A Survey on Control-Flow Integrity Means in Web Application Frameworks

Bastian Braun, Christian v. Pollak, and Joachim Posegga

Institute of IT Security and Security Law (ISL)
University of Passau, Germany
{bb,jp}@sec.uni-passau.de, chris.evp@gmail.com

Abstract. Modern web applications frequently implement complex control flows, which require the users to perform actions in a given order. Users interact with a web application by sending HTTP requests with parameters and in response receive web pages with hyperlinks that indicate the expected next actions. If a web application takes for granted that the user sends only those expected requests and parameters, malicious users can exploit this assumption by crafting harming requests. We analyze recent attacks on web applications with respect to user-defined requests and identify their root cause in the missing explicit control-flow definition and enforcement. Then, we evaluate the most prevalent web application frameworks in order to assess how far real-world web applications can use existing means to explicitly define and enforce intended control flows. While we find that all tested frameworks allow individual retrofit solutions, only one out of ten provides a dedicated control-flow integrity protection feature. Finally, we describe ways to equip web applications with control-flow integrity properties.

1 Introduction

Over the past two decades, the Web has evolved from a simple delivery mechanism for static content to an environment for powerful distributed applications. In spite of these advances, remote interactions between users and web applications are still handled using the stateless HTTP protocol, which has no protocol level session concept. Handling session state is fully left to the web application developer or to high-level web application frameworks.

Web applications often include complex control flows that span a series of multiple distributed interactions. The application developer usually expects the user to follow the intended control flow. However, if a web application does not carefully ensure that interactions adhere to the intended control flow, attackers can easily abuse the web application by using unexpected interactions. Several known attacks have exploited this kind of vulnerability in the past. The attacks' impact ranges from sending more free SMS text messages than actually allowed [1], over unauthorized access to user accounts [2–4], up to shopping expensive goods with arbitrarily low payments [5].

Almost every web application that implements a business logic spanning several request-response round trips has a need for control-flow integrity. So, a control-flow integrity module should be reusable. Web application frameworks provide sets of reusable features to facilitate web application development. In this paper, we examine the ten most prevalent web application frameworks on their support for control-flow integrity. This gives us an insight how far the majority of web applications can use and add control-flow integrity protection without changing the application or the underlying framework. Looking at it the other way round, missing support requires developers to manually implement protection means, which, as history shows, leads to more weaknesses because the implementation is often either omitted or flawed. We also check two crucial aspects of control-flow integrity: parameter integrity, which means that malicious users can not tamper with the HTTP parameters' data type, and race condition protection, which mitigates attack vectors based on the same request sent multiple times in parallel. The specific contributions are threefold: First, we explore the vulnerability pattern that leads to control flow-related attacks. Second, we explain how this class of vulnerabilities can be overcome. Third, we present the results of our investigative survey on mechanisms in web application frameworks that help the developer to achieve control-flow integrity.

This paper is structured as follows. In the next section, we explain the technical details of control-flow integrity. We describe real-world examples of attacks and identify their root cause. Then, in Section 3, we give the results of our survey on control-flow integrity means in web application frameworks. We check ten frameworks for their capabilities to mitigate attacks based on unexpected request sequences, concurrent requests on the same action, and HTTP parameter manipulation. In Section 4, we present related approaches concerning control-flow integrity in web applications. Finally, we conclude in Section 5.

2 Exploring Control Flow in Web Applications

In this section, we investigate in more detail the problem of control-flow integrity of web applications, analyze several real-world attacks, and discuss their root causes.

2.1 Technical Background

Modern web applications are usually developed with the help of web application frameworks. Such frameworks encapsulate basic functionality that can be reused for application development at a large granularity level. Typical features include session initialization and cookie delivery as well as HTTP communication and HTML content generation support. The application code then implements the actual business logic and uses high-level functions provided by the framework.

Technically, the user's web browser interacts with the remote application by sending HTTP requests. HTTP is a stateless protocol without session concept [6]. This means that each request is independent of all others. The protocol does not inherently link one request to the next. The logic of current web applications, however, is stateful. Users expect personalized accounts where they can log in and find their accustomed environment like a history of transactions, what their friends do etc. They can perform actions, for example, buy goods or add new friends. This requires the web application to keep the current state of the user's session, consisting of persistent information (e.g. the friends list) in a database and temporary information (e.g. the shopping cart) in a so-called session record.

From the web application's viewpoint, such user actions are composed of multiple steps, which correspond to multiple HTTP requests from the user to the web application. For each step, the client receives a web page with hyperlinks that offer possible next steps to a user. Upon clicking a link, the user's browser sends a particular HTTP request to the web application, which then performs actions in order to progress to the next step in the workflow. The actions are defined by the URI [7] of the HTTP request, the request parameters, and the server-side session record. For instance, a shopping workflow might first require to put items to the cart, then log in, provide a shipping address and shipping speed, choose a payment option, and finally review the complete order. For every step, the user is supposed to fill some form and press a button. A web application has to ensure that a malicious user does not enter the address of the review page into his browser without providing payment details.

2.2 Root Causes for Weaknesses

Web application developers assume that users first request one of possibly several application entry points, e.g. the base directory at `http://www.example.com`. Upon the first request, the web application sends a given response containing a set of hyperlinks or a redirect instruction to the user. As users tend to click on hyperlinks in order to navigate through the application, developers might assume that only the given requests will be accessed next. However, the user is technically not bound to click on one of the provided hyperlinks but she can still send requests that are not provided within this response. Sent requests can differ from provided hyperlinks in terms of addressed methods and HTTP parameters. Vulnerable web applications fail to handle unintended user behavior in terms of sequences of requests.

More formally, web application developers implement implicit control-flow graphs. In each state, sending a request leads to a subsequent state in the graph. Executing a step corresponds to changing the server-side state. Control-flow weaknesses occur if an attacker is able to address at least one method, i.e., cause a state-changing action, that is not meant to be addressed in the respective session state. This transition does not exist in the respective control-flow graph due to the developer's assumption that the request does not happen at that time. Vice versa, a web application implementing a control-flow graph with transitions for all requests in every state is not susceptible to control-flow weaknesses.

Control-flow weaknesses cannot be overcome with usual access control means. The attack vectors include only requests that are in the scope of the user's rights. Access control mechanisms prevent users from accessing sensitive API methods at all times. Control-flow integrity protection, however, prohibits access to regular API methods in an unsolicited order or context. The measure to achieve this can partially overlap with Cross-Site Request Forgery (CSRF) protection: web applications can issue tickets in the form of nonces that must be appended to requests [8]. A request without a ticket is not processed. This prevents that CSRF attackers can craft requests that are finally executed on behalf of the victim. In some cases, this can also prevent attacks on control-flow integrity: First, nonces must be unique for every request. Some web applications use only one ticket for a user session to save server-side resources. While a session-wide ticket reliably prevents CSRF attacks, it can not prohibit attacks on control-flow integrity. Second, a ticket must be bound to the whole request including all parameters. Otherwise, an attacker could tamper with unprotected parameters and change a request's context. The first example concerning HTTP parameter manipulation given in Sec. 2.3 describes such an attack. Third, the ticket must be invalidated immediately after use to prevent race condition exploits and faults due to "Back" button usage. Both of these scenarios use correct request-ticket combinations but more often than expected. Finally, even if all these measures are taken properly, there is still an open attack vector: the user can start the same workflow in different sessions up to the point where a race condition exploit should be run. Then, he can perform the next step in all sessions in parallel with all requests equipped with correct tickets.

Existing web applications enforce the intended control flow based on session-contained parameters. This allows only the implicit definition of workflows. The previous actions are assumed to set the parameters and, thus, allow the execution of next actions. The actual workflows are not explicitly determined preventing the proper assessment of enabled workflows. The central and explicit definition of facilitated workflows provides guarantees of request sequences to the relying web application. One crucial aspect of reliable request sequences are controlled HTTP parameters as we will show by the attacks in Section 2.3.

2.3 Examples

Several kinds of attacks exploit the fact that attackers can craft arbitrary requests instead of clicking on provided hyperlinks. Real-world examples of control-flow integrity violations are race conditions, manipulated HTTP parameters, unsolicited request sequences, and the compromising use of the browser's "Back" button.

Race Conditions In order to exploit race conditions [9] in web applications, attackers can send several crafted requests almost in parallel. Web applications are multi-threaded by design and, so, have an inherent concurrency property when receiving several requests in a short time frame. There is no low-level serialization of requests for performance reasons. If the web application does not handle concurrent requests by proper synchronization, the actual application semantics can be changed in this way. In one real-world example, a web application provided an interface to send a limited number of SMS text messages per day [1]. The web application first checked the current amount of sent messages (*time-of-check*), then delivered the message according to the received request, and finally updated the number of sent messages in the database (*time-of-use*). Attackers were able to send more messages than allowed by the web application by crafting a number of HTTP requests, each containing the receiver and text of the message to be sent. These requests were sent almost in parallel and the multi-threaded web application processed the incoming requests concurrently. This way, the attacker exploited the fact that the messages were sent before the respective database entry was updated, leading to the delivery of all requested messages. The developers' underlying assumption was that users finish one transmission process before sending the next message and do not request one operation of the workflow several times in parallel. While race conditions are in general known for years, they are a crucial aspect of control-flow integrity because the expected sequence of steps in a workflow can be manipulated. Instead of proceeding to the next step, the same action is executed repeatedly. This way, the attack leads to a corrupt application state.

Unsolicited Request Sequences Attackers can not only modify the requests' parameters but also craft requests to any method of the web application. Besides manipulated HTTP parameters, web applications might face unexpected requests to any method. For instance, in another given scenario by Wang et al. [5], a malicious shopper was able to add items to her cart between checkout and payment. She was only charged the value of her cart at checkout time. The recently added items were not invoiced.

HTTP Parameter Manipulation HTTP requests can contain parameters in addition to the receiving host, path, and resource. As the parameters are sent by the client, the user can control the parameters' values and which parameters are sent to the web application. Wang et al. [5] found a bunch of logic flaws in well-known merchant systems and Cashier-as-a-Service (CaaS) services. These flaws allowed them to buy any item for the price of the cheapest item in the store.

Compromising Use of the "Back" Button Current web browsers are fitted with a so-called "Back" button. It is meant to navigate back to the last visited web page. Depending on the configuration, the last request either has to be repeated in order to display the page or the content is loaded from the browser's local storage ("cache"). In the context of a workflow, the user takes one step back which in some cases is unwanted and also undetectable by the web application. In fact, the usage of this button usually invokes the last action again rather than rolling back the last changes. Hallé et al. [10] describe related navigation errors.

To sum up, we can say that uncontrolled sequences of user requests might cause confusions on the web application's state if it does not take care of handling even unprovided requests. In the next section, we dive deeper into precautions provided by web application frameworks.

3 Probed Web Application Frameworks

In this section, we describe our survey on control-flow integrity protection means of the most prevalent web application frameworks. We tested the top 10 web application frameworks according to the BuiltWith index [11] on 12 Jan 2013. The list contains the most common server technologies among the 10,000 most popular web sites. However, it also includes technologies that are out-of-scope for our survey because they only denote the platform, e.g. PHP. We are aware that PHP itself does not provide any control-flow integrity means, thus, we omitted all technologies that do not fall within the following definition:

> "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes." [12]

The in that way derived frameworks are Apache Tapestry [13], Google Web Toolkit [14], Spring [15], CodeIgniter [16], CakePHP [17], Kohana [18], ASP.NET [19] (Web Forms [20], MVC [21], and Web Pages [22]), and Ruby on Rails [23]. At the time of publication, Django [24] reached considerable popularity such that we quickly go into Django as well.

The testing procedure included first a check of the manuals on hints concerning control-flow integrity means. More precisely, we looked for existing functionality that can be configured, e.g. by providing a policy, and then enforces control-flow integrity features. The customer should not be required to implement but only configure enforcement. We compiled a chain of basic web pages that are connected via links and buttons and supplied a control-flow integrity policy whenever an enforcement feature is mentioned. Next, we tried to overcome the intended control flow by crafting requests.

Then, we tested each framework for race condition protection means which are a crucial part of control-flow integrity (see Section 2.3). We crafted a web page that accepts user requests and expects a textual parameter. The content of this parameter is posted to a message board, and a message counter keeps track on the number of posts. We allowed a maximum of five messages. A small script quickly sent message requests to that page trying to post more messages than actually allowed.

Finally, we wanted to learn how the request parser behaves. Therefore, we changed the given HTTP GET and POST parameters to see whether there is any enforcement based on the data type or a constant value.

3.1 Enforcing Sequences of Actions

In this section, we describe our findings on control-flow integrity means in the top 10 web application frameworks (see above). Our first reference point is each framework's manual. In case of promising hints, we conducted our practical test run, a simple flow definition and violating requests.

An incoming request can cause a sequence of server-side operations in Apache Tapestry [13]. Every request is first handled by a master dispatcher which forwards the request to the respective processing and page rendering routines. These routines can trigger new events (*event bubbling*). The web application reaches a stable state when all events finished processing. However, there is no enforcement mechanism to control the sequence of user actions.

Google Web Toolkit [14] allows the developer to write Java code which is then translated to server-side Java classes and client-side JavaScript code by the GWT SDK (Software Development Kit). Most operations and all user interaction happen on client side. The client-side code communicates with the web server using AJAX requests (*Asynchronous JavaScript and XML*) [25]. These requests are called *remote procedure calls* because they call procedures on server-side. There is no enforcement mechanism concerning the sequence of processed requests.

Spring [15] is actually a modular Java framework. It becomes a web application framework by including the web module. In that combination, Spring implements a model-view-controller (MVC) architecture without any control-flow integrity protection. However, Spring is extensible by so-called projects[1] among which Spring Web Flow [26] is meant to provide flow control for web applications. It inserts a special web flow controller into the MVC-based application in order to ensure that every incoming request can be checked for policy compliance. Developers can define intended control flows as XML or as Java code. A control-flow definition contains a number of states and for each state its outgoing transitions. Processed requests trigger a state transition if they contain the respective *flowExecutionKey* and *eventID*. The *flowExecutionKey* denotes the access key to the control flow while the *eventID* is the transition's identifier. Both are transmitted as HTTP parameters. This allows Spring Web Flow to distinguish between tabs and, thus, allow multiple control flows in separate browser tabs without interference. It can also control side effects caused by the usage of the browser's "Back" button in such a way that it prevents accidental re-execution of the last action (see Section 2.3). In our practical test runs, we made sure that the flow definition was properly enforced. We crafted requests to all existing actions but no spoofed request was processed.

CodeIgniter [16] is a PHP-based web application framework implementing a MVC architecture. A dispatcher receives all incoming requests and forwards them to their respective controller. A file named `routes.php` does the assignment of requests to controllers. The included *security library*[2] processes all incoming requests and outgoing responses after the dispatcher and before the controller. However, it only sanitizes user input to prevent cross-site scripting (XSS) and equips links in outgoing responses with nonces to prevent cross-site request forgery (CSRF). A control-flow integrity enforcement mechanism is not part of the framework.

[1] See `http://www.springsource.org/projects` for a complete list.

[2] See `http://ellislab.com/codeigniter/user-guide/libraries/security.html` for details.

CakePHP [17] like CodeIgniter is a PHP-based web application framework implementing a MVC architecture. The basic request processing is also similar: a dispatcher forwards all incoming requests to controllers according to the configuration file `routes.php`. CakePHP comes with a *security component*[3] that can be used by controllers to prevent CSRF and form tampering, require given HTTP methods (i.e. GET, POST, PUT, and DELETE) or SSL, or restrict communication between controllers. None of these features, however, allows enforcement of control-flow integrity properties.

Kohana [18] also falls into the category of PHP-based frameworks that implement a MVC architecture. The central configuration file is named `Bootstrap.php`. It gathers the basic configuration, lists included modules which provide additional functionality, and defines responsible controllers based on the requested URL. The supplied *security class*[4] offers protection routines against XSS, SQL injection, and to check input conformity. Control-flow integrity protection is not offered.

ASP.NET [19] is a web application framework built on the .NET framework for Windows operating systems. It allows to implement web applications in programming languages C# and VB.NET. ASP.NET comprises three distinct application paradigms:

– ASP.NET Web Forms [20] generates web applications that consist of objects called *pages*. Pages contain HTML code and server side controls. Those controls are triggered on incoming requests and perform data processing before a response is rendered and sent back to the client. The provided *state management*[5] offers data storage options across request-response round trips, similar to cookies and session records. There is no control concerning state transitions.
– ASP.NET MVC [21] again follows the model-view-controller architecture. The central dispatcher is named `Global.asax`. It assigns incoming requests to their respective controllers. An *authorization filter*[6] can be executed before the request is processed by the assigned controller. This filter checks a user's access rights to the requested action but does not control the sequence of actions.
– ASP.NET Web Pages [22] is the most lightweight web application framework of the ASP.NET family. Its application model is similar to Web Forms. Web Pages contain more HTML code enriched by dynamic server-side features while Web Forms generate most HTML elements dynamically. From a control-flow integrity point of view, there is no big difference between both.

With Ruby on Rails [23], a developer implements model-view-controller-based web applications in Ruby. The underlying principle is equivalent to the

[3] See `http://book.cakephp.org/2.0/en/core-libraries/components/security-component.html` for details.
[4] See `http://kohanaframework.org/3.3/guide/kohana/security` for details.
[5] See `http://msdn.microsoft.com/en-us/library/75x4ha6s.aspx` for details.
[6] See `http://msdn.microsoft.com/en-us/library/dd505057(v=vs.98).aspx` for details.

above described MVC-based web application frameworks: The *action dispatch* component forwards requests to controllers based on a given configuration file, named `routes.rb`. Filters can be applied before and after the execution of the controller. However, there is no given control-flow integrity protection mechanism.

Django [24] is also MVC-based and uses regular expressions to assign requests to views. The request can be checked by *middleware* components before and after being processed by the view.

In summary, it can be stated that Spring with Web Flow offers the only control-flow integrity protection feature in the field of common web application frameworks. Common security features are anti-CSRF tokens, authorization management, and input validation against cross-site scripting and SQL injection. It seems to us that control-flow integrity has not yet received much attention and is overlooked in web application development.

3.2 Race Condition Protection

Section 2.3 shows that race conditions can be a severe problem in web applications. Roughly speaking, they occur whenever some action can be executed next but only a limited number of times. This is usually the case for repetition-bounded state changing actions. It depends on the application's business logic which actions are concerned. So, a web application framework should offer means to define such actions and respective requests in order to make the framework process them sequentially instead of parallel. We could endorse the results given in Section 3.1 that none of the frameworks offers such protection with the exception of Spring Web Flow which we will take a deeper look at in this section.

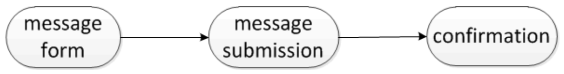

Fig. 1. The intended flow for sending a message. First, the message text is entered. Next, the message is transmitted, and finally, a confirmation is given.

We implemented a number of web pages that allow the user to first enter a message text. Then, the message is sent via HTTP POST to the message board and a confirmation is given in the last step. The intended flow is given in Figure 1. We crafted the respective Spring Web Flow policy. Listing 1.1 shows the pseudo code of the method that receives the request.

```
if db.sentMessages < 5 {
  board.includeMessage(m);
  db.update(sentMessages++);
}
```

Listing 1.1. Pseudo code of the message processing method

The goal was to send a high number of messages and make more than five accepted for the message board. In a first attempt, we requested the message form, learned the request target and parameters for the message submission and sent 20 messages almost in parallel. The result shows that only one of the messages was accepted. It seemed that the *flowExecutionKey* and *eventID* were checked before the actual application code handled the request.

In a next attempt, we started the same flow in ten distinct browser tabs, thus obtaining ten different *flowExecutionKey*s, as Web Flow is able to handle multi-tabbed browsing. We were able to sent eight messages upon virtually clicking "Send" simultaneously in all ten tabs. Just for the record, we repeated the last experiment using ten different browsers instead of browser tabs and succeeded again. The difference between the last two configurations from the server's point of view is that all ten requests belong to the same user session in the first case and to ten different user sessions in the second case. In both scenarios, the actual flow definition was not violated because all steps were performed in the right order and there were no interfering requests within each single flow. The actual exploit happened on a logical level. The number of parallel executions of the same control flow within the same session or the same user account was not limited. There is no policy statement to define such restrictions. So, developers need to take care and implement customized solutions.

3.3 Parameter Enforcement

Next, we checked whether changes of the expected data type in request parameters lead to faults in web applications. For instance, we sent a request `http://www.example.com/controller/action/foo` while the application expected a numerical parameter, e.g. `http://www.example.com/controller/action/13`.

Our observation shows that the underlying programming language plays a decisive role: the Java-based frameworks fail while casting the unexpected string type to the integer variable. Apache Tapestry can not find an appropriate handler for our request and responded with a default page. Google Web Toolkit and Spring (incl. Web Flow) raise exceptions, `undeclared` and `NoMatchingTransitionException` respectively. The type-safe nature of Java in this case prohibits unintended user input, albeit the request is processed in the opposite case: a method expecting a string also accepted a number which is then, however, interpreted as a string.

The situation is different for PHP-based frameworks, because PHP does not have inherent type safety. The web application frameworks, however, all offer type matching expressions. CodeIgniter knows types `:num` and `:any` which include numerical values and all values respectively. CakePHP and Kohana suggest to enforce data types by means of regular expressions. The expression `'param' => '[0-9]+'` makes sure that only integers are accepted for parameter `param`.

There is another problem for ASP.NET web applications because they can be implemented in C# or VB.NET, thus not benefit from underlying data types. The attempt to maintain type safety is similar to the PHP world. So-called *constraints* can define regular expressions. The integer definition looks like the following: `param = @ "\d +"` where `d` is the symbol for a digit.

Ruby on Rails also accepts *constraints*, i.e. regular expressions defining the range of accepted values for parameters. The integer definition is `:product => /[0-9]+/`

Finally, Django assigns requests to views based on regular expressions, i.e. requests with forged parameters can be sorted out before they are processed.

We can conclude that web application frameworks contribute to type safety in web applications. This makes those attacks harder which rely on request processing weaknesses based on parameter type manipulation.

3.4 Summary

Our tests show that support for control-flow integrity in web application frameworks is insufficient. Existing approaches relying on implicit control-flow enforcement are dangerous: Modules are per se not reusable; setting values to indicate that some action has been performed can have side effects allowing also subsequent actions of the same workflow or repeated execution of the next action; and finally, authorization must always be distributed because the permission is given in one method while the check is performed in a different method. The need for framework inherent control-flow integrity can only be fulfilled by Spring Web Flow (see Table 1).

Framework	Version	CFI	RC	Param.	Lang
Apache Tapestry	5	–	–	+	Java
Google Web Toolkit	2.5	–	–	+	Java
Spring/Web Flow	3.2.2/2.3.0	–/+	–/≈	+	Java
CodeIgniter	2.1.3	–	–	+	PHP
CakePHP	2.3.0	–	–	+	PHP
Kohana	3.3.0	–	–	+	PHP
ASP.NET Web Forms	4.5	–	–	+	C#, VB.NET
ASP.NET MVC	4	–	–	+	C#, VB.NET
ASP.NET Web Pages	2	–	–	+	C#, VB.NET
Ruby on Rails	1.9.3	–	–	+	Ruby
Django	1.5.1	–	–	+	Python

Table 1. The test results. A plus (+) denotes that the protection feature is provided in the framework. A minus (–) means that there is no regular support for such protection. CFI is the property to enforce the right order in request processing. RC stands for race condition protection. Param. is the ability to ensure type safety of received request parameters. The Spring Web Flow race condition protection is a special case because it can only protect against single flow race conditions.

Nevertheless, almost all frameworks in scope provide suitable execution points to hook into. The central dispatchers of the MVC-based frameworks can observe every request passing by. Equipping those dispatchers with a control-flow integrity feature seems natural. Moreover, most of the frameworks have filters, that are executed before and after the controller processes the request. Table 2 gives a list of dispatchers and filters.

Framework	Dispatcher	Filters
Apache Tapestry	Master Dipatcher	–
Google Web Toolkit	Web.xml	–
CodeIgniter	routes.php	pre_controller, post_controller
CakePHP	routes.php	beforeFilter, afterFilter
Kohana	Bootstrap.php	before, after
ASP.NET Web Forms	Global.asax	–
ASP.NET MVC	Global.asax	OnActionExecuting, OnActionExecuted
ASP.NET Web Pages	Global.asax	–
Ruby on Rails	ActionDispatch	beforeFilter, afterFilter
Django	URLconf	Middleware

Table 2. The frameworks have single points of processing determined by their design, so-called dispatchers. Some even provide filter routines that are executed before and after request processing.

4 Related Work

The *Open Web Application Security Project (OWASP)* coined the term *Failure to Restrict URL Access* [27] to describe a similar vulnerability as our control-flow weakness. However, it is more focused on access control flaws that can be exploited by *Forced Browsing attacks* [28] to find a *deep link* [29] to a high privilege web page. Workflows and control-flow integrity play a tangential role in the description.

In previous work, we developed a control-flow integrity monitor that is easily applicable to legacy and new web applications [30]. It is integrated into request processing between the central dispatcher and the controller in charge. The monitor expects a policy definition as input and provides guarantees to the web application concerning the sequence of incoming requests, their parameters and data types, as well as race condition protection. It supports multi-tabbing and usage of the "Back" button.

We divide other related work in navigation restriction means (Section 4.1), detection of server-side state violation (Section 4.2), protection against and detection of client-side manipulation (Section 4.3), and race condition detection (Section 4.4).

There are different names for the respective attacks and vulnerabilities though not big differences in their technical details. In some cases, the attack allowing a malicious user to compose his own sequence of actions is called *workflow violation attack* [31], *state violation attack* [32], *workflow attack* [33, 34] or the attack exploiting *web application logic vulnerabilities* [35].

Partial overlap exists with *HTTP parameter pollution attacks* [36] and *parameter tampering attacks* [37].

4.1 Navigation Restriction Means

These approaches restrict the web application's request surface towards the user. They limit the accepted requests to a predefined set and prevent arbitrary navigation by users.

BAYAWAK [34] is a powerful tool to enforce request integrity. The basic idea is to prevent access to all server-side resources by giving them unique temporary interface identifiers (IID). The IIDs are changed with every request. In each response, the hyperlinks carry the necessary IID to address the intended next resources. Requests to arbitrary resources are prevented due to missing identifiers. BAYAWAK appends the IID as an HTTP parameter, e.g. `?IID=x`. All necessary attributes in all web pages have to be modified to include the IID. It remains open how dynamically generated requests are equipped with the IID. By design, multitabbing and back button support as well as page reloads can not be granted as the session-bound IID must be outdated. Race condition protection depends on the actual implementation of this concept, namely whether parallel execution of requests with the same IID is possible or excluded.

Hallé et al. propose a model checking-based approach to prevent navigation errors [10]. They explain their navigation state machines that allow the execution of given actions only immediately after a preceding action. For example, the modification of user accounts is only admitted if requested right after listing all user accounts. Moreover, parameter values can be defined as a prerequisite for actions. The approach focuses more on unintentionally caused errors than on security issues based on malicious user behavior. Complete workflows can not be defined explicitly. Instead, only ordered pairs of actions can be set. Multitabbing and race conditions are not handled.

4.2 State Violation Detection

The approaches that we describe in this section aim at detecting unintended or unusual server states. The following approaches infer the intended application states during a training phase or by static code analysis. They raise an alarm as soon as the detected state deviates from the known states, but they do not intend to make workflows explicit and control the interactions with users.

MiMoSA [33] detects violations of workflow integrity if intended workflows are enforced based on PHP session variables, request parameters, and database tables. It uses a cascade of dynamic and static analysis of PHP code together with model checking techniques to identify program paths that finally lead to an insecure state – either due to workflow attacks or due to injection attacks, like Cross-Site Scripting (XSS) and SQL injection (SQLi).

Swaddler [31] detects anomalous combinations of session states and code execution points in PHP-based web applications after a learning phase. It assumes that attacks lead to observable differences in the application's state with respect to a threshold. In that sense, it is comparable to the functioning of an intrusion detection system (IDS).

BLOCK [32] follows a black box approach to detect state violation attacks based on input/output invariants. In this case, input means the requested action, input parameters, and the session state while the output is the new session state and the HTTP response. The invariants are derived during an attack-free training phase. Discrepancies between observed input/output and known invariants cause an alarm.

Waler [35] follows a similar but white box approach. It attempts to infer invariants by running dynamic analysis. Invariants are determined by `if` statements and equality relations between session variables and database entries. Finally, Waler uses model checking to find invariants-violating program paths.

4.3 Client-Side Manipulation Detection

Malicious users not only craft individual HTTP requests or manipulate request headers to achieve their goals. Depending on the business logic of the web application, changes on the client-side JavaScript code can cause damage to the application provider. The following approaches aim at detecting several kinds of client-side manipulation.

PAPAS [36] falls into the category of the above mentioned OWASP attack classes. It discovers HTTP parameter pollution vulnerabilities in web applications. The approach is to some extent similar to intelligent fuzzing attempts.

Ripley [38] replicates the client-side execution of JavaScript code on a server-side replica and, thus, detects manipulations, e.g. on AJAX requests. It causes a higher load on client- and server-side as well as an additional delay of responses.

Guha et al. [39] make use of static analysis to obtain a model of expected client behavior from the server's point of view. All requests not matching this expected behavior are considered harmful and are dropped.

NoTamper [37] detects differences in server-side and client-side validation of user input and HTTP parameters. Finally, if an input, that is rejected on client-side, gets accepted on server-side, a possible attack vector may exist by manipulating or disabling client-side checks. This approach is similar to Ripley [38] which however is supposed to be applied for new applications while NoTamper is meant for legacy applications that are considered as a black box.

4.4 Race Conditions

Race conditions [9] are explained in detail in Section 2.3. An attacker exploiting this vulnerability can execute one function more often than intended by the application developer.

Paleari et al. [1] describe an approach to detect race condition vulnerabilities in LAMP[7]-based web applications. They dynamically log SQL queries at runtime and analyze the log file to find possible race conditions based on the series of SQL clauses.

[7] LAMP stands for *Linux, Apache, MySQL, PHP*, the classical web server architecture.

5 Conclusion

We explained the complex problem of control-flow weaknesses and showed its high practical relevance by real-world examples, i.e. existing vulnerabilities and attacks. We identified the root causes in the modular addressability of web applications together with the implicit and scattered definition of workflows. Our findings on the current support for control-flow integrity in the most prevalent web application frameworks show that this problem does not yet receive the attention it deserves. All frameworks but Spring with the Web Flow project lack related properties. No framework provides race condition protection features beyond single flow request sequences. Only type safety of received HTTP parameters is commonly supported.

Acknowledgments

This work was in parts supported by the EU Project WebSand (FP7-256964), `https://www.websand.eu`. The support is gratefully acknowledged.

References

1. Paleari, R., Marrone, D., Bruschi, D., Monga, M.: On Race Vulnerabilities in Web Applications. In: DIMVA. (2008)
2. Chen, S.: Session Puzzles - Indirect Application Attack Vectors. [White Paper], `http://puzzlemall.googlecode.com/files/Session%20Puzzles%20-%20Indirect%20Application%20Attack%20Vectors%20-%20May%202011%20-%20Whitepaper.pdf`,(05/23/12)
3. Grossman, J.: Seven Business Logic Flaws That Put Your Website At Risk. [White Paper], `https://www.whitehatsec.com/assets/WP_bizlogic092407.pdf`,(05/19/12)
4. The New York Times: Thieves Found Citigroup Site an Easy Entry. [online], `http://www.nytimes.com/2011/06/14/technology/14security.html`,(05/24/12)
5. Wang, R., Chen, S., Wang, X., Qadeer, S.: How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In: IEEE Symposium on Security and Privacy. (2011)
6. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, `http://www.w3.org/Protocols/rfc2616/rfc2616.html` (June 1999)
7. Berners-Lee, T., Fielding, R., Irvine, U., Masinter, L.: Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, `http://www.ietf.org/rfc/rfc2396.txt` (August 1998)
8. Jovanovic, N., Kruegel, C., Kirda, E.: Preventing cross site request forgery attacks. In: Securecomm. (2006)
9. OWASP: Race Conditions. [online], `https://www.owasp.org/index.php/Race_Conditions`,(05/23/12)
10. Hallé, S., Ettema, T., Bunch, C., Bultan, T.: Eliminating Navigation Errors in Web Applications via Model Checking and Runtime Enforcement of Navigation State Machines. In: ASE. (2010)

11. builtWith: Framework Usage Statistics – Overview of Statistics for Framework Technologies. [online], http://trends.builtwith.com/framework
12. Johnson, R.E., Foote, B.: Designing Reusable Classes. In: Journal of Object-Oriented Programming. Volume 1. (1988)
13. The Apache Software Foundation: Tapestry. [online], http://tapestry.apache.org/
14. Google, Inc.: Google Web Toolkit. [online], https://developers.google.com/web-toolkit/
15. SpringSource: The Spring Framework. [online], http://www.springsource.org/
16. EllisLab, Inc.: CodeIgniter. [online], http://ellislab.com/codeigniter
17. Cake Software Foundation, Inc.: CakePHP. [online], http://cakephp.org/
18. Kohana Team: Kohana. [online], http://kohanaframework.org/
19. Microsoft: ASP.NET. [online], http://www.asp.net/
20. Microsoft: ASP.NET Web Forms. [online], http://www.asp.net/web-forms
21. Microsoft: ASP.NET MVC. [online], http://www.asp.net/mvc
22. Microsoft: ASP.NET Web Pages. [online], http://www.asp.net/web-pages
23. David Heinemeier Hansson: Ruby on Rails. [online], http://rubyonrails.org/
24. Django Software Foundation: Django. [online], https://www.djangoproject.com/
25. Mozilla Developer Network: AJAX. [online], https://developer.mozilla.org/en-US/docs/AJAX
26. Spring Projects: Spring Web Flow. [online], http://www.springsource.org/spring-web-flow
27. OWASP: Failure to Restrict URL Access. [online], https://www.owasp.org/index.php/Top_10_2010-A8-Failure_to_Restrict_URL_Access,(05/11/12)
28. OWASP: Forced Browsing. [online], https://www.owasp.org/index.php/Forced_browsing,(05/04/12)
29. Bray, T.: Deep Linking in the World Wide Web. [online], http://www.w3.org/2001/tag/doc/deeplinking.html (05/29/12)
30. Braun, B., Gemein, P., Reiser, H.P., Posegga, J.: Control-Flow Integrity in Web Applications. In: ESSoS 2013, LNCS, Springer (2013)
31. Cova, M., Balzarotti, D., Felmetsger, V., Vigna, G.: Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In: RAID. (2007)
32. Li, X., Xue, Y.: BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In: ACSAC. (2011)
33. Balzarotti, D., Cova, M., Felmetsger, V., Vigna, G.: Multi-Module Vulnerability Analysis of Web-based Applications. In: CCS. (2007)
34. Jayaraman, K., Lewandowski, G., Talaga, P.G., Chapin, S.J.: Enforcing Request Integrity in Web Applications. In: DBSec. (2010)
35. Felmetsger, V., Cavedon, L., Kruegel, C., Vigna, G.: Toward Automated Detection of Logic Vulnerabilities in Web Applications. In: USENIX Security. (2010)
36. Balduzzi, M., Gimenez, C.T., Balzarotti, D., Kirda, E.: Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In: NDSS. (2011)
37. Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., Venkatakrishnan, V.N.: No-Tamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In: CCS. (2010)
38. Vikram, K., Prateek, A., Livshits, B.: Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution. In: CCS. (2009)
39. Guha, A., Krishnamurthi, S., Jim, T.: Using Static Analysis for Ajax Intrusion Detection. In: WWW. (2009)