix together the values obtained by running several parallel sessions of the same web application.

Formal specifications describing the evolution of the internal state and of the expected user behavior are almost never available for web applications. This lack of documentation makes it very hard to find logic vulnerabilities. For example, while being able to add several times the same product to a shopping cart is a common feature, being able to add several times the same discount code is likely a logic vulnerability. A human can easily understand the difference between these two scenarios, but for an automated scanner without the proper application model it is very hard to tell the two behaviors apart.

Only recently the research community has started investigating automated approaches to detect logic vulnerabilities [9, 18, 21]. Unfortunately, the existing solutions have serious scalability problems that limit their applicability to small applications. Moreover, the source code of the application is often required in order to extract a proper model to guide the test case generation. As a result, to date the impact of available automated tools has been quite limited.

As an alternative approach, researchers have recently re-

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
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.2014.23021

sorted to manual analysis to expose several severe logic flaws in real world commercial applications [34, 35] resulting, for instance, in the ability to shop online for free. Following the step of these previous works, in this paper we show that it is possible to automatically infer an approximate model of a web application starting from a few network traces in which a user "stimulates" a certain functionality. Our goal is not to automatically reconstruct an accurate model of the application or of its protocol (several works already exist in this direction [14, 15]) but instead to empirically show that even a simple representation of the application logic is sufficient to perform automated reasoning and to generate test cases that are likely to expose the presence of logic vulnerabilities.

In this paper we propose a technique that analyzes network traces in which users interact with a certain application's functionality (e.g., a shopping cart). We then apply a set of heuristics to identify behavioral patterns that are likely related to the underlying application logic. For example, sequences of operations always performed in the same order, values that are generated by the server and then re-used in the following user requests, or actions that are never performed more than once in the same session. These candidate behaviors are then verified by executing very specific test cases generated according to a number of attack patterns. It is important to note that our approach is not a fuzzer, and both the trace analysis and the test case generation steps are performed offline. In other words, they do not require to probe the application or generate any additional interaction and network traffic.

While our approach is application-agnostic, the choice of the attack patterns reflects a particular class of logic flaws and application domain — and in our case were customized for E-commerce applications. In particular, we applied our prototype to seven large shopping cart applications adopted by millions of online stores. The prototype discovered ten previously-unknown logic flaws among which five of them allow an attacker to pay less or even shop for free.

In summary, this paper makes the following contributions:

1) We introduce a new black-box technique to test applications for logic vulnerabilities;
2) We present the implementation of a tool based on our technique and we show how the tool can be used to test several real web applications, even with a very limited knowledge and a small number of network traces;
3) We discover ten previously-unknown vulnerabilities in well-known and largely deployed web applications. Most of these vulnerabilities have a very high impact and would allow an attacker to buy online for free from hundreds of thousands of online stores.

Structure of the paper. Section II presents the black-box approach. Section III describes the experiments that we performed and Section IV shows the results. Section V discusses the limitations of our approach and Section VI presents related work on detecting logic vulnerabilities. Finally, Section VII concludes the paper.

II. Approach

The OWASP Testing Guide 3.0 [33] suggests a four-step approach to test for logic flaws in a black-box setting. First, the tester studies and understands the web application by playing with it and reading all the available documentation. Second, she prepares the information required to design the tests, including the *intended workflow* and the *data flow*. Then she proceeds with the design of the test cases, e.g., by reordering steps or skip important operations. Finally, she sets up the testing environment by creating test accounts, runs the tests, and verifies the results.

Our approach aims at automating the previous steps in a single black-box tool. First, starting from a list of network traces containing HTTP conversations, our system infers an application model and clusters resources related to the same workflow "step" (Section II-A). Second, our technique analyzes the model and extracts a set of *behavioral patterns* (Section II-B) modeling both the workflow and data flow of the application. Third, we apply a set of *attack patterns* to automatically generate test cases (Section II-C). Finally, we execute them against the web application (Section II-D), and we use an *oracle* to verify whether the logic of the application has been violated (Section II-E).

In the rest of the section we describe each phase in details using E-commerce web applications as a running example.

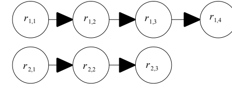
A. Model Inference

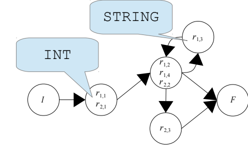
The technique we present is *passive* and *black-box*. We do not require any access to the application source code (both on the client- and server-side), and we do not actively crawl the application pages nor generate any traffic to probe its internal state. Instead, we take as input a list of HTTP conversations. These traces can be manually generated by the tester, or collected by logging real user activity.

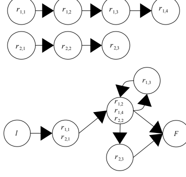
For simplicity, we consider only traces that exercise a specific functionality of the web application. For example, if the web application is a shopping cart, we use traces in which users log in, add items into the cart, and check out to buy the products. Nothing prevents the tester from generating traces that also contain other functionalities, such as browsing the online catalog or posting product reviews. However, focusing only on one aspect of the business logic helps our system to find the relevant operations with a minimum number of input traces.

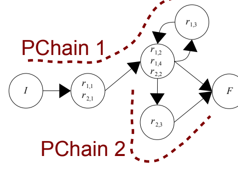
Web applications often involve multiple parties. For instance, E-commerce web applications typically involve the client, the store, and the payment service. However, the communication between them is normally channeled through the client and, therefore, we focus on this point to collect the traces. In addition, it is useful to collect data from different deployments of the same web application, to allow our inference method to identify parameter values hard-coded in a certain installation.

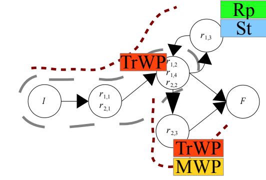
The first phase consists of building the model of the application, called *navigation graph*. This is done in two steps: resource abstraction, and resource clustering.

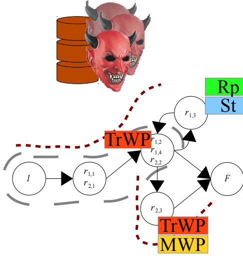












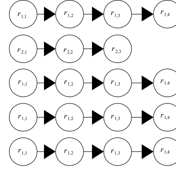


Fig. 1: Architecture of our approach.

1) Resource Abstraction

Input traces are sequences of pairs of HTTP requests and responses. The first step of the inference phase consists of creating a synthesis of the resources. Our approach currently supports JSON data objects [17] and HTML pages. However, it can be easily extended to other types such as SOAP messages [36].

We call *abstract HTML page* the collection of (i) its URL, (ii) the POST data, (iii) the anchors and forms contained in the HTML code and their DOM paths, (iv) the URL in the meta refresh tag, and, if any, (v) the HTTP redirection location header. We call *abstract JSON object* a collection of (i) its URL, (ii) the POST data, (iii) the pairs of value and path in the object, and (iv) the HTML links if any HTML code is contained. For example, Figure 2 shows the abstract resource of the following JSON object:

```
{'items': {
   'item1': ['price':19.9,'tax':1.6],
   'item2': [ ... ]}}
```

From each abstract resource we extract a set of elements corresponding to all possible parameters that appear in the URLs, in the POST data, and in all the links. Each element is characterized by a name, a value, a path, and an inferred syntactic type. Our approach supports the integer type, decimal type, URL type, email address type, word type (alphabetical strings e.g., "add", "remove", ...), string type, list type (comma-separated values), and *unknown* type (i.e., everything else). The type is associated to each element by inspecting the values of the element. Obvious priority rules are applied in

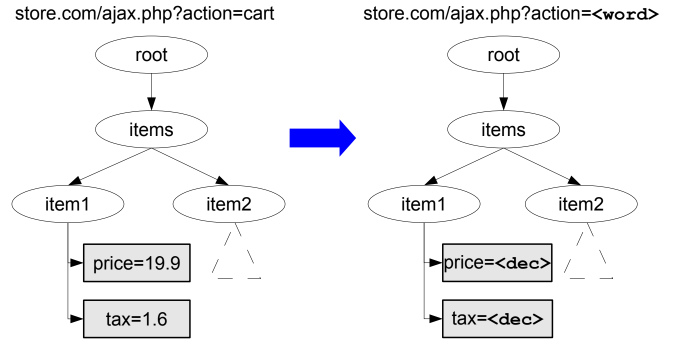


Fig. 2: Resource abstraction and syntactic type inference of a JSON data object

case of ambiguity – e.g., id=20 can be both a number and a string, but being the first a subset of the second, it is considered to be a number.

2) Resource Clustering

Modern web applications map application logic operations to different resources. For instance, the operation of displaying the shopping cart could involve an initial HTML page containing the skeleton of the web page and then use a number of asynchronous AJAX requests to populate the page with the list of items, tax, available vouchers, and so on. We cluster these resources in three phases. First, we relate asynchronous requests to the resource that originated them, i.e., synchronous resource. Then we group together resources considering both

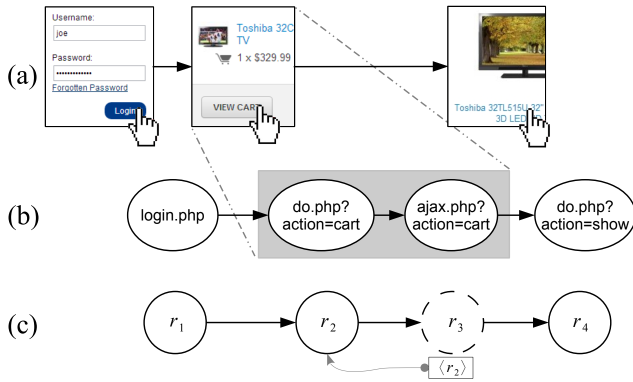


Fig. 3: (a) Application-level actions, (b) URLs requested, and (c) abstract resources with list of originators

similarity and the originators. Third, we split a cluster if a parameter of its resources encodes a command rather than carrying a value.

During the first phase, we pre-process input traces to identify AJAX requests. This can be done by checking the "X-Requested-With" HTTP request header [32] or by detecting JSON responses. After that, we associate each resource to its *originators*. Figure 3 provides an example of this first phase. In Figure 3.c we have the HTML page r_1 followed by the page r_2. Then r_2 requests r_3 by using AJAX that enriches r_2 with new HTML code, or new client-side scripts. The example then ends with r_4 that we assume to be caused by a link in r_2 or added by r_3. Figure 3.c also shows the list of originators of each resource. r_1, r_2, and r_4 have no originators, while r_3 was originated by r_2.

In the second phase, we cluster resources. In general, two resources are grouped in the same cluster if they have the same URL domain and path, the same GET/POST parameter names, and, if any, the same redirection URL. When comparing parameters we do not take into account their values, but only their syntactic types. For example, the following three URLs are equivalent:

```
store.com/do.php?action=add&id=3
store.com/do.php?action=add&id=7
store.com/do.php?action=show&id=3
```

We compare first synchronous resources as explained before, and then the asynchronous ones. Two asynchronous resources are in the same cluster if they have the same URL domain and path, GET/POST parameter names, redirection URL, and the same originators.

During the last phase, we identify the parameters that are encoding a command rather than transporting a value. For each parameter we take the pages that have the same value as that parameter. For example, the parameter `action` divides the gray cluster of Figure 4.a in two sub-groups, one for the `cart` value and one for the `show` value. We then compute the *page similarity* between pages in the same sub-group and between pages in different sub-groups. The comparison is done by looking at the DOM path of HTML forms, their action attribute (URL domain and parameter names), and the name of the nested input elements. The function is applied to sub-groups by calculating the percentage of pages that are similar.

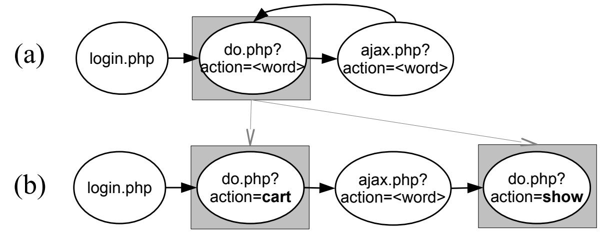


Fig. 4: (a) Clusters after comparing all the resources (b) Clusters after having identified parameters encoding a command

If the similarity inside the same sub-groups is high (more than 55%), and between different sub-groups is low (less than 45%), then we assume the parameter is used to specify an operation and we create a different node for each value. Otherwise we leave the cluster unmodified. The result of this phase is shown in Figure 4.b.

The navigation graph is a directed graph $G = (\mathcal{C} \cup \{I, F\}, E)$ where $\mathcal{C}$ is the set of clusters, I the source node, F the final node, and E the set of edges. We place the edge (u, v) if there exists one input trace π in which a resource $r' \in u$ immediately precedes a resource $r'' \in v$. Then, for each r_j at the beginning of each trace (i.e. $\pi = \langle r_j, \ldots \rangle$), we place the edge (I, u) where $r_j \in u$ and for each r_j at the end of each trace, (i.e. $\pi = \langle \ldots, r_j \rangle$) we place the edge (u, F) where $r_j \in u$. Finally, we associate to each node u the set of all the elements for every $r \in u$.

B. Behavioral Patterns

Behavioral patterns are workflow and dataflow patterns that are likely related to the logic of the application. We divide workflow patterns into *Trace Patterns*, that model what users normally do in our input traces, and *Model Patterns* that model what the navigation graph allows to be done. Finally, *Data Propagation Patterns* model how data is propagated throughout the navigation graph.

1) Trace Patterns

Trace patterns model the actions performed by the user in the input traces. In particular, we focus on three patterns:

Singleton Nodes
A node is a singleton if it is never visited more than once by any input trace. Some of the users may visit these nodes, and some may not - but no one visits them twice. For example, submitting a discount voucher can be an operation observed in some of input traces but none of them is submitting a voucher twice.

Multi-Step Operations
A Multi-Step Operation is a sequence of consecutive nodes always visited in the same order. This is very common in many functionalities in web applications. For example, payment procedures or user registrations often consist of a precise sequence of steps, and all

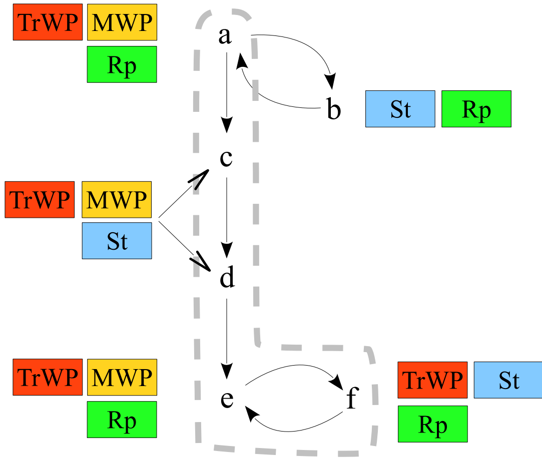


Fig. 5: Example of behavioral patterns using $\pi_1 = \langle a, b, a, c, d, e, f, e\rangle$ and $\pi_2 = \langle a, c, d, e, f, e\rangle$

traces going through those processes always execute them in the same exact order.

Trace Waypoints
We use the term *waypoint* to describe nodes that play an important role in the interaction between the user and the application. In particular, trace waypoints are those nodes that appear in all the input traces. For example, if all our traces contain a purchase, then the redirection to the payment website (e.g., PayPal) is a trace waypoint.

2) Model Patterns

Model patterns model the sequences of actions that are allowed according to the navigation graph:

Repeatable Operations
Nodes that are part of a loop in the navigation graph are associated to operations that can potentially be repeated multiple times.

Model Waypoints
Model waypoints are nodes that belong to every path in the navigation graph that goes from the source node to the final node. These nodes are not only visited in all input traces, but there is no way in the navigation graph to bypass them. By definition, every model waypoint is also a trace waypoint but not vice versa.

Figure 5 shows an example to better describe the difference between model and trace patterns. The example shows the behavioral patterns of a navigation graph extracted from two input traces $\pi_1 = \langle a, b, a, c, d, e, f, e\rangle$ and $\pi_2 = \langle a, c, d, e, f, e\rangle$. The symbols St, TrWP, Rp, and MWP stand for, respectively, singleton nodes, trace waypoints, repeatable nodes, and model waypoints. The thick dotted line delimits the multi-step operation.

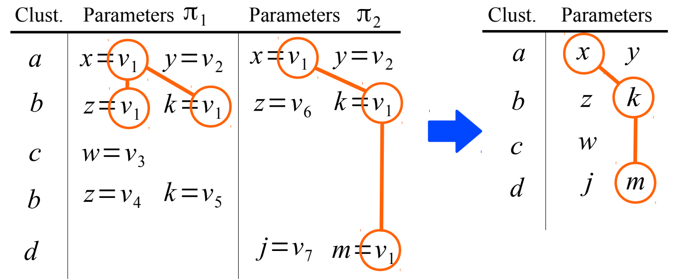


Fig. 6: Propagation Chains: from traces to the navigation graph

3) Data Propagation Patterns

A propagation chain is a set of parameters with the same value which is sent back and forth between the client and the web application during the HTTP conversation. We say that two parameters have the same value if there are some input traces in which they hold the same value, and there are no traces in which the values are different (since the user does not perform the same actions in all the traces, a certain parameter may not be present in all of them). We say that the chain is *client generated* if the initial value is chosen by the user, and *server generated* otherwise. A similar classification is used by Wang et al. [34]. However, their notion is limited to single input traces while ours is extended to traces of different lengths and to the navigation graph.

We compute propagation chains in two steps. First, we identify the propagation chain of each value within a trace. Let us consider the example in Figure 6. Here, in the input trace π_1, the parameter x has the same value of z and of k. In trace π_2, the parameter x is still equal to k, but it is now different from z. Moreover, the same value matches the parameter m. Second, by comparing the chains of traces, we remove contradictions reaching the result shown in the right side of Figure 6.

C. Test Case Generation

In this section we describe the generation of test cases. This is done by adopting a number of attack patterns that model how an attacker can use the application in an unconventional way. In particular, we focus on a set of actions an attacker could perform: repeating operations, skipping operations, subverting the order of operations, and mixing parameter values across user sessions. For each action we designed a pattern. These patterns are presented in Figure 7 and are based on the navigation graph of Figure 5. We enriched Figure 7 with numbers for showing the order in which the nodes are visited. For simplicity, we are omitting the source node I and the final node F, respectively connected to a and e.

It is important to note that, while the approach presented in this paper is generic, the choice of the attack patterns needs to reflect a particular class of logic flaws (in our case, the subversion of either the control or data-flow of the application). Other types of logic vulnerabilities, such as authentication bypass, may require the use of other patterns (e.g., randomly access administration pages) that could be added to our system

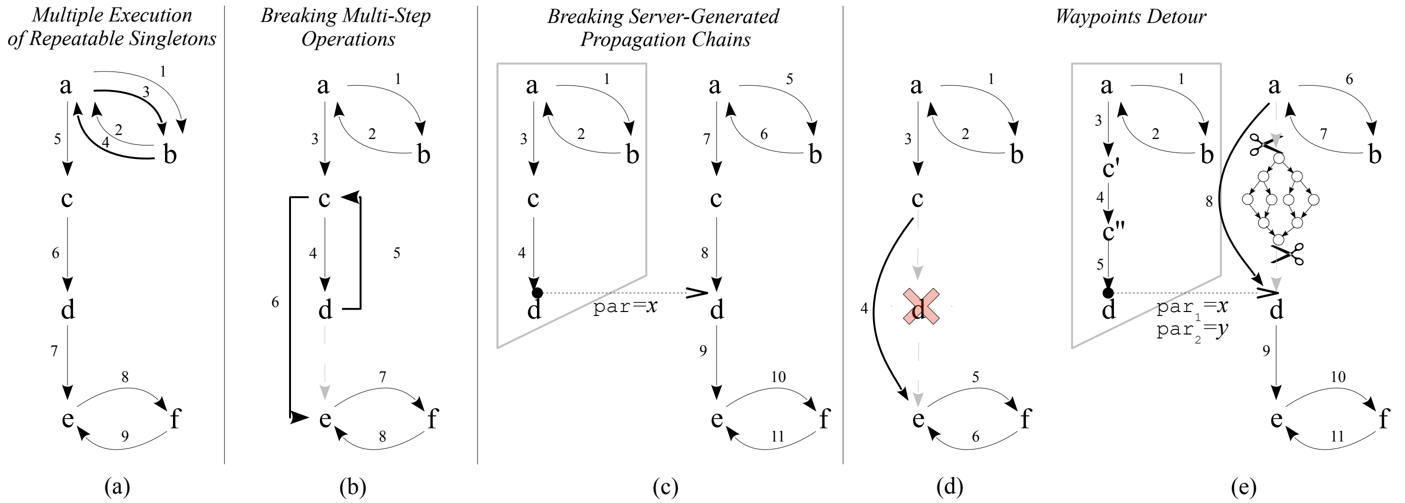


Fig. 7: Test case generation patterns

but that are outside the scope of our paper. However, the use of custom techniques to detect certain vulnerabilities is common to many other tools and approaches - e.g., a technique designed to find SQL injections cannot be used out of the box to detect other types of input sanitization vulnerabilities.

1) Multiple Execution of Repeatable Singletons

This pattern models an attacker that tries to execute an operation several times. If the model has a node that is repeatable and singleton, it means that even though there is a way to repeat an operation multiple times, this was never observed in our input traces. Therefore, the attacker tries to visit it twice.

Figure 7.a shows the steps of the test case. We select an input trace that visits b (e.g., $\langle a, b, a, c, d, e, f, e \rangle$), a repeatable and singleton node. Then we split it into two parts at the node after the singleton (e.g., $\langle a, b \rangle$ and $\langle a, c, d, e, f, e \rangle$). We call these two parts *prefix* and *suffix*. Second, we find the shortest loop from the singleton node to itself (e.g., $\langle b, a, b \rangle$). Finally, the test case is the concatenation of the prefix, the loop without the first node, and the suffix.

2) Breaking Multi-Step Operations

This pattern models an attacker that breaks multi-step operations. For example, once the payment page is reached, the attacker goes back and adds an item into the cart. In general, there are several ways of breaking the multi-step operation of Figure 5. The first approach is to use a different ordering (e.g., $\langle a, d, c, e, f \rangle$). A second approach is to interleave other steps. In this pattern, we focused on the latter approach in which we repeat a step already included in the multi-step later in the test case. For example, in the test case $\langle a, c, d, c, e, f \rangle$ in Figure 7.b we repeat c after d. In this pattern, we repeat c also after e and f, but not after a.

3) Breaking Server-Generated Propagation Chains

The goal of this attack pattern is to tamper with the data flow of the web application. An example of test case is shown in Figure 7.c. The first part of the test interacts with the application and captures the value x of a server-generated propagation chain. In the second part, we start another session and interrupt the propagation chain by replacing the value of `par` with x.

Since web applications contain many server-generated propagation chains (e.g., all the item or message IDs), this attack pattern may generate a very large number of test cases. Therefore, we focus only on two types of propagation chains: the ones containing unique values (i.e., that differ in all the input traces and are therefore related to the session) and the ones containing installation-specific values (i.e., values that are constant only within the same installation).

The test case generation is the following. First, we select the parameters belonging to the chain that appear inside an HTTP request. These parameters are called *injection points* and model the point in which an attacker can replace the value generated by the server. For example, in Figure 7.c the parameter `par` of the node d is an injection point. Second, we select two traces from different user sessions that are visiting the node of the injection point. The first is truncated at the injection point and the second is appended to the first one. With reference to Figure 7.c, the two parts are respectively at the left- and right-hand side.

4) Waypoints Detour

Waypoints are operations that are executed always by all the input traces such as payment, or providing shipping data. When these operations happen only once per input trace, they seem to indicate some sort of milestone in the execution of the business process of the web application. In the waypoint detour pattern, the attacker tries to skip these type of operations by using one of two possible techniques. If the waypoint node is not part of a propagation chain, we simply try to skip it.

Otherwise, we try to remove the part of the navigation graph between two waypoints, reconstructing the propagation chains by fetching the missing data values from another user session.

Figure 7.d shows an example of this pattern. On the left side we skip the waypoint d, while on the right side we cut the subgraph between a and d. In this second case, if the URL of node d depends on a value that appears in the segment between a and d, we prepare another user session by selecting an input trace and interrupting it at d. The generation of this part is similar to breaking propagation chains. The first user session is then $\langle a, b, a, c', c'', d \rangle$. Afterwards, we prepare the second user session that skips the sequence between a and d. In this example, there are two possibilities: skipping $\langle b, a, c', c'' \rangle$ or $\langle c', c'' \rangle$. Figure 7.e shows only the latter. In this case the test case is the concatenation of $\langle a, b, a, c', c'', d \rangle$ and $\langle a, b, a, d, e, f, e \rangle$. For this case we also support the variant in which the first user session is not interrupted at the node d.

D. Test Case Execution

The test cases described in Section II-C are abstract and still miss the details to be properly executed. For example, the values of some parameters cannot be determined from the model and need to be collected during the test case execution. In addition, it is important that after each test the application is reset to its initial state to avoid interferences between consecutive executions. For example, a test may leave a number of items in the shopping cart, thus affecting following experiments. In general, it is often sufficient to delete the cookies and empty the shopping cart at the end of each test.

The execution engine iterates over each node of the test case, concretizes the POST/GET parameters, and submits the HTTP request. The responses are parsed according to the propagation chains in order to extract server-generated parameters to be used in latter requests. If the execution engine is not able to properly reconstruct a chain (e.g., because the page that was supposed to generate its value returned an error) the execution engine aborts the execution and reports that the test was *not executed*; it reports *executed* otherwise.

E. Test Oracle

The approach we propose in this paper is completely independent from the business logic of the web application. Our technique can automatically identify behavioral patterns, and then generate test cases to break those patterns in a number of different ways. The system can also determine if a given test was executed correctly, but this is as far as it is possible to go with an application-agnostic approach. For example, replacing the value of a security token in a payment workflow would probably make the entire process fail. Unfortunately, without any knowledge about the underlying business logic, the test verdict could only say whether the pattern was applied successfully, but it can not draw any conclusion about the possible implications. Therefore, if we want our tool to be able to report possible violations of the application logic, we need to extract the sequence of events that occur during a test case execution and compare them with the *logic property* that we want to violate.

A simple way to express a logic property for shopping carts could be the following: *if an order is approved for a user, then the user must have completed a payment for the corresponding amount*. In this formulation two events play a central role: the fact that an order is placed, and the fact that a user has paid a certain amount. Another important aspect of this property is the time dependency between the two events. Since propositional logic can only express truth regardless of the time, in our approach, we model logic properties as Linear Temporal Logic (LTL) formulas [30, 23]. LTL adds temporal connectives like $\mathtt{O}$ (once in the past) to traditional logical operators like $\wedge$ (and), and $\implies$ (implies). This enables us to verify whether one event will eventually happen in the future or it already happened in the past.

For example, the above logic property can be written in LTL as follows:

$$ord_{placed} \wedge onStore(S) \implies \mathtt{O}(paid(U, I)) \qquad (1)$$

where ord_{placed}, $onStore(S)$, and $paid(U, I)$ are respectively the events *order placed*, *operation performed on the store* S, and *user* U *paid the price of item* I. Now, the problem of identifying a violation of the logic property is recast into the problem of checking whether the LTL formula is satisfied or not by a given test case.

In our approach, the *Test Oracle* is the component that given an execution of a test case returns *true* if a certain predefined logic property is violated, and *false* otherwise. The oracle is composed of two parts: an *events extractor* and an LTL formula checker. The extractor collects from the executed test a partially ordered set of events (events can happen in sequence or in parallel) grouped by user sessions. The second part verifies whether all sequences satisfy the provided LTL formula.

It is important to note that both the events and the LTL formula depend on the type of applications under test and on the type of vulnerabilities that we are interested to find. For example, to find authentication bypass vulnerabilities it would be interesting to observe events related to the user login and to the access of private pages. However, since in this paper we focus on the test of E-commerce applications, we are more interested in monitoring the money transfer and the value of the purchased items, as described in more details in the next section.

III. Experiments

We could use our tool to test online stores (e.g., Amazon). However, our tests require to attempt malformed operations and to complete a large number of checkout processes. This would be both unethical, since the application can malfunction as a result of our tests, and very expensive, since it requires to buy at least one product for each test case. Therefore, we opted to run our tests on seven well-known open source applications available for offline testing, as reported in Table I. The table also shows an estimation of their popularity, measured with the search results obtained by performing a

Web App.	No. of Installations
OpenCart	9,710,000
Magento	3,130,000
PrestaShop	650,000
CS-Cart	260,000
TomatoCart	119,000
osCommerce	80,500
AbanteCart	21,200
Total	13,970,700

TABLE I: Popularity index

number of *googledorks* [1]. Each Google query was built by combining both the URL structure (e.g., the path of the checkout endpoint) and some static HTML content extracted from the web pages (e.g., "powered by..." of the footer). As such, the numbers reported in the table are only a lower bound of the number of publicly-accessible installations available on the Internet. This conservative measurement shows that these seven applications are used by almost 14 million E-commerce installations. As a comparison, the two applications tested by Wang et al. [34] returned less than 40,000 hits using similar Google dorks.

A. General Setup

We installed two instances of each web application (hereinafter Store A and Store B). All installations except for AbanteCart and PrestaShop were then configured to use both the PayPal Express Checkout [3] and the PayPal Payments Standard [4] methods. In total we prepared 12 configurations[1].

All applications were configured in *sandbox* mode. In this configuration, each application performs transactions by using the PayPal sandbox payment gateway. These payments do not involve real money as they are performed between the seller and buyer testing accounts.

B. Testing Oracle

In their experiments, Wang et al. [35] used the following shopping cart property:

"The store S changes the status of an item I to "paid" with regard to a purchase being made by user U if and only if (i) S owns I; (ii) a payment is guaranteed to be transferred from an account of U to that of S in the CaaS; (iii) the payment is for the purchase of I, and is valid for only one piece of I; (iv) the amount of this payment is equal to the price of I."

However, this property is not entirely verifiable in a black-box setting. For instance, it is not possible to test the truth of the predicate "S owns I" nor to check whether the due amount has been transferred to the merchant's account. Therefore, we simplified the above invariant by removing the non-verifiable clauses. The new property that can be used for automated black-box testing becomes:

[1]When we did the experiments, AbanteCart and PrestaShop were providing, respectively, only PayPal Payments Standard and PayPal Express Checkout.

When the store S confirms the user U that an order has been placed, then in the past U paid S the amount equal to the price of I and U agreed on purchasing I from S.

We modeled the logic property using the following events extracted during each test case execution:

- ord_{placed} when the shop confirms that the order has been placed;
- $onStore(S)$ when an operation has been performed on the store S;
- $paid(U, I)$ when the user U authorizes the payment gateway to pay the price of I;
- $toStore(S)$ when the payment is meant for the store S;
- $ack(I)$, when the user acknowledges to buy I.

The logic property is then formulated as:

$$\begin{aligned} &ord_{placed} \wedge onStore(S) \implies \\ &\quad \mathtt{O}(paid(U, I) \wedge toStore(S) \;\wedge \\ &\qquad \mathtt{O}(ack(U, I) \wedge onStore(S))) \end{aligned} \tag{2}$$

C. Input Traces

To generate the input traces we created two user accounts, U_1 and U_2, each controlling a PayPal buyer testing account. For each web application we captured in total six HTTP conversations, three for each store: one with U_1 buying one item, one with U_2 buying another item, and one with U_1 buying two different items. All the input traces satisfy the logic property 2. These input traces were sufficient to stimulate the main shopping cart functionalities, but a better training could be used in the future to expose also more subtle features, or for detecting different types of logic flaws.

D. Test Case Generation

Table II shows the test cases grouped by attack pattern. The test case generation produced about 3100 test cases, an average of 262 per application. Table II shows also the test execution result. An execution failed when the test case brought the application in a state in which it was impossible to proceed (e.g., because of error pages in intermediate steps). This is a common result, since by definition our tests stress the application to expose some unexpected behavior. The number of test cases violating the LTL formula is reported in Table III. As mentioned before, there are events that are not visible to the oracle. Therefore, a violation to the LTL formula does not always correspond to a vulnerability. In fact, it is possible that further checks performed in the back end of the application would detect and block the attack. To distinguish logic vulnerabilities from other bugs (e.g., erroneously reporting to the user a failed transaction as successful) we manually inspected the balance sheets of the merchant, the list of orders, and their status. Whenever the result was not confirmed by our manual

Web App.		Test Case Generation Time hh:ss	(a)	(b)	(c)	(d), (e)	Test Case Execution Time hh:ss	Exec.	Not Exec.	Total
AbanteCart	Std	$\ll$ 00:01	9	51	21	152	04:51	74	159	233
Magento	Exp	00:02	10	82	5	246	16:23	240	103	343
	Std	00:02	14	62	7	303	14:50	210	176	386
OpenCart	Exp	00:01	10	77	3	83	02:34	140	33	173
	Std	00:01	15	38	22	60	02:08	71	64	135
osCommerce	Exp	$\ll$ 00:01	4	13	6	142	03:22	117	48	165
	Std	00:01	8	63	10	144	03:42	128	97	225
PrestaShop	Exp	$\ll$ 00:01	12	22	3	100	02:42	85	52	137
TomatoCart	Exp	00:02	9	68	10	215	04:54	238	64	302
	Std	00:02	17	32	37	138	04:36	115	109	224
CS-Cart	Exp	00:05	8	24	6	562	12:02	347	253	600
	Std	00:02	16	54	15	137	05:29	127	95	222
Total			132	586	145	2282		1892	1253	3145

TABLE II: Statistics per application on the test case generation and test case execution phases. Columns (a), (b), (c), (d), and (e) are the attack patterns in Figure 7 while columns Exec. and Not Exec. refer to the two possible outcomes of the test execution engine.

Web App.		No. of Viols.	Caused by Bugs	Vulns.
AbanteCart	Std	17	16	1
Magento	Exp	65	65	-
	Std	126	126	-
OpenCart	Exp	58	46	12
	Std	30	30	-
osCommerce	Exp	42	22	20
	Std	35	34	1
PrestaShop	Exp	-	-	-
TomatoCart	Exp	90	65	25
	Std	24	24	-
CS-Cart	Exp	313	313	-
	Std	109	108	1
Total		909	849	60
		100%	93.4%	6.6%

TABLE III: Number of test cases violating Property 2 and the root cause.

inspection, we classified it as a normal bug. The remaining cases correspond instead to anomalous behaviors associated to real software vulnerabilities, as explained in the next Section. It is important to note that over 28.9% of the test cases generated by our approach brought the application in a state that violated the LTL formula, and 1 test out of 52 exposed a previously-unknown logic vulnerability.

Test case generation does not require much resources, while the execution phase can be quite time consuming (16h for Magento). This is largely due to the lack of parallelization in our experiments, and to the fact that the PayPal sandbox is much slower than its live counterpart. The model inference – omitted from Figure II – required an average of 9m per application to build the navigation graphs that, in average, contained 34 nodes and 48 edges.

IV. Results

Table III reports the total number of violations of the security property 2. In other words, by tampering with either the workflow or the data flow according to our attack patterns, our system was able to bring the web application in a faulty state in 909 cases. All these cases corresponded to tests that were executed until the final page in which the store congratulates the customer for the successful purchase (that caused the generation of the events $ord_{placed} \wedge onStore(S)$) even though the paid amount was not correct. While these violations are all the consequences of bugs in the application code, not all of them can be exploited by an attacker.

This is an important point and a fundamental limitation of black-box approaches. Our tool can only observe the application state "from the outside", and therefore it cannot distinguish between a presentation bug (in which the information displayed on the web pages are wrong but the internal state of the application is correct) and a more serious vulnerability (in which also the internal state is compromised).

To distinguish between the two types of bugs, we manually inspected the state of the backend database: the result is the distinction summarized in Table III between harmless presentation bugs (93.4%) and real vulnerabilities (6.6%). While these results indicate that the *true positive* rate of our tool is 6.6%, also the remaining 93.4% of the violations correspond to real bugs in the application that need to be fixed by the developers. Once all the presentation issues have been solved, the alarms raised by our tool would correspond only to exploitable vulnerabilities.

A. Vulnerabilities

Table III shows that 60 of our test cases (1.9% of the total) exposed a logic vulnerability in the target applications. We discovered the following flaws:

- In osCommerce 2.3.1, CS-Cart 3.0.4, and AbanteCart 1.0.4 with PayPal Payments Standard a malicious

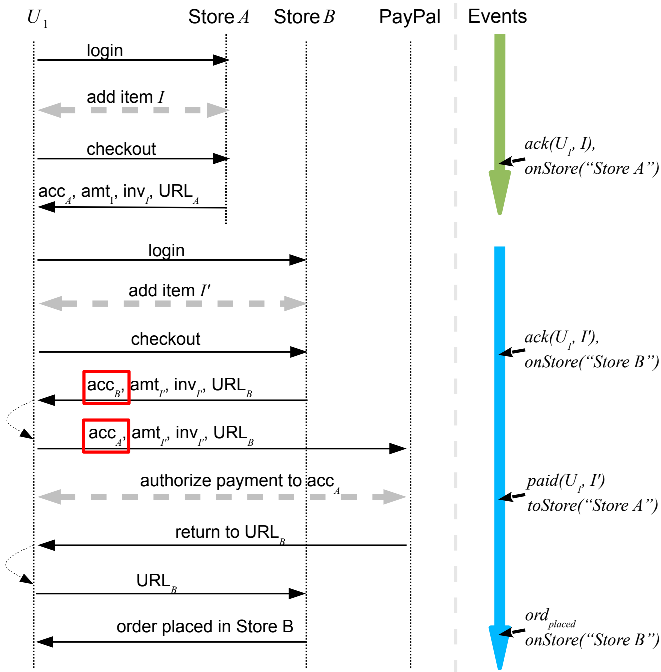


Fig. 8: Shopping for free with osCommerce 2.3.1 and AbanteCart 1.0.4

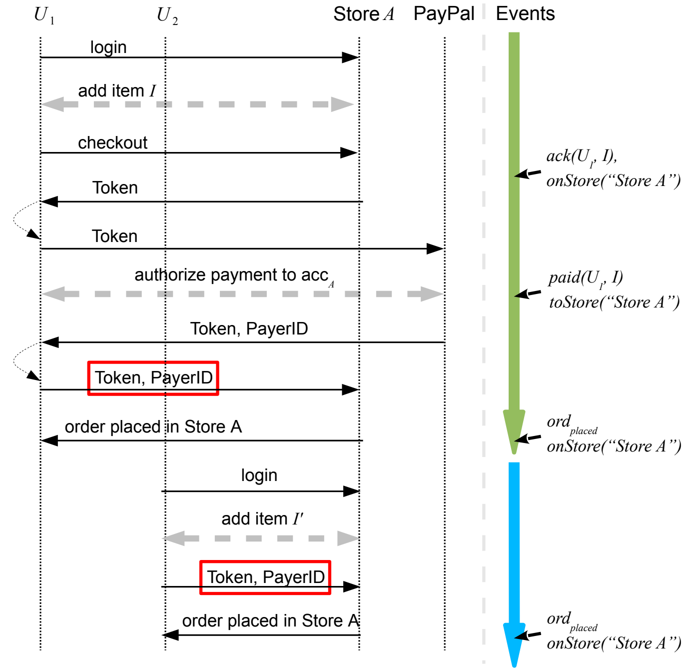


Fig. 9: Paying less with OpenCart 1.5.3.1 and TomatoCart 1.1.7

customer can shop for free (exploitable)

- In OpenCart 1.5.3.1 and TomatoCart 1.1.7 with PayPal Express Checkout a malicious customer can pay less (exploitable)
- In TomatoCart 1.1.7 with PayPal Express Checkout a malicious customer can shop for free (exploitable)
- OpenCart 1.5.3.1, TomatoCart 1.1.7, and osCommerce 2.3.1 with PayPal Express Checkout a customer can pay an amount different from what she authorized (not exploitable)
- TomatoCart 1.1.7 with PayPal Express Checkout a customer pays another customer's cart (not exploitable)

All the exploitable flaws have been already responsibly disclosed. When the developers did not answer within two weeks of our notification, we reported the vulnerabilities also to the US Cert[2]. In the following we describe each class of vulnerability we discovered in our experiments.

1) osCommerce, CS-Cart, and AbanteCart with PayPal Payments Standard - Shopping for Free

These flaws were discovered by tests that interrupted the server-generated propagation chain transporting the PayPal account of the merchant. An example is shown in Figure 8. The left-hand side of the Figure shows the message sequence chart while the right-hand side shows events grouped by user session. Each user session begins with a $login$ message. The events show how the violation was detected by the oracle. At the end of the execution, $ord_{placed} \wedge onStore(\text{“}Store\ B\text{”})$ is satisfied as all the events in it were observed. However, the left-hand side of the Formula (1) is not satisfied because none of the events in it were observed.

The manual inspection verified that (i) no payment was made to the Store B, (ii) the status of the order in the back office of Store B was "completed", and (iii) the invoice was paid. It is straightforward to turn the above test into a real attack. Indeed, when redirected to PayPal, an attacker can replace the seller PayPal account with another PayPal account under her control. In this case, the attacker can pay herself for an item she buys in an online shop.

2) OpenCart and TomatoCart with PayPal Express Checkout - Pay Less

In OpenCart and TomatoCart with PayPal Express Checkout an attacker can pay less than the value of the items. The flaw has been detected by using the waypoints detour pattern. The test case generator produced 11 test cases for OpenCart and 11 for TomatoCart in which the user U_2 skips the nodes of the redirection to PayPal for the payment and reconstructs the URL with values taken from the user session of U_2. A representative test case is shown in Figure 9. In the second user session $ord_{placed} \wedge onStore(\text{“}Store\ A\text{”})$ is satisfied. However, the other clauses of the formula are not satisfied because neither the user acknowledgment nor the payment were observed.

The manual inspection found two distinct orders in the list of orders, one for I and for I'. Both orders were in the state "paid" and ready for shipping. However, the balance sheet of the merchant contains only the transaction for I, while nothing is recorded for I'.

[2]See http://www.kb.cert.org/vuls, IDs 459446, 207540, and 583564

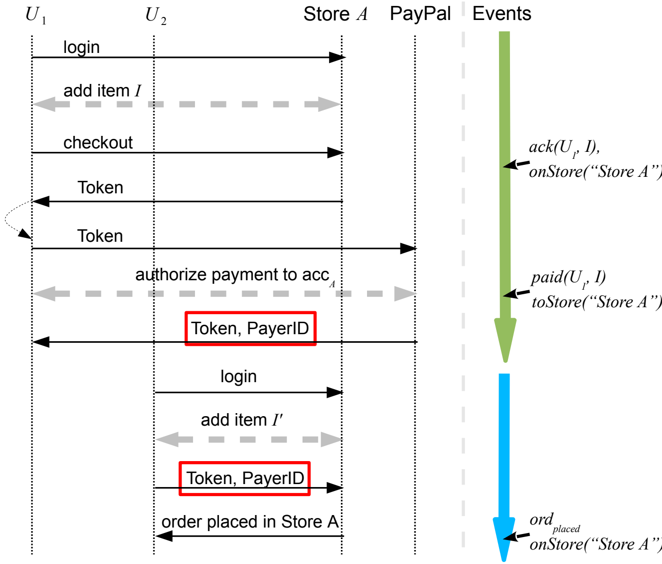


Fig. 10: Shopping for free with TomatoCart 1.1.7

This test can be turned into an attack by first buying a cheap item and intercepting the redirection URL from PayPal to the store. Then the attacker can login again, add an expensive item to the cart, and replay the URL captured before to complete the transaction. Even worse, we verified that the attacker (or any other user) can reuse the same $TokenID$ and $PayerID$ to complete an arbitrary number of additional fake transactions. This process is only bounded by the timeout set by PayPal on the token.

3) TomatoCart with PayPal Express Checkout - Shopping for Free

This problem has been identified by 11 different test cases generated with the waypoint detour pattern. A representative test case is shown in Figure 10. Figure 10 shows that in the second user session $ord_{placed} \wedge onStore(\text{"}Store\ A\text{"})$ is satisfied. However, the other clauses of the formula are not satisfied because neither user acknowledgment nor the payment were observed.

The manual inspection verified that no payment for I and for I' were done. However, the list of orders contained the order for I' in a "paid" state and ready for shipping. This test case can be turned into an attack as shown before with the difference that the attacker ends the first user session after receiving $Token$ and $PayerID$ from PayPal.

4) osCommerce, OpenCart and TomatoCart with PayPal Express Checkout - Pay Less

In osCommerce the test was generated by the waypoints detour pattern, while in OpenCart and TomatoCart tests were generated by breaking server-generated propagation chains.

In osCommerce, the test is similar to the one shown in Figure 10 while for OpenCart and TomatoCart, the tests are similar to the one in Figure 8. When PayPal Express Checkout is selected, the store and PayPal are exchanging the $Token$ via redirections. Here, the pattern interrupted the chain of $Token$ when the user is redirected to PayPal for the payment. In both cases the oracle verified that the user U_2 had a confirmation and that $paid(U_2, I') \wedge toStore(A)$ is satisfied. However, the oracle could not verify $\mathtt{O}(ack(U_2, I') \wedge onStore(A))$ because it observed $\mathtt{O}(ack(U_2, I) \wedge onStore(A))$.

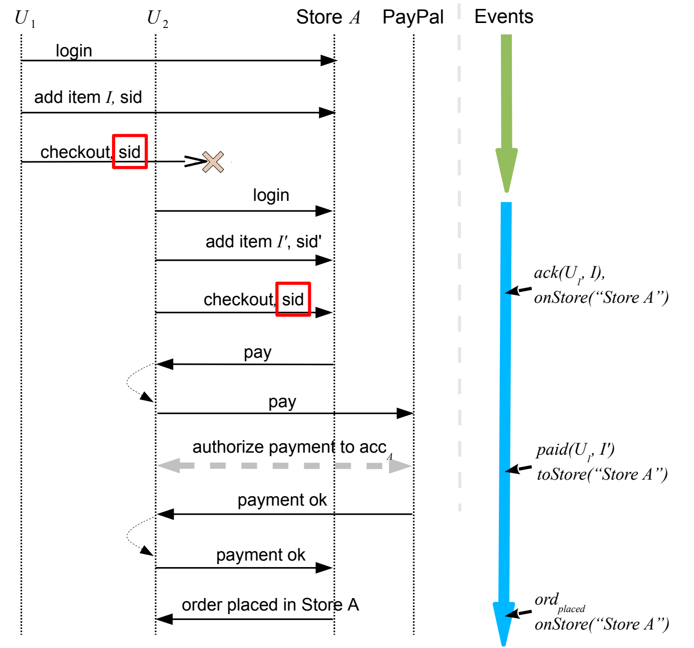


Fig. 11: Session fixation in TomatoCart 1.1.7

A manual inspection confirmed that only the order for I' was in the list of the orders with status "paid", while the order for I was still "payment pending". However, in the balance sheet of the merchant, the payment for I' was done by U_1 instead of U_2. In this case, U_1 authorized PayPal to pay for I while her credit card was charged for I'.

In order to turn this tests into a real attacks, the attacker needs to intercept the redirection URL that is carried over SSL/TLS channels. In addition, it must block the user-victim from executing the redirection. This could require the attacker to either break the SSL/TLS encryption layer or to mount a SSL/TLS MITM (Man-In-The-Middle) attack. However, in both cases the attacker will be able to capture also the payment data of the victim enabling her to shop for free in any case.

5) TomatoCart with PayPal Express Checkout - Session Fixation

Our experiments discovered a session fixation vulnerability in which U_2 could impersonate another user. The test cases were created by breaking the propagation chain of the parameter sid in two points. Figure 11 shows one of them. The events of Figure 11 did not satisfy the formula because the payment I' was of a different amount than the one the user acknowledged for I.

The parameter sid carries the same value in the cookie and breaking it causes a *session fixation* in which, in our case, U_2

results logged in as U_1. From that point on, U_2 can access the data of U_1. As a consequence, U_2 (now logged as U_1) pays the cart of U_1. However, we could not find any exploitation of this flaw. Supposing that the victim (i.e. U_2) "clicks" on an URL crafted by the attacker (U_1), then the victim could notice the fraud in three moments (i) when checking the summary of order, (ii) when providing the shipping address (it shows the attacker's one), and (iii) during the payment because the amount is different.

V. LIMITATIONS

Our approach uses attack patterns that tamper with the observed data flow and workflow. However, it does not test for other types of logic vulnerabilities such as unauthorized access to resources. Moreover, we did not consider cases in which the attacker can also play the role of a malicious store, or the cases in which the attacker can intercept and tamper with the messages between the application and the payment service. We believe that our techniques could also be effective at detecting other kinds of logic flaws, even though we have not experimentally tested this hypothesis. This could be achieved by adding input traces of privileged user (e.g., admin), by adding other behavioral patterns, or by adding new attack patterns.

Second, the test generation favors efficiency over coverage. This means that only a few values are used for each test category, to maximize the possibility to find bugs in a limited amount of time. A more thorough exploration of the attack space could be used to discover more vulnerabilities, however this could require a considerable amount of execution time. The focus of this paper is to show how an automated approach can be used to find logic vulnerabilities in many real-word applications, and not to analyze in depth a single application (a scenario that would also require more input traces to better explore the application's logic).

Finally, we modeled logic properties in LTL. The use of LTL enables us to verify events with time dependency. However, LTL do not support algebra whose terms appear at different moment of the execution. For example, our oracles cannot verify whether the payment is the sum of the items the user added into the cart at some point in the past. There are works that extend LTL with constraints on integer numbers [10], and they could be used by our oracle for checking more fine-grained properties.

VI. RELATED WORK

A large number of solutions have been proposed to detect vulnerabilities in web applications. However, most of the previous work focus on the automated detection of well-known classes of vulnerabilities related to insufficient input validation, such as Cross-Site Scripting (XSS) [26], Cross-Site Request Forgery (CSRF) [2, 27] and SQL injection [22]. Since our goal is to find logic flaws, we will not present these solutions in this section.

a) Detection of Logic Vulnerabilities

When the source code of the application is available, tools such as MiMoSA [9], Waler [21], and Swaddler [16] can be used to discover logic vulnerabilities. MiMoSA and Waler extract a model from the source code and then use a model checker to detect a violation of invariants. Swaddler [16] detects attacks when the software is at the deployment phase of its life-cycle. It first learns the normal behavior of the application and then monitors state variables at runtime looking for deviations from the normal behavior.

When the source code is not available, the problem of extracting a model becomes more difficult. Doupé et al. [19] and Li and Xue [28] proposed two black-box testing tools. The former presents a state-aware input fuzzer to detect XSS and SQLi vulnerabilities. The tool infers a model that is used as an oracle for choosing the next URL to crawl. Both our approach and this technique infer models to improve the automatic detection of vulnerabilities. However, we use a passive learning technique tailored to generate test cases to detect logic flaws, and not an active scanning to drive an input fuzzer. The second work presents BLOCK, a tool that learns model and invariants by observing HTTP conversations and then detects authentication bypass attacks. As opposed to BLOCK our approach does not aim at intercepting attacks, but at generating security tests for detecting flaws. Both works could not be used to find this class of vulnerabilities. The former work proposes a stateful crawler with an input fuzzer that does not attempt to violate the logic of the application. The latter focuses on the detection of authentication bypass attacks by inferring session variable invariants.

An approach similar to BLOCK is InteGuard [37]. InteGuard aims at protecting multi-party web applications from exploitation of vulnerabilities in the API integration. InteGuard focuses mainly on the browser-relayed messages in which data values are exchanged between the parties through the web browser. In particular, InteGuard uses a passive model inference technique based on data-flow analysis and differential analysis to extract inter-services dataflow-related invariants. The former is used to extract the flows of data values while the latter is used to detect properties of data flows such as transaction-specific or implementation-specific values. Our approach uses similar techniques to extract these type of invariants. However, in addition to that, it extracts also invariants of the observable workflow of the application, and takes into account both intra-service invariants, e.g., idempotent operations, and inter-service invariants, e.g., multi-step operations.

Given the limited success of automated black-box techniques, manual methodologies have been recently proposed. Our work is mainly inspired by Wang et al. [34, 35], who presented an analysis of Cashier as a Service (CaaS) based web stores, and a large-scale analysis of web Single Sign-On protocols. The former work describes a black-box methodology that given a number of HTTP conversations, labels API arguments and shows with which ones an attacker could play in the attempt of violating security invariants. The latter refines the previous one by (i) considering the role played by the attacker during the protocol execution and (ii) adding semantic and syntactic labels to protocol parameters. Both techniques

had a large impact due to the severe vulnerabilities the authors were able to find in real-world applications. However, these papers propose techniques and guidelines that need to be manually applied by a security expert. Our work extends their technique in four ways. First, it infers a model from set of HTTP conversations. Second, it generalizes the notion of propagation chain of a single trace into propagation chain of an application model. Third, it infers observable characteristics of the workflow of the business function. Finally, it automatically generates and executes test cases using a number of attack patterns.

AUTHSCAN [8] is an approach similar to our work. It infers a model from implementations combining white-box and black-box techniques. AUTHSCAN focuses on the detection of flaws specific to authentication protocols (See Lowe et al.[29] for a survey of authentication property) and it requires a list of application-specific JavaScript function signatures in order to infer an accurate model of the protocol participants. On the contrary, our approach focuses on business-related web application properties and uses an application-independent model inference technique.

b) Model Inference

There is a large body of works addressing the problem of inferring a model for testing purposes. Model inference is divided in two categories: active learning and passive learning. Active learning techniques interact with the application under inference in order to explore its behavior whereas passive learning techniques build a model from a set of observations. Hossen et al. [25] proposed to apply the active learning algorithm L* [5] to infer a deterministic finite automaton and refining it with testing. Dury et al. [20] described an approach based on passive learning of web-based business applications. They used Parameterized Finite Automaton (PFA) that enriches the classic notion of finite automaton [24] with guards on transitions and parameters on states. PFA models control flow and data flow of an application. Guards are inferred using data mining algorithms like C4.5 [31]. Models are then translated into the Promela language and fed to the model checker SPIN [23] for verifying application-dependent properties. However, in the first approach the authors proposed a direction and say little on the type of flaws they aim at detecting, while in the second the authors focus on the inference part and do not cover the actual testing.

c) Model-Based Security Testing

New ideas have been proposed in order to use models for the (semi-)automatic security testing of web applications when models are available. For example, Armando et al. [7] proposed to detect logic flaws and testing web-based security protocols. The approach consists of using the SAT-based Model Checker [6] to validate a formal specification against security desiderata. If a violation occurs, it is executed against a real implementation. Büchler et al. [13] proposed an approach that assumes (*i*) a model is given (*ii*) and the model is secure. Then they propose to mutate the model by injecting vulnerabilities and to use a model checker for detecting violations. If a problem is found, then they use the counterexample returned by the model checker as an abstract test case for testing implementations. Bodei et al. [11] proposed to model Service-Oriented applications in CaSPiS (Calculus of Services with Pipelines and Sessions), a process calculus with the notion of session and pipelines [12], to perform a control flow analysis for detecting misuse of the application. The authors tested their technique on a known vulnerable version of the CyberOffice shopping cart detecting the price-modification attack. However, for all these works still remains the problem that a model of the application is often not available in practice.

VII. CONCLUSIONS

In this paper we presented a new technique for the black-box detection of logic flaws in web applications. Our approach uses a passive model inference technique that builds a navigation graph from a set of network traces. We then apply a number of heuristics to extract behavioral patterns that are likely related to the underlying application logic. These behaviors, together with a number of attack patterns, are used for generating test cases.

We developed a prototype tool and tested seven E-commerce applications. The prototype generated and executed more than 3100 test cases, 900 of which violated the expected behavior of the application. As a result, our tool detected ten previously-unknown logic vulnerabilities in the applications under test. Five of them allow an attacker to pay less or even shop for free.

ACKNOWLEDGMENT

This work has been partially supported by the European Union Seventh Framework Programme under grant agreement no. 257007 (project SysSec) and no. 257876 (project SPaCIoS Secure Provision and Consumption in the Internet of Services).

REFERENCES

[1] “The google hacking database at hacking for charity.” [Online]. Available: http://www.hackersforcharity.org/ghdb/

[2] “Requestrodeo: Client side protection against session riding,” in *the OWASP Europe 2006 Conference, Report CW448, Departement Computerwetenschappen, KU Leuven, May 2006*, 2006.

[3] “Paypal express checkout integration guide,” August 2012. [Online]. Available: https://cms.paypal.com/cms_content/US/en_US/files/developer/PP_ExpressCheckout_IntegrationGuide.pdf

[4] “Paypal payments standard integration guide,” June 2012. [Online]. Available: https://cms.paypal.com/cms_content/US/en_US/files/developer/PP_WebsitePaymentsStandard_IntegrationGuide.pdf

[5] D. Angluin, “Learning regular sets from queries and counterexamples,” *Inf. Comput.*, vol. 75, no. 2, Nov. 1987.

[6] A. Armando, R. Carbone, and L. Compagna, "Ltl model checking for security protocols," in *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, July 2007, pp. 385–396.

[7] A. Armando, G. Pellegrino, R. Carbone, A. Merlo, and D. Balzarotti, "From model-checking to automated testing of security protocols: Bridging the gap," in *TAP*, ser. LNCS, A. D. Brucker and J. Julliand, Eds., vol. 7305. Springer, 2012.

[8] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, San Diego, California, USA, February 24-27, 2013.

[9] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-module vulnerability analysis of web-based applications," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007.

[10] M. M. Bersani, L. Cavallaro, A. Frigeri, M. Pradella, and M. Rossi, "Smt-based verification of ltl specifications with integer constraints and its application to runtime checking of service substitutability," *CoRR*, vol. abs/1004.2873, 2010.

[11] C. Bodei, L. Brodo, and R. Bruni, "Static detection of logic flaws in service-oriented applications," in *ARSPA-WITS*, ser. LNCS, P. Degano and L. Viganò, Eds., vol. 5511. Springer, 2009.

[12] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti, "Sessions and pipelines for structured service programming," in *FMOODS*, ser. LNCS, G. Barthe and F. S. de Boer, Eds., vol. 5051. Springer, 2008.

[13] M. Büchler, J. Oudinet, and A. Pretschner, "Semi-automatic security testing of web applications from a secure model," in *SERE*. IEEE, 2012.

[14] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering," in *Proceedings of the 16th ACM Conference on Computer and Communication Security*, Chicago, IL, November 2009.

[15] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, ser. SP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 110–125. [Online]. Available: http://dx.doi.org/10.1109/SP.2009.14

[16] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An approach for the anomaly-based detection of state violations in web applications," in *RAID*, ser. LNCS, C. Krügel, R. Lippmann, and A. Clark, Eds., vol. 4637. Springer, 2007.

[17] D. Crockford, "RFC4627: The application/json media type for javascript object notation (json)," July 2006. [Online]. Available: http://tools.ietf.org/html/rfc4627

[18] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the ear: discovering and mitigating execution after redirect vulnerabilities," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011.

[19] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Vulnerability Scanner," in *Proceedings of the 2012 USENIX Security Symposium (USENIX 2012)*, Bellevue, WA, August 2012.

[20] A. Dury, H. H. Hallal, and A. Petrenko, "Inferring behavioural models from traces of business applications," in *Proceedings of the 2009 IEEE International Conference on Web Services*, ser. ICWS '09. Washington, DC, USA: IEEE Computer Society, 2009.

[21] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010.

[22] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[23] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[24] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[25] K. Hossen, R. Groz, and J. Richier, "Security vulnerabilities detection using model inference for applications and security protocols," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, march 2011.

[26] M. Johns, "Code injection vulnerabilities in web applications: Exemplified at cross-site scripting," Ph.D. dissertation, 2011.

[27] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing cross site request forgery attacks," in *SecureComm*. IEEE, 2006.

[28] X. Li and Y. Xue, "Block: a black-box approach for detection of state violation attacks towards web applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011.

[29] G. Lowe, "A hierarchy of authentication specifications," in *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, 1997, pp. 31–43.

[30] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE Computer Society, 1977.

[31] J. R. Quinlan, *C4.5: programs for machine learning*. San

Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[32] The jQuery Foundation, “jQuery,” January 2013. [Online]. Available: http://jquery.com/

[33] The OWASP Foundation, “OWASP Testing Guide,” December 2008. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Project

[34] 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 the 2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012.

[35] 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 the 2011 IEEE Symposium on Security and Privacy*, ser. SP ’11. Washington, DC, USA: IEEE Computer Society, 2011.

[36] World Wide Web Consortium, “Simple Object Access Protocol (SOAP) 1.2,” April 2007. [Online]. Available: http://www.w3.org/TR/soap/

[37] L. Xing, Y. Chen, X. Wang, and S. Chen, “Integuard: Toward automatic protection of third-party web service integrations,” in *20th Annual Network and Distributed System Security Symposium, NDSS 2013*, San Diego, California, USA, February 24-27, 2013.