Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs

Giancarlo Pellegrino
CISPA, Saarland University
Saarland Informatics Campus
gpellegrino@cispa.saarland

Martin Johns
SAP SE
martin.johns@sap.com

Simon Koch
CISPA, Saarland University
Saarland Informatics Campus
s9sikoch@stud.uni-saarland.de

Michael Backes
CISPA, Saarland University
Saarland Informatics Campus
backes@cispa.saarland

Christian Rossow
CISPA, Saarland University
Saarland Informatics Campus
rossow@cispa.saarland

ABSTRACT

Cross-Site Request Forgery (CSRF) vulnerabilities are a severe class of web vulnerabilities that have received only marginal attention from the research and security testing communities. While much effort has been spent on countermeasures and detection of XSS and SQLi, to date, the detection of CSRF vulnerabilities is still performed predominantly manually.

In this paper, we present Deemon, to the best of our knowledge the first automated security testing framework to discover CSRF vulnerabilities. Our approach is based on a new modeling paradigm which captures multiple aspects of web applications, including execution traces, data flows, and architecture tiers in a unified, comprehensive property graph. We present the paradigm and show how a concrete model can be built automatically using dynamic traces. Then, using graph traversals, we mine for potentially vulnerable operations. Using the information captured in the model, our approach then automatically creates and conducts security tests, to practically validate the found CSRF issues. We evaluate the effectiveness of Deemon with 10 popular open source web applications. Our experiments uncovered 14 previously unknown CSRF vulnerabilities that can be exploited, for instance, to take over user accounts or entire websites.

1 INTRODUCTION

No other vulnerability class illustrates the fundamental flaws of the web platform better than Cross-Site Request Forgery (CSRF): Even a brief visit to an untrusted website can cause the victim's browser to perform authenticated, security-sensitive operations at an unrelated, vulnerable web application, without the victim's awareness or consent. To achieve this, it is sufficient to create a single cross-origin HTTP request from the attacker webpage, a capability that is native to the Web ever since Marc Andreessen introduced the `img` HTML tag element in February 1993 [2].

Since its discovery in 2001 [36], CSRF vulnerabilities have been continuosly ranked as one of the top three security risks for web applications, along with cross-site scripting (XSS) and SQL injection (SQLi) [6, 11, 31]. Successful CSRF exploitations can result in illicit money transfers [43], user account takeover [38], or remote server-side command execution [19], to name only a few publicly documented cases. In the past, similar vulnerabilities have been discovered in many popular websites including Gmail [34], Netflix [12], ING Direct [43], and, more recently, in Google, Skype, and Ali Express websites [38].

Despite its popularity, CSRF has received only marginal attention, compared to SQLi and XSS. Most of the previous efforts have been spent in proposing active [20, 21, 24] or passive [6] defense mechanisms, and little has been done to provide developers and practitioners with effective techniques to *detect* this class of vulnerabilities. Classical vulnerability detection techniques utilize dynamic [4, 10, 32, 33] and static analysis techniques [3, 9, 18, 28, 39], while mainly focusing on injection vulnerabilities [9, 10, 18] or flaws specific to the application logic layer [10, 28, 32, 39]. Unfortunately, none of the existing techniques are easily applicable to CSRF. As a result, to date, CSRF vulnerabilities are still predominately discovered by manual inspection [38].

研究sqli和xss的文章

Our Approach—We take a step forward by presenting Deemon, a model-based security testing framework to enable the detection of CSRF vulnerabilities. To the best of our knowledge, this is the first automated technique that targets the detection of CSRF. Deemon automatically augments the execution environment of a web application, to enable the unsupervised generation of dynamic *execution traces*, in the form of, e.g., network interaction, server-side execution, and database operations. Using these traces, Deemon infers a *property graph*-based model of the web application capturing different aspects such as state transitions and data flow models in a unified representation. Operating on the resulting model, Deemon uses graph traversals to identify security-relevant state-changing HTTP requests, which represent CSRF vulnerability candidates. Finally, leveraging the augmented application runtime, Deemon validates the candidate's vulnerability against the real web applications.

We assessed Deemon against 10 popular open source web applications and discovered 14 previously-unkown CSRF vulnerabilities in four of them. These vulnerabilities can be exploited to take over websites, user accounts, and compromise the integrity of a database. Finally, we analyzed our test results to assess the current awareness level of the CSRF vulnerabilities. In two cases, we identified alarming behaviors in which security-sensitive operations are protected in a too-selective manner.

To summarize, we make the following contributions:

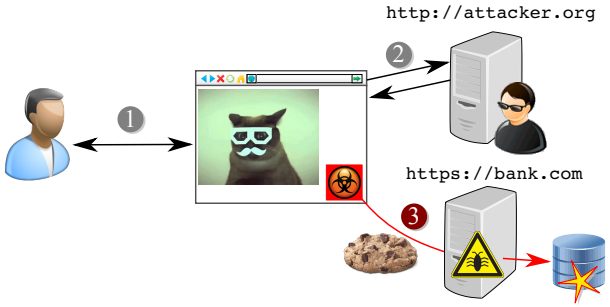


Figure 1: Authenticated CSRF attack.

- We present Deemon, an automated, dynamic analysis, security testing technique to detect CSRF vulnerabilities in productive web applications;
- We present a new modeling paradigm based on *property graphs*, that is at the core of Deemon;
- We show how Deemon's models can be instantiated in an unsupervised, automatic fashion, requiring only selected GUI interaction recordings;
- We report on a practical evaluation of Deemon using 10 popular web applications, which uncovered 14 severe CSRF vulnerabilities; and
- We assess the CSRF awareness level and discover alarming behaviors in which security-sensitive operations are protected in a selective manner.

2 CROSS-SITE REQUEST FORGERY (CSRF)

In CSRF attacks, an attacker tricks the web browser of the victim to send a request to a vulnerable honest website in order to cause a desired, security-sensitive action, without the victim's awareness or consent. Desired actions can be, for example, illicit money transfers [43], resetting account usernames [38], or the execution of specific server-side commands [19]. CSRF attacks can be distinguished into two main categories: authenticated and login CSRF. In an *authenticated* CSRF (aCSRF), a pre-established, authenticated user session between the victim's web browser and the targeted web application exists. In a *login* CSRF, such a relationship does not exist, but the goal of the attacker is to log the victim in by using the attacker's credentials. In the remainder of this paper, we focus on aCSRF attacks, the significantly larger category. An extensive overview of login CSRF is provided by Sudhodanan et al. [38].

Figure 1 shows an example of an aCSRF attack. The actors of an aCSRF attack are the user (i.e., the victim), a vulnerable target website (e.g., `bank.com`, a home banking website), and an attacker controlling a website (e.g., `attacker.org`). In an aCSRF attack, the victim is already authenticated with the target website. Upon a successful authentication, the website of the bank persists an authenticated session cookie in the user's web browser. From this point on, whenever the user visits the website of the bank, the browser includes this session cookie [5]. An attacker can exploit this behavior of the browser as follows. First, she prepares an HTML page containing malicious code. The goal of this code is to perform a *cross-origin* HTTP request to the website of the bank. This can be implemented in different ways, e.g., with an HTML `iframe` tag, a hidden HTML form with self-submitting JavaScript code, or via the XMLHttpRequest JavaScript API [40]. Then, when a victim visits the malicious page, her browser generates such a request, which automatically includes the the session cookie. The bank checks the cookie, and executes the required operation. If the HTTP request encodes, e.g., a request to update user password, then the bank executes it without the actual consent of the bank account owner.

More formally, we define an aCSRF vulnerability as follows.

Definition 1. A web application (e.g., bank.com) exposes an aCSRF vulnerability, if the web application accepts an HTTP request (e.g., message 3) with the following properties:

(P1) The incoming request causes a security-relevant state change of the web application.

(P2) The request can be reliably created by an attacker, i.e., the attacker knows all the required parameters and values of the request.

(P3) The request is processed within a valid authentication context of a user.

Cross-origin requests can be used in other attacks without necessarily causing a server-side state transition, e.g., accessing user data stored in the target website. These attacks are addressed by the *same-origin policy* (SOP) [5] for cross-origin requests, which blocks the access to HTTP responses. However, the SOP does *not* prevent the browser from performing HTTP requests. To defend against malicious cross-origin requests, the server-side program can check the request origin via the header `Origin`. However, this header may not be present in a request. The current best-practice aCSRF protection is the so-called *anti-CSRF token* [6]. An anti-CSRF token is a pseudo-random value that is created by the server and explicitly integrated into the request by the client. Various methods exist to implement anti-CSRF tokens, including hidden form fields or custom HTTP headers. Further implementation details are left out of this document for brevity.

3 CHALLENGES IN DETECTING ACSRF

A security testing approach designed to detect aCSRF vulnerabilities faces two distinct classes of challenges, neither of them met by the current state-of-the-art in security testing: *detection challenges* and *operational challenges*, as discussed next.

3.1 Detection Challenges

Detecting aCSRF requires reasoning over the relationship between the application state, the roles and status of request parameters, and the observed sequences of state transitions. This leads to a set of specific detection challenges that directly result from the unique characteristics of the vulnerability class.

(C1) State Transitions—The first challenge is to determine when a state transition occurs. Server-side programs implement several operations; not all of them affect the state of the application. Consider, for instance, the function of searching for a product in an online store: The user provides search criteria, causing the server-side program to search its database for matching products. The permanent state of the user's data in the application is unaffected by this process. However, other operations change the state of the program.

Consider a user that wants to change their login password. The server-side program uses the new password to update the database entry. From that point on, the old password is no longer accepted; thus, the state has changed.

Existing tools such as *web application scanners* (See, e.g., [11, 23]) mainly operate in a black-box manner. They crawl a web application and send requests with crafted input. Vulnerabilities are detected by inspecting responses. This approach works well with XSS and SQLi, but does not scale to CSRF as it cannot discern when a request changes the server-side state. Web crawlers can be made aware of server-side states by inferring a model capturing transitions via webpage comparisons: If the HTML content is similar, then they originate from the same state (See, e.g., Doupé et al. [10]). However, as pages contain dynamic content, the similarity may not be determined precisely, thus resulting in inaccurate models. Finally, techniques to infer models are often specific to the function being tested (See, e.g., [32, 41]). aCSRF vulnerabilities can affect any function of a web application; thus, function-specific models cannot be easily used to detect aCSRF vulnerabilities.

(C2) Security-Relevant State Changes—The second challenge is to determine the relevance of a state transition. State transitions can be the result of operations such as event logging and tracing user activity. These operations indeed change the state of the server, but they are not necessarily security relevant. While a human may distinguish the two cases, automated tools without a proper description of the application logic may not tell the two transitions apart. Especially for static analysis approaches, security-neutral state changes are indistinguishable from aCSRF candidates.

(C3) Relationships of Request Parameters and State Transitions—The third challenge consists in determining the relations between request parameters and state transitions. The identification of these relations is relevant for the detection of aCSRF vulnerabilities. For example, consider a parameter carrying a random security token. An attacker may not be able to guess such a parameter, thus preventing her from reconstructing the HTTP request. The identification of these parameters is important, as it suggests the presence of anti-CSRF countermeasures, and can be used to develop a testing strategy. For example, the tester may replay the request without the token to verify whether the web application properly enforces the use of the security token. Another example is a parameter carrying a user input, e.g., a new user password, that is stored in the database. An attacker can use this parameter to hijack a user account by using a password that she controls.

Existing techniques do not determine the relations between parameters and state transitions. Web scanners attempt to identify security tokens by matching parameter names against a predefined list of patterns, e.g., the parameter being called `token`. In general, to determine the role of a request parameter, we need to determine the type of relations with state transitions. As these parameter values traverse the tiers of an application, we may need to track their flow across all tiers, e.g., presentation, logic, and data. The resulting model of data flows can be enriched with type information, e.g., both semantic and syntactic types, to determine the nature of the value, e.g., user-controlled or pseudo-random.

3.2 Operational Challenges

The operational challenges in detecting aCSRF are direct consequences of addressing the *detection challenges* in the context of dynamic security testing.

(C4) Transitions in Non-Trivial Application Workflows—The fourth challenge is to reach state-changing requests in non-trivial web application workflows. Dynamic analysis techniques such as unsupervised web scanners explore HTML webpages using breadth- or depth-first search algorithms. However, these algorithms are too simplistic to cope with the complexity of modern web application workflows in which users need to perform a specific sequence of actions. Likewise, static analysis techniques look for patterns in the source code to determine the presence of a vulnerability. However, without a proper description of the workflow, static approaches scale poorly to large applications.

(C5) Side-Effect-Free Testing—Dynamic testing for aCSRF vulnerabilities is centered around the iterative detection of state-changing HTTP requests (Challenges C1 & C2). However, as such requests indeed *change* the application state, all further test requests attempting to assess the relationships of request parameters and state transitions (C3) will most likely operate on a now-invalid state. Take for example the dynamic testing for aCSRF vulnerabilities in a shopping cart web application. As soon as a test request has submitted the cart beyond the check-out state, no further security testing on this state transition can be conducted, as the active shopping cart ceases to exist. Thus, a testing method is needed, that allows evaluation of HTTP request-induced state changes in a side-effect-free manner.

(C6) Comprehensive, Reusable Representation of Application Functionality—The final challenge results from the previous challenges. To detect security-relevant state changes, we need to combine aspects of the web application. On the one hand, we have transitions describing the evolution of the internal states of the server-side program. On the other hand, we have data flow information capturing the propagation of data items across tiers and states. These aspects can be represented by means of *models*.

In literature, there are many languages and representations to specify models, ranging from formal languages [13] to custom models tailored to the specific application function being tested (e.g., [32, 41]). Often, the combination of models has been addressed in a custom way. The shortcoming of this approach is that the combination is achieved without specifying the relationships between the models, thus making it hard to reuse it for other techniques. Another approach is to create representations that combine elements of individual models, such as extended finite-state machines that fire transitions when certain input conditions hold [13]. However, defining new modeling languages may not scale well, as a new language is required as soon as new aspects need to be included.

4 DEEMON: OVERVIEW

To overcome the challenges of Section 3, we developed `Deemon`[1], an application-agnostic, automated framework designed to be used by developers and security analysts during the security testing

[1]Source code and documentation of `Deemon` can be downloaded here https://github.com/tgianko/deemon

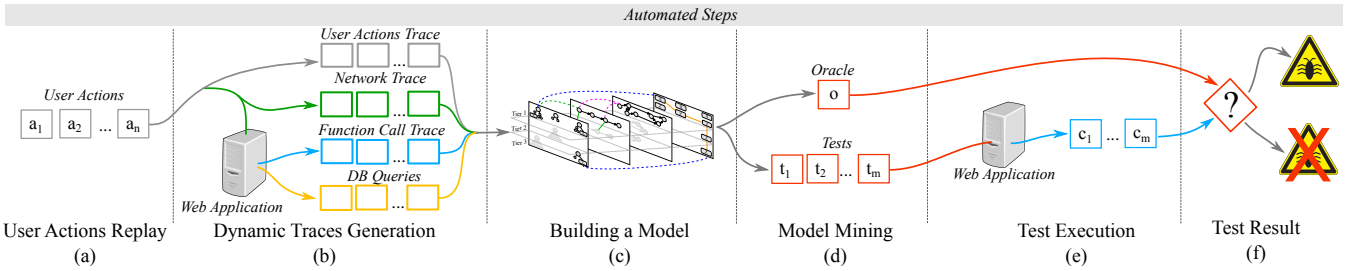


Figure 2: Overview of the detection phase of Deemon.

phase of the software development life-cycle. The current version of Deemon supports PHP-based web applications that use MySQL databases, and it can be easily extended to support other languages and databases. The key features of Deemon that allow for addressing our challenges are the following:

- Deemon infers models from program execution observations capturing state transitions and data flow information (Challenges C1 & C3).
- Deemon uses *property graphs* to represent these models. This provides a uniform and reusable representation and defines precise relationships between models by the means of *labeled edges* (Challenge C6).
- Deemon leverages a programmatic access to the property graph via *graph traversals* to identify security-relevant state changes (Challenge C2).
- Deemon augments the execution environment of a web application and then reproduces a set of user actions to observe server-side program execution (Challenge C4).
- Deemon relies on virtualized environments to test web applications. This enables full control of the web application by taking and restoring *snapshots* (Challenge C5);

Deemon takes as input a set of user actions and an application container of the web application under test. Deemon operates in phases: *instrumentation* and *detection*. In the first phase, Deemon modifies the application container to insert sensors for the extraction of network traces, server-side program execution traces, and sequence of database operations. In the second phase, Deemon automatically reproduces user actions, infers a model from the resulting traces, and tests the web application to detect aCSRF vulnerabilities.

4.1 Preparation

Deemon is meant to support developers and security analysts. In this section, we briefly present the tool as seen by a user.

Inputs—The inputs of Deemon are a set of user actions and an application container of the web application under test.

User Actions: The first input is a set of user action sequences (see Figure 2.a) that are provided by the tester. User actions are artifacts commonly used in security testing [30] and there is a plethora of automated tools to create them via web browsers and use them when testing web applications [30]. A user action is performed on the UI of the web application. For example, a user action can be a mouse click, a key stroke, or an HTML form submission. The sequence of actions represent a web application functionality. For example, consider the operation of resetting user credentials. The user actions trace contains the following actions: *load* `index.php` page, *click* on change credential link, *type* new username and password, and *click* submit. Input traces can also be actions of a privilegded user, e.g., website administrator, when changing the website configuration from the administrator panel.

Application Container: The second input of Deemon is an application container of the web application under test. An application container consists of a runtime environment with software, dependencies and configuration. Web application containers contain the web application (binary or source code), database server, and application configuration. Containers are convenient tools as they allow the deployment of ready-to-use web applications. Nowadays application containers are gaining momentum and are becoming a popular means to distribute and deploy web applications.

Outputs—Deemon returns a vulnerability report, listing state-changing HTTP requests that can be used to perform aCSRF attacks.

4.2 Instrumentation

Given an application container, Deemon automatically installs sensors to monitor the program execution. For example, for PHP-based web applications, Deemon adds and enables the *Xdebug* [35] module of the PHP interpreter, an extension that generates full function call trees. Furthermore, Deemon installs a local HTTP proxy to intercept HTTP messages exchanged between the server and the browser.

4.3 Detection

The core function of Deemon is the detection of aCSRF vulnerabilities. The main steps are shown in Figure 2 and are all automated. The detection begins by reproducing the user actions against a running instance of the web application (Figure 2.a). The sensors installed during the instrumentation produce execution traces that include network traces and function call traces (Figure 2.b). Deemon runs this step twice to observe, for example, sources of non-determinism such as generation of pseudo-random data items. Each run is called *session*. From these traces, Deemon infers a model which is the composition of simpler models, e.g., finite-state machine and data flow model with data type information (Figure 2.c). Then, Deemon uses model queries to mine both security tests and an oracle (Figure 2.d), and runs them against the web application (Figure 2.e). Finally, it evaluates test results against the oracle to detect CSRF vulnerabilities (Figure 2.f).

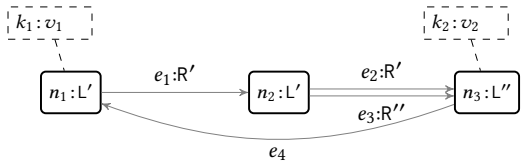
5 MODELING

The overall goal of our modeling approach is to create a representation of a web application that can address challenges C1-3 and C6. Challenge C1 requires obtaining an adequate model that allows determining when a change of state occurs. We address this challenge by building a finite-state machine (FSM) from execution traces captured by our probes. Challenge C2 consists in determining which state transitions are security-relevant. We observe that security-relevant transitions are likely to occur less frequently than other transitions. From this observation, we derive state invariants based on frequency. Challenge C3 consists in determining the relationship between request parameters and state transitions. In particular, we are interested in identifying two types of HTTP parameters: parameters carrying unguessable tokens and parameters carrying user input. We address this challenge by using a data flow model (DFM) with types (see [41]). The DFM represents a state as a set of variables and can capture the propagation of data items from HTTP requests to the SQL query. Each data item can have syntactic types, e.g., string, integer, boolean, and semantic types, e.g., constant, unique, user input. We use types to identify tokens and user-generated inputs. Finally, we need a representation for our models that can support (i) the creation of a model with inference algorithms and (ii) the identification of security-relevant transitions. To address this challenge, i.e., C6, we map models into *labeled property graphs* and use *graph traversals* to query them.

This section details the building blocks of our modeling approach. In Section 5.1, we present property graphs, the mapping of models to graphs, and elementary graph traversals. In Section 5.2, we present the construction of a property graph.

5.1 Labeled Property Graph

A labeled property graph is a directed graph in which nodes and edges can have labels and a set of key-value properties. An example of a labeled property graph is shown below.



This example shows three nodes. Nodes n_1 and n_3 have one property each, i.e., $k_1 = v_1$ for n_1 and $k_2 = v_2$ for n_3. Nodes have labels. For example, nodes n_1 and n_2 are labeled with L′ whereas node n_3 is labeled with L″. Edges are also labeled. The edges e_1 and e_2 are labeled R′, and edge e_3 is labeled R″.

5.1.1 Mapping Models to Property Graphs. We now present the mapping of traces, FSM and DFM to a property graph. Figure 3 shows the operation of updating the user password as a property graph. This example covers the logic and data tiers of a web application. For the sake of readability, user actions are not shown.

Traces and Parse Trees—In our approach, traces and parse trees are important artifacts that are used throughout the analysis. First, traces and parse trees are the input of the inference algorithms to generate FSMs and DFMs. Second, traces are used to derive state invariants, e.g., the number of distinct HTTP requests triggering the same state transition. Third, parse trees are used for the generation of tests to detect aCSRF vulnerabilities. Accordingly, we decided to include them in the property graph.

A trace is a sequence of events observed by our sensors, e.g., HTTP messages or SQL queries. We represent an event with a node of label `Event`. We chain events using edges with label `next`. Parse trees represent the content of a trace event. For example, with reference to Figure 3.d, the event e' is the following HTTP request:

```
POST /change_pwd.php  HTTP/1.1
Host: bank.com
Cookie: SESSION=X4a
Content-Length: 15
Content-Type: application/x-www-form-urlencoded

password=pwnd
```

We parse HTTP requests and store the resulting parse tree in the property graph. An example of a parse tree for the example is shown in Figure 3.c.i. For simplicity, Figure 3.c.i does not show the `Host`, `Content-Type`, and `Content-Length` HTTP headers. We map parse trees into a property graph as follows. Parse trees have three labels: `Root`, `NTerm`, and `Term`. The `Root` node label is used for the root of a parse tree. The `NTerm` node is used for non-terminal nodes of the parse tree, whereas `Term` is for the terminal nodes. Nodes are connected using the `child` edge label.

Finite State Machines—We use FSMs to represent program states and transitions between states. Our goal is the identification of state transitions triggered by an HTTP request. Accordingly, we use HTTP requests as the symbols accepted by a transition. However, in our model, HTTP requests are represented as nodes, and property graphs do not support edges between a node, e.g., an HTTP request, and an edge, e.g., a transition. As a result, we model a transition between two states as nodes with three edges. The first edge is directed to the node representing the accepted HTTP request. The second edge is from the initial state of the transition to the transition node. The third edge is directed to the new state. The mapping of FSM elements to nodes, edges, and labels is shown in Table 1.

Dataflow Information and Types—To determine the relationship between request parameters and state changing operations, we use dataflow models (DFMs) with types as presented by Wang et al. [41]. The data flow model was originally designed to enrich HTTP request parameters with abstract types such as syntactic and semantic tables. Consider an HTTP request with a parameter `password=pwnd` with the value `pwnd` provided by the user. The DFM associates the parameter `password` with a syntactic label, e.g., `string`, and semantic labels, for example, user-generated (`UG`). In our graph, we represent a DFM as a set of variables. A variable is a node graph with a name (e.g., parameter name), a value (e.g., parameter value), and a type (e.g., semantic and syntactic type). Variables can carry the same data item. In these cases, we say that there is a propagation of data values. The rules that determine whether a propagation exists are presented in Section 5.2.

An example of a DFM is shown in Figure 3.a. This DFM comprises four variables, two for HTTP request parameters, i.e., session cookie and password parameter, and two for the SQL WHERE and SET clauses. Each variable has a type. For example, variable v_1 has semantic type `SU`, which means that the value is different for each

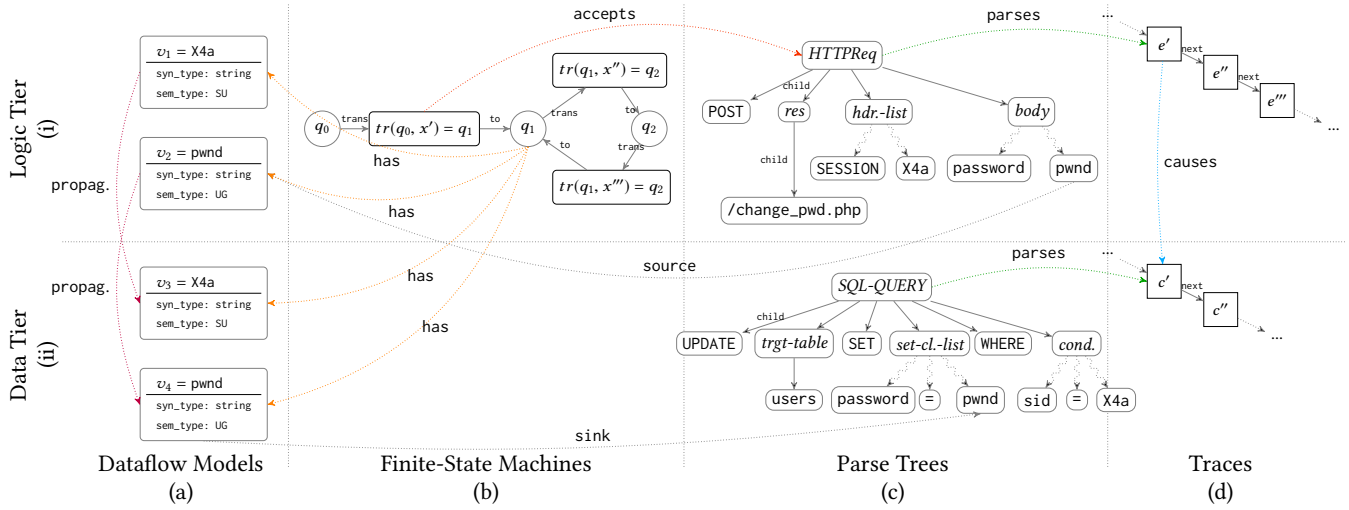


Figure 3: Excerpt of property graphs for a model showing two tiers (logic and data).

user session, whereas varuable v_2 has type UG. We represent the propagation of data items with a source, a propagation chain and a sink. For this, we use three types of edges, source, propag., and sink. Figure 3 shows the complete propagation chain for the pwnd data item. Finally, DFM variables are linked to FSM states with has edges. This link determines the relationship between request parameters and state-changing operations.

5.1.2 Relationships. The elements of our graph have relationships. Consider, for example, a parse tree that represents the HTTP request causing a state transition. Our framework defines a set of relationships between these elements. We now briefly present these relationships. The mapping of these relationships into a property graph is shown in Table 2.

Dataflow Information—This relationship connects a DFM to a FSM, or a DFM to a parse tree. In the first case, the variable can be used to determine the state of a FSM. We model this relationship with an edge from a state to a variable. In the second case, a variable carries values from a source, e.g., HTTP parameters, or values used to create a query.

Data Propagation—This relationship captures the propagation of data items during the execution of a program. In our model, this relationship is between two DFMs and represents the propagation of data items across the tiers of a web application. For example, consider a data value that is first provided with a user action; then the value is included in an HTTP request; and, finally, it is inserted in a SQL query to be stored in the database.

Abstractions—Abstractions represent the link between an abstract element and its concrete counterpart. Abstractions are an expedient to reduce the complexity of a problem or to focus the analysis on relevant parts. For example, abstractions remove variable parts such as data values from SQL queries. The resulting abstract SQL query is then compared with other abstract queries to group them. This expedient is used by our model inference algorithms and we present abstractions in Section 5.2.

Event Causality—This relationship can occur, for example, between a user click on a link and the resulting HTTP request. Our sensors can establish this type of relationship.

Accepted Inputs—This relationship captures the connection between HTTP requests and state transitions. Iff HTTP requests cause a transition, we say that the FSM accepts the HTTP request.

5.1.3 Graph Traversals. Graph traversals are the means to retrieve information from property graphs. They allow querying a graph based on nodes, edges, and properties. Deemon uses traversals written in the Cypher query language [29], a graph query language supported by popular graph databases such as Neo4j. The Cypher language follows a declarative approach in which each query describes *what* we want to retrieve and not *how*. The *what* is specified with *graph patterns*, a description of a subgraph using nodes, edges, labels, and properties. Deemon uses graph queries for the creation of FSM and DFS (See Section 5.2) and to generate tests for the detection of aCSRF (See Section 6).

For the sake of readability, we do not present the Cypher syntax but a simplified notation that retains the declarative approach. We use sets of nodes and edges to represent Cypher queries. For example, a query Q can be defined as all nodes n in the property graph for which a given predicate p is true, i.e., $Q = \{n : p(n)\}$. In our notation, the predicate p is the graph pattern. We use parametric logic predicates for graph patterns. In the following, we present elementary graph patterns that allow establishing a basic language to operate with the property graph.

We start with an example to show elementary queries to retrieve nodes and edges via labels. These queries are generic and are not tied to our framework.

Example 5.1 (Elementary Queries). To create queries, we first define the graph pattern. Then, we use the predicate to define a set. The first elementary pattern is true iff a node has a given label L:

$$\texttt{Label}_{\texttt{L}}(n) \stackrel{def}{:=} \text{“}n : \texttt{L}\text{”}$$

Component	Node label(s)	Relationship(s)
FSM	State, StateTrans	$q \xrightarrow{\text{trans}} t$, $t \xrightarrow{\text{to}} q$, $t \xrightarrow{\text{accept}} q$
DFM	Variable	$v' \xrightarrow{\text{propagat}} v''$
Trace	Event	$e' \xrightarrow{\text{next}} e''$
Parse tree	Root, NTerm, Term	$n \xrightarrow{\text{child}} m$

Table 1: List of nodes and edges for our models.

Name	Mapping into a Property Graph
Data Flow Inform.	v : State $\xrightarrow{\text{has}}$ q : Variable
Data Propagation	v_1 : Variable $\xrightarrow{\text{propag.}}$ v_2 : Variable or t : Term
Abstractions	apt : Root $\xrightarrow{\text{abstracts}}$ pt : Root, ae : Event $\xrightarrow{\text{abstracts}}$ e : Event
Event Causality	e_1 : Event $\xrightarrow{\text{causes}}$ e_2 : Event
Accepted Inputs	st : StateTrans $\xrightarrow{\text{accepts}}$ pt : Root

Table 2: List of relationships between models.

The second example pattern is true iff a graph edge has a given label $\mathcal{R}$:

$$\texttt{Label}_{\mathcal{R}}(n, m) \stackrel{def}{:=} \text{“}e = (n, m) \wedge e : \mathcal{R}\text{”}$$

These predicates can be used to define queries. For example, to find all nodes with label L we can write the following query:

$$\texttt{Q}_{\texttt{label}} = \{n : \texttt{Label}_{\texttt{L}}(n)\}$$

As graph patterns may have more than one parameter, we can use quantifiers (i.e., $\forall$ or $\exists$) to broaden or limit the scope of a query. For example, consider the query to retrieve *all* nodes with an outgoing edge R, we can use the following query:

$$\texttt{Q}_{\texttt{out}} = \{n : \forall m, \texttt{Label}_{R}(n, m)\}$$

From these elementary patterns and queries, we create a basic query language that can express elements of our models.

Example 5.2 (Queries for Models). Consider the example of retrieving the states of a FSM. First, we define a predicate for the pattern, called $\texttt{State}(q)$, that is defined as $\texttt{Label}_{\texttt{State}}(n)$. Then, we use this pattern in a query that searches for all states q:

$$\texttt{Q}_{\texttt{States}} \stackrel{def}{:=} \{q : \texttt{State}(q)\}$$

We create similar patterns for relationships. For example, with reference to Figure 3, consider the graph pattern between the state q_0 and q_1. We can call this pattern $\texttt{Trans}(q_0, t, q_1)$ and we define it as $\texttt{Label}_{\texttt{trans}}(q_0, t) \wedge \texttt{Label}_{\texttt{to}}(t, q_1)$.

In a similar way, we create patterns for all nodes and edges in Table 1 and in Table 2. We also create patterns using properties. For example, $\texttt{HTTPReq}(pt)$ is a pattern for a Root node pt whose property t = HttpReq. This gives us a basic language to operate with our models.

The notation of these two examples adheres to the declarative approach followed by Cypher. The actual search of all nodes matching the predicates used in the set definition is performed by the query processor. The query processor is a graph database component that transforms declarative queries into a sequence of operations to traverse the graph and search for all matching nodes.

5.2 Model Construction

After having presented the building blocks of our modeling approach, we present the construction of our model. The first step of the construction consists in importing traces and parse trees in the property graph. Then, we use inference algorithms to create FSMs and DFMs.

5.2.1 Importing Traces and Parse Trees.

We import traces and parse trees in the following order:

User Actions—We first import user actions traces. For each element of the trace, we create a node Event. If two events are consecutive in a trace, then we place an edge next between the two nodes. Then, we parse the user action into the three main elements: the type of action (e.g., mouse click or key stroke), the UI element on which the action is performed (e.g., HTML element), and, if present, the user input (e.g., username). Then, we connect the root node of the parse tree to the trace node with a parses edge. To distinguish user action events from other events (i.e., HTTP messages), we add a node property t to UA which stands for user action. Finally, we add a node property for the user performing these actions. For example, if the user actions are performed by an administrator, we add the property user = admin.

HTTP Messages—First, we import a trace as seen for user actions. Second, for each HTTP message, we create parse trees for HTTP requests, responses, URLs, cookies, HTTP POST data, and JSON objects. We link the root with the event with a parses relationship. Then, we link the HTTP messages to network events with parses edges, and causes edges between user actions and HTTP request events. The property t is set to HTTPReq. Finally, as described in Section 4.3, Deemon reproduces user actions twice, thus generating two HTTP message traces, i.e., sessions, which can be different due to newly generated cookies or anti-CSRF tokens. When importing traces, we add the trace session number as a node property.

Database Queries—We parse the call trees to extract calls to database APIs and retrieve SQL queries. We add a parses relationship between the parse trees and the trace event. Then, we add causality edges between HTTP request events and the resulting query events. Similarly as for HTTP messages, we add the trace session number as a node property. Finally, the property t is set to SQL.

5.2.2 Finite-State Machines.

After importing traces and creating parse trees, we construct the FSM.

Abstract Parse Trees—The rule to build a FSM is the following: A state transition occurs when similar HTTP requests cause similar SQL queries. Similarity between HTTP requests and queries is achieved by the means of *abstract parse trees*, i.e., parse trees that omit a few selected terminal nodes. For HTTP requests, we neglect URL parameter values and POST data values. For SQL queries, we neglect terminal nodes at the right-hand side of SQL comparison operations. Figure 4 shows the parse tree of an HTTP request to update a user password and an abstract parse tree in which terminal

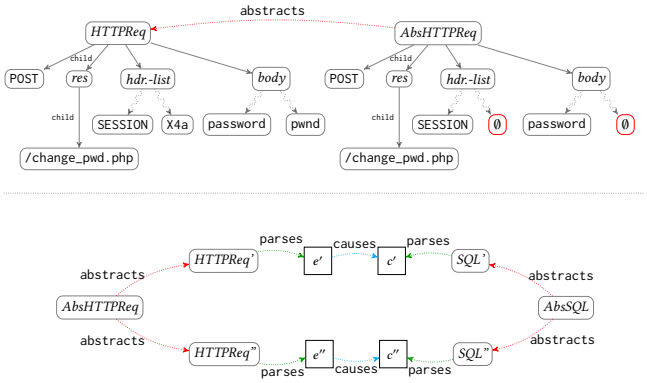


Figure 4: On top: abstract relationships between a parse tree and an abstract one. Below: visualization of the graph pattern to identify transitions.

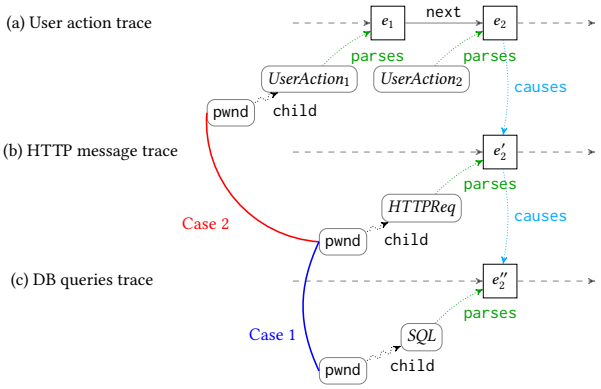


Figure 5: Example of propagation along causality edges (Case 1) and backward propagation chain (Case 1).

nodes were neglected. Abstract parse trees are unique. If two parse trees result in the same abstract tree, we place two edges `abstracts` from the abstract parse tree to the two parse trees.

Clustering—After the creation of abstract parse trees, we extract HTTP requests triggering the same transition from the graph . Figure 4 exemplifies this situation, showing the roots of parse trees and trace events. Two requests, e.g., the roots $HTTPReq'$ and $HTTPReq''$, trigger the same transition if (i) the HTTP requests have the same abstract parse tree, i.e., with root $AbsHTTPReq$, (ii) the HTTP requests cause SQL queries, i.e., parse tree roots SQL' and SQL'', via a causality edge, and (iii) the SQL queries have the same abstract parse tree, i.e., $AbsSQL$. HTTP requests matching this description can be found with this query:

$$\begin{aligned} Q_{\text{Aux}} \stackrel{def}{:=} \{(abs'_h, h', abs'_{sql}, sql') : \exists e', c', \texttt{Abs}(abs'_h, h') \wedge \\ \texttt{Parses}(h', e') \wedge \texttt{Causes}(e', c') \wedge \\ \texttt{Parses}(sql', c') \wedge \texttt{Abs}(abs'_{sql}, sql')\} \end{aligned}$$

This query returns a set of 4-tuples. For example, with reference to Figure 4, this query returns two 4-tuples: the first with $AbsHTTPReq'$, $HTTPReq'$, $AbsSQL'$, and SQL', and the second with $AbsHTTPReq''$, $HTTPReq''$, $AbsSQL''$, SQL''. If we group these tuples by abstract HTTP request and abstract SQL query, the resulting groups represent transitions satisfying our rule. The HTTP requests in each group are the symbols causing the state transition.

FSM—To create a FSM, we create one state node for each edge `next`, and a transition for each HTTP request. Then, we minimize the FSM using the clustering algorithm [16].

5.2.3 Dataflow Model and Information.

Finally, we construct the data flow model with types.

Variables—Variables are derived from terminal nodes in parse trees. The terminal nodes are the same ones neglected in abstract parse trees. The value of the variable is the symbol of the terminal node, whereas the variable name is the path of the terminal node from the root. Then, we link variables to states with an edge `has`.

Data Propagation—After the creation of variable nodes, we reconstruct the propagation of data values traversing application tiers. Consider the example in Figure 5 which models a user changing her password. The user types a new password `pwnd` via a user action, i.e., e_1. This user action is parsed by the parse tree with root $UserAction_1$. Then, the user submits the password (e_2) which is received by the server (e'_2) in an HTTP request with root $HTTPReq$. Finally, the server uses the password in a query (e''_2) with root SQL. In this example, we can distinguish two cases of data propagation. In the first case, the data item `pwnd` propagates along causality edges, i.e., from e'_2 to e''_2. In these cases, we create a query to retrieve terminal nodes of HTTP and SQL trees that are reachable via causality edges as shown in Figure 5. The variables associated to these terminal nodes are then linked via a `propag.` edge. In the second case, the data items propagates from e_1 to e'_2 using first an edge `next`, and then a causality edge. We create a query to retrieve the terminal nodes from user actions to HTTP requests using the query pattern in Figure 5, and then we place `propag.` edges between the variables.

Type Inference—We use types to distinguish security-relevant data values (e.g., anti-CSRF tokens) from uninteresting ones (e.g., constants). Starting from a state transition, we select all variables of a state and group by variable name. Each group is passed to a type inference algorithm which returns the types matching each group. The type inference extracts both syntactical types, e.g., integer, decimal, and boolean, and semantic ones, e.g., session unique (`SU`), user unique (`UU`) and constant (`CO`). The rules to infer a semantic type are the following. If all values are the same, then the type is `CO`. If the data values are the same within a trace session but different between sessions, then the type is `SU`. If the data values are the same within the traces of a user, but different between users, then the type is user unique, i.e., `UU`. The user-generated (`UG`) semantic type is added when there is a propagation chain that starts from a user action. For example, the chain for `pwnd` is of type `UG`.

6 MODEL MINING AND TEST EXECUTION

We now present the test generation via model mining (Section 6.1) and the process of test execution and evaluation (Section 6.2).

6.1 Test Generation

A test of our approach is a state-changing HTTP request and, optionally, an HTTP request parameter carrying an anti-CSRF token. First, we query our model to retrieve all relevant state-changing HTTP requests. Second, for each HTTP request, we mine our model to retrieve HTTP parameter names that carry an anti-CSRF token. As a final step, we query our model to extract the *oracle*. The oracle represents expected behavior that we need to observe during a test to decide whether a relevant state transition occurred.

We begin with a query to detect HTTP requests that trigger security-relevant state transitions. Then, we present the query to identify parameters. Finally, we present a traversal to extract the test oracle.

6.1.1 State Transitions. State-changing HTTP requests can be retrieved by starting from all state transition nodes, and then by traversing the `accepts` to reach an HTTP request. If such an edge exists, then the HTTP request is causing a change of state. We can express this graph traversal as follows. The graph pattern representing connections between an HTTP request parse tree pt, and a state transition node t, is the following:

$$\texttt{SC}(pt, q', tr, q'') \stackrel{def}{:=} \texttt{Trans}(q', tr, q'') \wedge \texttt{Accepts}(tr, pt) \wedge \texttt{HTTPReq}(pt)$$

where q' and q'' are the two states involved in the state transition tr and pt is an HTTP request. Then, we use the predicate in a query:

$$\texttt{Q}_{\texttt{SC}} \stackrel{def}{:=} \{pt : \forall q', q'', tr, \texttt{SC}(pt, q', tr, q'')\}$$

This set contains all parse tree roots pt that can trigger *any* transition of state.

6.1.2 Relevant State Transitions. $\texttt{Q}_{\texttt{SC}}$ contains all HTTP requests that cause a change of state. However, not all changes of state are relevant. For example, requests may result in database operations to log user activities, which is not a security-critical action. To identify such non-critical state changes, we hypothesize that irrelevant queries are likely to occur multiple times within a trace. The occurrence of queries can be determined via abstract parse trees for queries. As a result of the FSM construction, all SQL parse trees reachable via `abstracts` from the same abstract SQL query are *similar* queries. The number of outgoing `abstracts` edges is the number of occurrences of similar queries.

Starting from this observation, we refine $\texttt{Q}_{\texttt{SC}}$ to take into account abstract parse trees of SQL queries and their outgoing `abstracts` edges. The refinement extends $\texttt{Q}_{\texttt{SC}}$ by traversing (i) an edge `parses` from the HTTP request to the HTTP message event, (ii) a causality edge from HTTP message to the data layer event, (iii) a `parses` edge from the data event to the SQL query, and (iv) the SQL query to the abstract SQL query. This query returns a list of pairs of the root of an HTTP request and the root of an abstract SQL query. From this list, we remove all pairs whose abstract SQL query has a number of outgoing edges greater than 1. The HTTP requests of the remaining pairs are called *relevant* state change transitions. We show the accuracy of this heuristic in Section 7.

6.1.3 Security Tokens. After having identified relevant state-changing requests, we search for parameters carrying anti-CSRF tokens. Anti-CSRF tokens can be transported as URL parameters, POST parameters, or in custom HTTP headers. During the construction of the DFM, we created variables with semantic types. For example, variables labeled as `SU` or `UU` carry a value that changes across sessions. As anti-CSRF tokens are required to be unpredictable for the attacker, these variables can carry these tokens. For each state-changing HTTP request, we select all variables with type `SU` or `UU`. Given the root of the parse tree of an HTTP request, we traverse the `accepts` to reach the transition node. From the transition node, we traverse the `to`, thus reaching the new state. Then, we retrieve all variables with `sem_type` $\in \{\texttt{UU}, \texttt{SU}\}$. The output of these queries is a list of pairs of a state-changing HTTP request and a variable name carrying a potential anti-CSRF token.

6.1.4 Oracle. The HTTP request and, optionally, the parameter carrying an anti-CSRF token are used to generate a test against the web application. At the end of a test, we need a way to establish whether a security-relevant state transition occurred. As discussed, a state transition is relevant if it executes a non-reoccurring SQL query. Accordingly, for each HTTP request that we intend to test, we retrieve the abstract parse tree roots of SQL queries with an out-degree equal to one. The traversal to reach abstract SQL queries is shown in Figure 4. These abstract SQL queries are the oracle for the HTTP request.

6.2 Security Tests

We now have pairs of parse trees of state-changing HTTP requests and parameters. The goal of our security tests is to verify the replayability of the requests and check whether they cause SQL queries that are similar to ones in the oracle.

We test web applications as follows. If the HTTP request has an anti-CSRF parameter, we generate an HTTP request by omitting the parameter. If the HTTP request does not have an anti-CSRF parameter, we generate an HTTP request from scratch. In both cases, we update the request's session cookie by replaying the user login user actions[2]. During the test execution, we retrieve the resulting server-side call graph trace to extract SQL queries. Then, we compare SQL queries with our oracle. The comparison can result in one of the following cases. If one of the observed queries matches a relevant query of our model, then our test managed to reproduce the same change of state. In this case, we mark the test as *successful*. If all queries either match a repeated query or are not in our model, then we conclude that we cannot reproduce the same state-changing operation, and mark the test as *failed*.

7 EVALUATION

We now present the evaluation of `Deemon` against popular web applications.

Category	Web Application	Version	LoC
Accounting	Invoice Ninja (IN)	2.5.2	1,576,957
	Simple Invoices (SI)	2013.1b.8	601,532
eCommerce	AbanteCart	1.2.4	151,807
	OpenCart	2.1.0	153,863
	OXID eShop	4.9.8	370,723
	PrestaShop	1.6.1.2	420,626
Forum	MyBB	1.8.8	150,622
	Simple Machines Forum (SMF)	2.0.12	153,072
eMail	Horde Groupware Webmail (Horde)	5.12.14	178,880
	Mautic	1.4.1	2,190,920

Table 3: Web applications for the evaluation.

7.1 Testbed

We assessed Deemon against ten web applications retrieved from the Bitnami catalog [8]. Bitnami is a provider of packaged, ready-to-deploy applications that are typically created upon a customer request. Based on this model, we consider the Bitnami catalog to contain popular web applications.

We selected web applications from four categories, i.e., accounting, eCommerce, email, and forum, in order of appearance. We collected initially 20 applications. Then, during the instrumentation and trace generation, we decided to discard 10 of them: Four used an unsupported runtime environment (i.e., Java or Python), two required paying fees, three of them suffered from a bug in Xdebug (an important component for our approach), and one required a publicly available email server. The list of selected web applications is shown in Table 3.

7.2 Instrumentation

The first step of our evaluation is the instrumentation of the Bitnami applications. Bitnami applications are distributed as self-contained virtual machine (VM) images. Deemon first extracts the virtual disk from the VM image, assigns the disk local mount point, and creates a folder to store program traces. Then, Deemon edits the PHP interpreter configuration file (i.e., `php.ini`) to enable Xdebug—a PHP extension that generates function call tree files—and to change the default Xdebug settings parameters[3]. Finally, Deemon adds a system user and enables the OpenSSH server for the remote access to retrieve call tree files.

After the instrumentation, Deemon imports the VM image in the Virtual Box hypervisor. It boots the VM and takes a snapshot. This snapshot will be the starting point for the rest of the analysis.

7.3 User Actions Input Trace

We captured user actions traces using Selenium IDE [37], a plugin for Firefox. For each category of web application, we used two user roles: regular user (e.g., customer for eCommerce applications) and administrator. For each role, we registered user actions for a selection of web application workflows. We focused on workflows that are common to all categories, such as user sign-up and credential update, and workflows which are specific to a category, e.g., invoice creation for accounting web applications.

Deemon uses user actions traces both to generate dynamic traces and to test the web application against aCSRF. In the first case, Deemon replays all user actions (See Section 7.4). In the second case, Deemon replays only user login actions to update the HTTP request's session cookie (See Section 6.2). To distinguish user login actions from the rest, we use the trace factoring functionality of Selenium IDE. More specifically, we captured input traces as follows:

- *New workflow and no traces for a role*: We use Selenium IDE to capture the entire sequence of user actions of the workflow. Then, we factor actions in two sub-traces: one contains user login actions and the other contains workflow-specific actions. Each sub-trace is stored in its own file;
- *New workflow and a trace for the user exists*: We import user login actions in Selenium IDE and then capture the new workflow-specific user actions;
- *Same workflow but new user*: We duplicate the existing trace files, and replace credentials in the user login trace file. As traces are plain-text files, we use a script to find and replace user credentials.

The number of workflows (WFs) per web application is shown in Table 4. The number varies according to availability of off-the-shelf functionalities and the types of roles.

7.4 Dynamic Traces Generation

To generate dynamic traces, Deemon replays user actions against an instrumented VM. Action replaying is done step-by-step using Selenese Runner Java (SRJ) [37], an interpreter of Selenium user actions, that controls a headless Firefox. The resulting requests are sent to an HTTP proxy that forwards them one-by-one to the server. When the rendering process of the browser is finished, SRJ signals that all statically referenced external resources are retrieved (e.g., images, CSS). Then, Deemon waits for 4 seconds (configurable) to honor any JavaScript asynchronous requests. After that, no more requests are accepted, and the next action is fired. The first request that entered the queue is associated to the fired user action. The association is used during the model construction to establish causality. Images and CSS are not likely to change the state and Deemon does not include them in the network trace. Deemon uses a customizable list of MIME-types and file extensions to exclude these resources.

Throughout the replaying of user actions, whenever Deemon receives an HTTP response, it accesses the VM to retrieve the generated PHP function call tree and session data. The call tree file is associated to the request. This association is used during the model construction to establish causality. Finally, the call tree files are then processed to extract the MySQL queries executed by the web application.

7.5 Performance

In our assessment we used two computers. To generate traces and test for execution, we used a workstation with an Intel i5-4690 CPU,

[2]User actions traces are factored in two parts: actions for the user login and actions for the web application operation. Existing tools to capture user actions, e.g., Selenese IDE [37], support trace factoring. Factoring can be done during the capture or after the generation by searching for user credentials in the trace. We detail the creation of factored user actions traces in Section 7.

[3]Deemon requires the collection of full function variable name and content, function return values, and a computer readable trace file format. These are disabled by default. For more details, please refer to [35].

Web Apps	WFs	Tr. Gen.	Mod. Gen.	Nodes	Edges	Test
AbanteCart	10	212s	1,446s	1,689,083	2,174,622	142s
Horde	3	177s	218s	23,395	30,920	153s
IN	11	152s	215s	97,465	123,419	82s
Mautic	6	176s	485s	191,038	237,036	196s
MyBB	12	214s	261s	96,766	119,270	183s
OpenCart	8	179s	312s	160,401	224,351	123s
Oxid	14	163s	372s	484,651	611,986	333s
Prestashop	13	296s	396s	214,369	273,865	283s
SI	9	128s	170s	34,248	44,983	31s
SMF	7	134s	159s	61,738	78,893	493s

Table 4: Execution time of Deemon.

Web Apps.	Reqs	SC Reqs	Rel. SC Reqs(*)	
AbanteCart	335	335	8	*-98%*
Horde	21	21	3	*-86%*
IN	103	103	11	*-89%*
Mautic	58	21	8	*-62%*
MyBB	104	104	21	*-80%*
OpenCart	117	117	11	*-91%*
Oxid	165	165	10	*-94%*
Prestashop	267	195	16	*-92%*
SI	92	7	7	*0%*
SMF	118	118	69	*-42%*
Total	1,380	1,186	164	*-86%*

(*) *descrease % from SC Reqs*

Table 5: Analysis results for the identification of relevant state-changing (SC) requests.

an SSD disk and 32 GB of RAM. The workstation hosted a VirtualBox hypervisor that Deemon used to deploy Bitnami application containers. To generate our graph, we used a workstation with an Intel i7-4600U CPU, an SSD disk and 12 GB RAM. We used a single instance of Neo4j to handle property graphs of all applications with a total of three million nodes and four million edges.

Overall, Deemon took about 13 minutes to produce the output report for a single web application (see Table 4). About 50% of the execution time is spent to generate traces and testing, which are largely influenced by the web application behavior. For example, the first time that a Prestashop webpage is requested, it creates a cache for frequently requested resources. As we reset the virtual machine to the initial state, Deemon waits for Prestashop to re-create the local cache. Finally, model generation took in average 7 minutes per web application. The execution of queries takes less then 60s.

7.6 Detection of aCSRF

Deemon discovered 29 security-relevant state-changing requests. 17 of these tests detected a vulnerability in four web applications: AbanteCart, Mautic, OpenCart, and Simple Invoices. The remaining 12 requests did not detect vulnerabilities. We present attacks in Section 8.

aCSRF Candidates—Table 5 shows the number of state-changing operations (column "SC Reqs") compared with the total number of operations (column "Reqs"). Results are aggregated by web application. Almost all operations change the state. However, not all of these operations are necessarily relevant for the security analysis. For example, some operations may merely log user activities or be used to manage user sessions. Thus, within a workflow, these operations most likely reoccur multiple times. Table 5 (column "Rel. Reqs") shows the total number of relevant state-changing operations. The number of relevant operations decreased considerably, i.e., on average by -86%, from 1,186 to 164. The decrease is more evident in applications like AbanteCart, where the number of operations decreased by 98% (from 335 to 8), whereas in other cases like Simple Invoice, the number remained unchanged.

We manually inspected SQL queries that were excluded to assess the accuracy of our heuristic. The total number of abstract SQL queries of our testbed is 704, of which 285 are considered not relevant. All these queries are used to perform one of the following operations: session management (e.g., creating a user session and refreshing of session token validity), logging URL access, tracking user activity, and cache management (e.g., MyBB stores entire CSS files in the DB). As these queries are not relevant for our analysis, we conclude that our heuristic is accurate.

Security Tokens—Deemon identified 356 variables of HTTP requests. 248 of them are discarded as they are cookies (192 variables), boundary markers of the multi-part form data encoding (29 variables), and parameter names used with timestamps[4] (27 variables). These parameters cannot successfully protect against aCSRF vulnerabilities. The remaining 108 variables may be anti-CSRF tokens and are used by 53 operations out of 164. The remaining 111 state-changing operations are not protected.

Security Testing—Table 6 shows the total number of tests that were generated for each approach. In total, we executed 111 tests for unprotected operations and 108 for protected ones. Deemon monitored the test execution by using the sensors installed during the instrumentation of the application container. In total, 29 tests were successful and discovered severe vulnerabilities. We discuss these results in detail in Section 8. The remaining 190 tests failed. The majority of failed tests among the protected operations are caused by the presence of an anti-CSRF token. In Section 8, we present an in-depth discussion of the use of this token.The remaining failed tests (including several unprotected operations) are caused by multi-step workflows in which the tested HTTP request depends on another request that is not part of the test. We leave the study of dependencies between requests as a future research direction.

8 RESULTS

We now detail the vulnerabilities that Deemon discovered in the four vulnerable web applications. We also discuss tests that discovered state transitions that cannot be exploited in a aCSRF attack.

8.1 Exploitable Vulnerabilities

Four web applications of our testbed are vulnerable to aCSRF attacks. The severity of this vulnerability ranges from very high, i.e., customer account takeover, website takeover, and database deletion, to low, i.e., adding items into a shopping cart. These vulnerabilities can potentially affect millions of websites. For example, according to Pellegrino et al. [32], OpenCart is used by at least nine million websites whereas AbanteCart is used by 21 thousand websites. We

[4]This technique is often used to bypass browser caching mechanisms

Web Apps.	Protected				Unprotected			
	TCs	Fail.	Succ.	Expl.	TCs(*)	Fail.	Succ.	Expl.
AbanteCart	3	2	1	**1**	5	2	3	**2**
Horde	3	3	-	-	-	-	-	-
IN	12	12	-	-	-	-	-	-
Mautic	19	17	2	**2**	-	-	-	-
MyBB	1	1	-	-	20	9	11	-
OpenCart	2	1	1	**1**	9	5	4	**4**
Oxid	33	33	-	-	-	-	-	-
Prestashop	7	7	-	-	11	11	-	-
SI	-	-	-	-	7	-	7	**7**
SFM	20	20	-	-	47	47	-	-

(*) one TC for each unprotected operation

Table 6: Generation and assessment of test cases. *TCs=nos. of testcases, Fail./Succ.=nos. of un/successful tests, and Expl.=nos. of tests that exploited an aCSRF vulnerability*

responsibly disclosed these vulnerabilities to the developers. In this section, we present a comprehensive overview of our findings and a detailed description of the most severe issues.

8.1.1 Overview of all Vulnerabilities. In summary, we discovered the following vulnerable operations:

AbanteCart—An attacker can (i) take over a customer's user account and (ii) add or modify the shipping address. Developers have already fixed this vulnerability.

OpenCart—An attacker can (i) take over a customer's user account, (ii) add or modify the shipping address, and (iii) add items to a customer's shopping cart[5].

Mautic—An attacker can (i) delete a marketing campaign (part of the core logic of the web application), and (ii) delete recipients from a marketing campaign. Developers of Mautic were unresponsive and we requested and obtained a CVE entry (CVE-2017-8874).

Simple Invoices—An attacker can (i) create new website administrators and customers, (ii) enable payment methods, (iii) create new invoices, and (iv) change taxation parameters. Developers of Simple Invoices acknowledged the presence of the flaw, but they were not working on a patch yet. Accordingly, to protect SI users, we requested and obtained a CVE entry (CVE-2017-8930).

8.1.2 Attack #1: Account Takeover with AbanteCart and OpenCart. The vulnerable state-changing operations of both web applications are not protected by anti-CSRF tokens.

The attack against OpenCart exploits two aCSRF vulnerabilities in the operations to (i) change the user email address and (ii) to update user passwords. When changing this security-sensitive information, OpenCart neither uses anti-CSRF tokens, nor requires users to provide their current password. As a result, an attacker can use aCSRF to reset both email and password to hijack an account.

The attack against AbanteCart exploits the aCSRF vulnerability in the operation to change user data (e.g., email address, first and last name). As opposed to OpenCart, AbanteCart does not use the email address as username. However, it permits recovering usernames and resetting user passwords via the "forgot username" and "forgot password" features. To reset the username, AbanteCart asks for an email address and the last name of the customer, then sends the username in an email. As the attacker can change the email and last name with an aCSRF attack, she can successfully retrieve the username. The "forgot password" requires the username and the email address. As the attacker possesses both, she receives a link to reset the password via email.

[5]This vulnerability was also found and reported by a third party in independent and parallel research.

8.1.3 Attack #2: Database Corruption in Mautic. Our tests discovered two aCSRF vulnerabilities in Mautic which allow an attacker to compromise the core functionalities of the software. Mautic is a marketing automation web application which allows users to create email marketing campaigns and to manage the contacts of the campaign. Our tests discovered aCSRF vulnerabilities in these two operations in which an attacker can delete a specific campaign or a contact. The identifier used to refer to both campaigns and contacts is an incremental integer number. An attacker can either compromise specific campaigns by deleting them or by deleting users, or can delete all existing campaigns and contacts.

8.1.4 Attack #3: Web Application Takeover with Simple Invoices. Our analysis discovered that seven state-changing operations in Simple Invoices are not protected by any session-unique or user-unique data value. In total, six workflows are vulnerable to aCSRF vulnerabilities. These workflows are: creation of a new website administrator, creation of a new customer account, enabling payment methods (e.g., PayPal), adding a new invoice to the database, and changing both global and invoice tax rates.

8.2 Non-Exploitable Tests

11 tests caused a change of state in MyBB. The operations under test were privileged operations performed by the website administrator. While the tests were successfully executed, they cannot be exploited by an attacker. MyBB uses a secret user-unique API key which authenticates the user when performing state-changing requests. If the key is valid, then the operation is executed. While for regular users, in our model this key is correctly labeled unique per user, for the administrator, the key is labeled constant. In our analysis, we used traces from a single administrator user, as MyBB has no concept of multiple administrator accounts. Thus, all these traces contained the same key, causing our type inference algorithm to infer the constant type. Accordingly, the key is included in our tests. The server-side program verifies that the key belongs to the administrator and executes the requested operation.

9 ANALYSIS

Despite its popularity and severity, our results show that the risk posed by aCSRF vulnerabilities is overlooked or even misunderstood. An analysis of our results exposes three distinct classes of developer awareness—complete, partial and nonexisting:

Complete Awareness—At one end of the awareness spectrum, we have full awareness, in which developers deploy aCSRF countermeasures for *all* state-changing operations. Examples of this group are Horde, Oxid, and Prestashop. For example, in the case of Oxid, all 33 tests failed when omitting an anti-CSRF token.

Unawareness—At the other end, we have complete unawareness. Developers may still not be aware of aCSRF nor of the security

implications of successful exploitations. As a result, developers may leave state-changing operations unguarded. Simple Invoices is an example of such a case, in which all state-changing operations are vulnerable to aCSRF attacks.

Partial Unawareness—We observed two interesting cases in which protections are deployed in a selective manner. From our testbed, we can distinguish two clear cases.

Role-based Protections: Examples for this case are OpenCart and AbanteCart, which treat regular users and administrators differently. Our tests showed that administrator operations are protected by anti-CSRF tokens. Omitting these tokens results in rejected state-changing operations. This shows that developers are aware of the security risks and that they deployed adequate countermeasures. However, user operations are not equally protected. As we have seen, even critical operations, such as password change, are exposed to severe attacks leading to customer account takeover. We speculate that this may be the result of an inadequate or incomplete risk analysis and threat modeling during the design phase.

Operation-based Protections: As opposed to the previous case, the distinction is not based on the role of the user, but on the type of operation. In general, web applications offer operations to create, delete, and update elements in a database. Elements can be anything including users, contacts, and products. In Mautic, we observed that creation and updating are guarded by anti-CSRF tokens. Deemon verified that when a token is omitted, a test fails. Similarly for the cases of AbanteCart and OpenCart, this behavior shows that the developers may be aware of the security risks. However, deletion operations are not protected, allowing attackers to compromise the database. In contrast to role-based protections, this may not be caused by inadequate threat modeling. We believe that developers just overlooked this operation.

10 DISCUSSION AND FUTURE WORK

Scalability of the Model—Our assessment showed that a modern workstation can efficiently handle a single graph database instance with three million nodes. We believe that this would be an average use case of our tool. However, property graphs can scale to hundreds of millions of nodes [3]. In these scenarios, Deemon can also be run on servers, exploiting the availability of additional hardware resources.

Performance—The main bottleneck of our approach is the interaction with a running web application. In our experiments, we used one virtual machine at a time, but, we plan to improve performance by spawning parallel, multiple virtual machine instances of the same web application.

Generality of the Approach—Our evaluation was conducted on PHP-based web applications using a MySQL database. While these are popular among web developers, web applications can use different SQL databases or can be written in other programming languages. The modeling framework is independent from the programming language. However, instrumentation and sensors may require new connectors in order to acquire traces.

Detection Power—Deemon was conceived to target aCSRF. However, as for CSRF, other classes of severe vulnerabilities have been neglected by the security community, e.g., session management issues and race conditions. The lowest common denominator of these classes is that they are much more complex to detect when compared to XSS and SQLi. The detection of these classes require learning in-depth behaviors of a program and synthesizing the relevant aspects in models. From this point of view, our modeling paradigm has to be seen as an initial effort toward this long-term goal. Deemon provides a unified representation for artifacts and models used in dynamic analysis, and furthermore, it provides a semantic of the relationships between them. However, our representation may not be sufficient to capture relevant aspects for the detection of other classes of vulnerabilities.

11 RELATED WORK

To the best of our knowledge, this is the first work proposing a technique for the detection of aCSRF vulnerabilities. Existing work focused mainly on defense techniques, proposing new HTTP headers (See, e.g., [6, 20, 21, 25]) and new CSRF-based attacks (e.g., [38]). As opposed to these works, Deemon does not protect from exploitation, but it allows discovery of CSRF during the testing phase of the development of web applications.

Property Graphs and Vulnerability Detection—Our approach relies on graph databases for the representation and composition of models. Similar to our idea, Yamaguchi et al. [42] and Backes et al. [3] combined different code representations in a property graph. While these works focused on static source code representations, we model dynamic behaviors of the application. Furthermore, these works, similarly to others in the area of web security, focused on input validation vulnerabilities. In contrast, our work presented a technique to discover aCSRF.

Dynamic Analysis—Research on dynamic analysis has been very active over the last decade, proposing new techniques and tools to detect a variety of vulnerabilities. For example, unsupervised web application scanners are very popular tools routinely used to detect vulnerabilities in web applications. Starting from a URL, a web application scanner crawls a web application and then, for each discovered input, it probes the application with crafted input strings. There are plenty of commercial and non-commercial scanners, including tools proposed by the research community [10, 17, 23, 27, 33]). While web scanners are effective in the detection of XSS and SQLi, they still perform poorly or even fail in the detection of more sophisticated vulnerabilities, including aCSRF vulnerabilities [7, 11]. Compared to web scanners, Deemon does not include a crawler component. Crawlers use breadth- or depth-first algorithms which are not adequate to reach security-relevant state-changing requests. As opposed to this technique, Deemon—similarly to other dynamic approaches (See, e.g., [26, 32])—follows a different approach in which input traces are used to explore in depth the functionalities of web applications. Other approaches have been proposed in order to address more complex flaws, e.g., user authentication (see, e.g., [4, 44]), and logic vulnerabilities (e.g., [32]), often combining model inference with dynamic testing. These approaches analyze components and functionalities that are specific to the vulnerability being targeted, thus making them inherently limited in the ability to reason about the presence of CSRF vulnerabilities.

Static Analysis—Static program analysis has been used to detect several classes of vulnerabilities, e.g., input validation vulnerabilities [3, 9, 18, 22], authorization vulnerabilities [28], and logic flaws [39]. Similarly as for dynamic techniques, none of the existing approaches target CSRF vulnerabilities. Second, more and more web applications tend to use programming languages and coding patterns, e.g., runtime second-order function calls [14, 15] and SQL query construction [1], that are hard to treat statically. Static analyzers often address these shortcomings by calculating over- or under-approximations that can cause high rates of false positives [3]. In these scenarios, dynamic techniques such as Deemon are a valid alternative; however, existing approaches lack the sophistication to detect CSRF.

12 CONCLUSION

We presented Deemon, to the best of our knowledge the first security testing framework that can detect aCSRF vulnerabilities. At the core of Deemon is a new modeling paradigm based on property graphs that defines (i) searchable model components to represent multiple aspects of web applications, and (ii) a query language that allows expression of suspicious or vulnerable behaviors. Our experiments detected 14 severe aCSRF vulnerabilities affecting four web applications that can be used to take over websites, or user accounts, and compromise database integrity. Finally, we assessed the current awareness level of the aCSRF vulnerabilities and showed alarming behaviors in which security-sensitive operations are protected in a selective manner. This work has successfully demonstrated the capabilities of our paradigm, which comprehensively captures non-trivial, cross-tier aspects of modern web applications. In the near future, we intend to leverage the opportunities provided by our paradigm and extend the approach towards additional vulnerability classes.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback and our shepherd Adam Doupé for his support in addressing reviewers' comments. We would like also to thank Benny Rolle and Florian Loch for their contribution to the development of Deemon. This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345, 16KIS0656), the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), and the project BOB (FKZ: 13N13250).

REFERENCES

[1] David Anderson and Mark Hills. 2017. Query Construction Patterns in PHP. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 452–456. DOI: https://doi.org/10.1109/SANER.2017.7884652

[2] Marc Andreessen. 1993. proposed new tag: IMG. [Posting to the www-talk mailing list], http://1997.webhistory.org/www.lists/www-talk.1993q1/0182.html. (February 1993).

[3] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application. In *2nd European Symposium on Security & Privacy (EuroS&P 2017) (to appear)*.

[4] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. 2013. 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*.

[5] A. Barth. 2011. The Web Origin Concept. RFC 6454 (Proposed Standard). (Dec. 2011). http://www.ietf.org/rfc/rfc6454.txt

[6] Adam Barth, Collin Jackson, and John C. Mitchell. 2008. Robust Defenses for Cross-site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 75–88. DOI: https://doi.org/10.1145/1455770.1455782

[7] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. 2010. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *2010 IEEE Symposium on Security and Privacy*. 332–345. DOI: https://doi.org/10.1109/SP.2010.27

[8] Bitnami. 2016. Bitnami Applications. (2016). https://bitnami.com/stacks

[9] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-Order Vulnerabilities in Web Applications. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 989–1003.

[10] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 523–538.

[11] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny Can'T Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'10)*. Springer-Verlag, Berlin, Heidelberg, 111–131.

[12] Dave Ferguson. 2009. Netflix CSRF Revisited. [online], http://appsecnotes.blogspot.de/2009/01/netflix-csrf-revisited.html. (January 2009).

[13] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. 2009. Using Formal Specifications to Support Testing. *ACM Comput. Surv.* 41, 2, Article 9 (Feb. 2009), 76 pages. DOI: https://doi.org/10.1145/1459352.1459354

[14] Mark Hills. 2015. Evolution of dynamic feature usage in PHP. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. 525–529. DOI: https://doi.org/10.1109/SANER.2015.7081870

[15] Mark Hills, Paul Klint, and Jurgen J. Vinju. 2013. An empirical study of PHP feature usage: a static analysis perspective. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. 325–335. DOI: https://doi.org/10.1145/2483760.2483786

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

[17] Yao-Wen Huang, Chung-Hung Tsai, Tsung-Po Lin, Shih-Kun Huang, D. T. Lee, and Sy-Yen Kuo. 2005. A Testing Framework for Web Application Security Assessment. *Comput. Netw.* 48, 5 (Aug. 2005), 739–761. DOI: https://doi.org/10.1016/j.comnet.2005.01.003

[18] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, New York, NY, USA, 40–52. DOI: https://doi.org/10.1145/988672.988679

[19] Martin Johns. 2007. The three faces of CSRF. talk at the DeepSec2007 conference, https://deepsec.net/archive/2007.deepsec.net/speakers/index.html#martin-johns. (November 2007).

[20] Martin Johns and Justus Winter. RequestRodeo: client side protection against session riding. In *in Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*. 5–17.

[21] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. 2006. Preventing Cross Site Request Forgery Attacks.. In *SecureComm*. IEEE, 1–10.

[22] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP '06)*. IEEE Computer Society, Washington, DC, USA, 258–263. DOI: https://doi.org/10.1109/SP.2006.29

[23] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. 2006. SecuBat: A Web Vulnerability Scanner. In *Proceedings of the 15th International Conference on World Wide Web (WWW '06)*. ACM, New York, NY, USA, 247–256. DOI: https://doi.org/10.1145/1135777.1135817

[24] Florian Kerschbaum. 2007. Simple cross-site attack prevention. In *2007 Third International Conference on Security and Privacy in Communications Networks and the Workshops - SecureComm 2007*. 464–472. DOI: https://doi.org/10.1109/SECCOM.2007.4550368

[25] Ziqing Mao, Ninghui Li, and Ian Molloy. 2009. *Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection*. Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255.

[26] Sean Mcallister, Engin Kirda, and Christopher Kruegel. 2008. Leveraging User Interactions for In-Depth Testing of Web Applications. In *Proceedings of the*

11th International Symposium on Recent Advances in Intrusion Detection (RAID '08). Springer-Verlag, Berlin, Heidelberg, 191–210. DOI: https://doi.org/10.1007/978-3-540-87403-4_11

[27] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web* 6, 1, Article 3 (March 2012), 30 pages. DOI: https://doi.org/10.1145/2109205.2109208

[28] Maliheh Monshizadeh, Prasad Naldurg, and V. N. Venkatakrishnan. 2014. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 690–701. DOI: https://doi.org/10.1145/2660267.2660337

[29] Neo Technology, Inc. 2017. The Cypher Query Language. (2017). http://tinkerpop.apache.org/

[30] OWASP. 2017. OWASP Testing Guide v4. (2017). https://www.owasp.org/index.php/OWASP_Testing_Project

[31] OWASP. 2017. The OWASP Top 10 Project (from 2007 to 2013). (2017). https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[32] Giancarlo Pellegrino and Davide Balzarotti. 2014. Toward Black-Box Detection of Logic Flaws in Web Applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*.

[33] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. *jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications*. Springer International Publishing, Cham, 295–316. DOI: https://doi.org/10.1007/978-3-319-26362-5_14

[34] Petko D. Petkov. 2007. Google GMail E-Mail Hijack Technique. (2007). http://www.gnucitizen.org/blog/google-gmail-e-mail-hijack-technique/

[35] Derick Rethans. 2017. Xdebug Extension for PHP. (2017). https://xdebug.org/

[36] Thomas Schreiber. 2004. Session Riding - A Widespread Vulnerability in Today's Web Applications. (2004). http://www.securenet.de/papers/Session_Riding.pdf

[37] Selenium Committers. 2017. SeleniumHQ. (2017). http://www.seleniumhq.org/

[38] Avinash Sudhodanan, Roberto Carbone, Luca Compagna, Nicolas Dolgin, Alessandro Armando, and Umberto Morelli. 2017. Large-Scale Analysis & Detection of Authentication Cross-Site Request Forgeries. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. 350–365. DOI: https://doi.org/10.1109/EuroSP.2017.45

[39] Fangqi Sun, Liang Xu, and Zhendong Su. 2014. Detecting Logic Vulnerabilities in E-commerce Applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*.

[40] Anne van Kesteren, Julian Aubourg, Jungkee Song, and Hallvord R. M. Steen. 2016. XMLHttpRequest Level 1. (2016). https://www.w3.org/TR/XMLHttpRequest/

[41] Rui Wang, Shuo Chen, and XiaoFeng Wang. 2012. 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 (SP '12)*. IEEE Computer Society, Washington, DC, USA, 365–379. DOI: https://doi.org/10.1109/SP.2012.30

[42] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 590–604. DOI: https://doi.org/10.1109/SP.2014.44

[43] William Zeller and Edward W. Felten. 2008. Cross-Site Request Forgeries: Exploitation and Prevention. (2008). http://www.cs.utexas.edu/~shmat/courses/cs378/zeller.pdf

[44] Yuchen Zhou and David Evans. 2014. SSOScan: Automated Testing of Web Applications for Single Sign-on Vulnerabilities. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 495–510. http://dl.acm.org/citation.cfm?id=2671225.2671257