Control-Flow Integrity in Web Applications

Bastian Braun, Patrick Gemein, Hans P. Reiser, and Joachim Posegga

Institute of IT-Security and Security Law (ISL), University of Passau
{bb,hr,jp}@sec.uni-passau.de, pgemein@gmx.de

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. Based on this result, we provide our approach, a control-flow monitor that is applicable to legacy as well as newly developed web applications. It expects a control-flow definition as input and provides guarantees to the web application concerning the sequence of incoming requests and carried parameters. It protects the web application against race condition exploits, a special case of control-flow integrity violation. Moreover, the control-flow monitor supports modern browser features like multi-tabbing and back button usage. We evaluate our approach and show that it induces a negligible overhead.

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].

This paper presents novel approaches for avoiding problems related to control-flow integrity in web applications. The specific contributions are as follows. First, we define a formal language for specifying explicitly the control flow of a web application; Second, we define a control mechanism that makes sure that only client requests that comply with the control-flow specification are executed; Third, we integrate the control mechanism in a framework based on the Model-View-Controller (MVC) model, making our approach both easy to use for newly developed applications and easy to integrate in already existing applications. Finally, we show that our approach is effective and practical by demonstrating that it enables the removal of several kinds of real-world security problems, while having a low run-time overhead.

This paper is structured as follows. The next section provides an in-depth discussion of technical aspects of control-flow integrity in web applications and explains known attacks and vulnerabilities. Section 3 presents novel approaches for controlling flow integrity at the server-side. Section 4 analyzes the benefits of our approach and evaluates the performance of our prototype. Section 5 compares our approach to related work, and Section 6 concludes.

2 Web Application Control Flow

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

In a typical web application, 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. Dynamic web applications, however, have workflows that are composed of multiple steps, which corresponds 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.

2.2 The Attacks

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 [8] in web applications, attackers can send several crafted requests almost in parallel. If the web application does not handle concurrent requests by proper synchronisation, 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.

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. In 2011, the Citigroup faced an attack on their customers' data [4]. The attackers were able to access names, credit card numbers, e-mail addresses and transaction histories. All the attackers had to do was simply changing the HTTP parameters in the web browser. By automation, they obtained confidential data of more than 200,000 customers.

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.

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. [9] describe related navigation errors.

2.3 Root Causes

All described attacks share common root causes. 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. In the respective control-flow graph, this transition does not exist 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 time while control-flow integrity protection prohibits access to regular API methods at the wrong time.

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 have shown by the attacks in Section 2.2.

3 Preserving Control-Flow Integrity

The attacks described in Section 2.2 are caused by user actions that violate given control flows. This section provides detailed information of how we prevent an unintended action from getting executed and, thus, from violating the integrity of a control flow.

3.1 Technical Background

For every web application, the application developer knows the intended control flow. This control flow can be denoted as a sequence of actions. Considering each action as a transition in a graph, we finally obtain the control-flow graph of the web application. So, the application developer deploys the control-flow graph of the web application.

The enforcement of the intended control flow requires a central entity that takes care of each incoming request. The popular Model-View-Controller architecture provides such an entity by design (see Figure 1). Every request has to pass the application's controller, which encapsulates the business logic. The controller consists of several classes, each containing various methods. Therefore, one action of a control flow in our definition language is defined as `<class name>.<method name>`. From a granularity view, this is appropriate because a request addresses one method. In sum, a control-flow graph is given as a sequence of methods of controller classes.

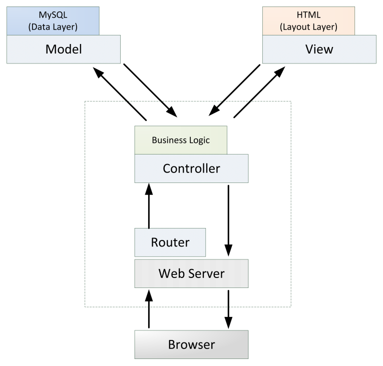

Fig. 1. The Design Pattern of MVC-based Web Applications

3.2 Protection Goals

Our approach protects web applications from malicious users that perform attacks using arbitrary request sequences. As a side effect, the approach protects honest users against *Cross-Site Request Forgery (CSRF)* attacks to some extent because attackers have to follow the intended control flow to finally commit their abusive request. In more detail, our approach has the following goals:
– Upon each incoming request, the monitor shall determine the control-flow context of this request and take a decision whether the request is permitted. If the monitor allows the request to pass, it updates the context accordingly.
– The approach must be usable with state-of-the-art browser features, including the use of a back button as well as multiple browser tabs (multi-tabbing) for the same session. Each tab shall be permitted to use a different control flow.
– The monitor must prevent race conditions for actions that might serve as a target for an attacker. These actions can be specified in the control-flow graph.
– The monitor must be able to control HTTP parameters and their values.
– All web applications have unclassified resources such as "About us" information. These resources shall be accessible without restrictions, independent of ongoing workflows.

3.3 Enforcing Control-Flow Integrity

In this section, we provide details how we achieve the above-mentioned goals. We propose an architecture based on explicit control-flow specification and server-side control-flow enforcement. The central control-flow monitor combines several mechanisms to enforce control-flow integrity. We show that all user interactions are intercepted and checked by our control-flow monitor. Besides simple checks that sequences of requests are compatible with sequences in the control-flow graph, several situations require dedicated treatment, as explained in the following.

Back Button Support A widespread feature of modern browsers is the back button that allows the user to view the last web page again. As users are used to click that button whenever they feel like revisiting the last page, we implemented support for this step in our monitor. Therefore, the control-flow monitor records the trace of steps of the user. A request is considered a step backwards if it addresses the last method and this method is not meant as a next step in the control-flow graph. However, the control-flow monitor by default prohibits the backwards traversal due to the issues described in Section 2.2. Instead, the usage of the back button has to be allowed in the control-flow graph for each step.

Multi-Tabbing Support Modern web browsers usually allow several tabs in the same window. As these tabs share the client-side data, e.g. cookies [10], across all instances, they are hardly distinguishable from the server side. Hence, without multi-tabbing support, actions in one tab would violate the control flow in another. In order to overcome this drawback, the control-flow monitor inserts client-side identifiers for different tabs to tell them apart. This way, each tab can be treated individually though logged in at the same web application.

Race Condition Prevention The monitor prevents the exploitation of race condition vulnerabilities (see Section 2.2), by disabling parallel execution of susceptible actions. In general, these are actions that add, update, or delete data after reading. We achieve this goal with a locking mechanism. The control-flow monitor creates a temporary lock named by the session ID of the user. This means race condition protection on session level. Moreover, protection on control-flow and user level is possible by using a control-flow ID and the user ID respectively. Even a system-wide protection can be implemented using one unique ID file for all users.

Parameter Validation The client-side manipulation of HTTP parameters can lead to unintended application states (see Section 2.2). Thus, request parameters have to be checked for validity on the server side before they are processed. Instead of leaving this task to each method, the control-flow monitor provides means to centrally enforce given parameter properties. First, the data type of each parameter can be defined. As a side effect, this feature also mitigates *injection attacks (XSS, SQLi)* that need to transmit control characters. Second, parameters can be marked as “write once read many” (WORM). This allows to set the parameter’s value once but not change it afterwards, meaning that this value is immutable for the rest of the session. This provides an invariant guarantee to the web application. One use case is the user ID that is supposed to not change during a session. Third, parameter names can be excluded for given workflows. This feature can protect web applications from unintended data manipulation. For instance, it prevents the setting of control flow-invariant parameters.

Definition of Uncritical Methods All web applications contain uncritical methods. Accessing these methods does not harm the application's control-flow integrity. For instance, a chat function can be allowed beside the enforced workflow. Similarly, AJAX calls that update the user's view but do not change the application's state can also be allowed.

Control-Flow Definition In this section, we provide details on the syntax of the control-flow graph definition language. The following clauses and operators can be combined recursively.

`Method1` → `Method2` — After accessing `Method1`, the user is allowed to access `Method2`.

`(Method1|Method2)` — The user is allowed to access `Method1` or `Method2` in the first place, but she is not allowed to change her decision after clicking the back button.

`(&Method1|&Method2)` — Like above but the user is allowed to change her decision after clicking the back button – denoted by the `&` symbol.

`@Method{x}` — The user is allowed to access `Method` repeatedly. It is possible to define a maximum number `x` of allowed executions.

`?Method` — The back button support for `Method` is enabled, i.e. the user can navigate one step backwards after having called this method.

`!Method` — The race condition protection is active for this method. As long as this method is executed, no other protected method is executed in the context of the same session, user account, or system-wide (see above).

`Method[+par1=type1,*par2=type2]` — Only parameters `par1` and `par2` are allowed for `Method` where they can be sent via POST (`+`) or GET (`*`) and have data types `type1` and `type2` respectively. Predefined data types include `bool`, `numeric`, and `string`. A policy for the whole control flow can be set by `addParameterTypeGlobal("*par=type")`.

`addForbiddenParameters("par")` — Parameter `par` must not occur in the whole control flow.

`addParametersGlobal("par")` — Parameter `par` can be set once but is immutable afterwards.

The nesting of clauses allows for defining complex control-flow policies. We provide simple examples in Section 3.5 and a more sophisticated case in Section 4.

3.4 The Implementation

For implementing our control flow monitor, several challenges need to be addressed. Most importantly, the monitor has to be integrated into an application framework, which can be a complex task especially for existing applications. In addition, handling race conditions and multitabbing also deserve more detailed attention.

Integration into Web Applications We implemented our control-flow monitor as a PHP module. It is run by the router (see Figure 2) before the controller class is called. This strategic position makes sure that, first, all requests have to pass our control-flow monitor before being processed by the web application and, second, the monitor is easy to integrate into existing web applications.

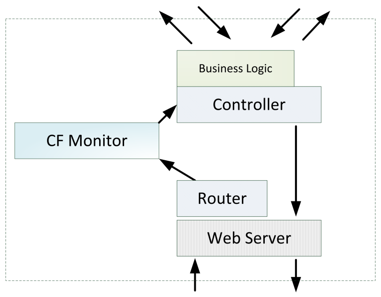

Fig. 2. Implemented Modification of the Design Pattern of MVC-based Web Applications w.r.t. Figure 1

As a proof of concept, we integrated the monitor into a web application that is based on the CodeIgniter framework [11]. In fact, the only change on an existing web application affects the one line of code that calls the responsible controller. This line has to be slightly modified to include our monitor (see Listing 1.1).

```
include(APPPATH.'controllers/'.$RTR->fetch_directory().$RTR->
    fetch_class().'.php');
//must be changed to
AOP::process(APPPATH.'controllers/'.$RTR->fetch_directory().
    $RTR->fetch_class().'.php',
  $_SESSION[''atom_parentFramework'']->getCacheFolderName());
```

Listing 1.1. Dynamic Inclusion of Controller Classes in the CodeIgniter Framework [11]

We use *Aspect Oriented Programming (AOP)* to inject the control-flow monitor as a processing step into the call sequence of all controllers. This allows the developer to apply changes on the application the same way as if there were no control-flow monitor.

Multi-Tabbing Support As explained before, multi-tabbing requires the unique identification of tabs. This identification is implemented in JavaScript. Moreover, a tab handler is implemented on server side as part of the control-flow monitor. The JavaScript code triggers an AJAX message whenever a tab is opened, closed, or a tab switch is performed by the user. A tab switch message by the client makes the tab handler change the tab context on server side. When the user opens a new tab by clicking on the "open link in new tab" option in the browser, this tab is assigned a session-unique identifier. We use the `window.name` property of the `window` DOM object to store the identifier. An AJAX request transmits the new identifier to the tab handler. The new tab is assigned the advanced position in the control-flow graph while the first tab holds the former position. Both tabs then run the same control flow, however, it is enforced individually, i.e. a control-flow violation in one tab has no effect on the other tab as the respective tab record is duplicated when the new tab is opened.

The control-flow monitor stores flow-related information per tab, i.e. the active control-flow graph that is currently enforced in this tab and the respective position in the graph. The user's session ID and other high level information is still stored in the session record. This allows, for instance, to consider several products in different tabs, then add some of them in the same shopping cart and finally check out in one tab that starts the checkout control flow.

An attacker stripping or manipulating the embedded tracking code can not trick the system to gain advantages. The code only signals the current tab to the web application. A manipulation would cause the web application to assign the next request to a different tab. This, however, is equivalent to perform the request in the respective tab. The intended action is only executed if the request is allowed there. Then, however, the attacker has not increased his scope of action. In all other cases, the manipulation leads to voiding the current control flow.

Race Condition Prevention Whenever a protected method is executed, the control-flow monitor tries to create a file with the current session ID. If this creation fails due to an existing file with the same name, the request is not processed and an error page is displayed. After processing the protected method, the file lock is released again. This allows the next protected method to be executed.

The race condition protection mechanism does not prevent the processing of unprotected methods, e.g. in a different tab. The fine granularity of the locking makes sure that a single locked method has no impact on the usability of other sessions of the user or the interactions of other users.

3.5 Simple Examples

In this section, we show the usage of our control-flow definition language. We give examples with respect to the real-world scenarios in Section 2.2 but assume a simplified technical implementation to keep the control flows simple and clear. We give details on the application of our control-flow monitor in the context of the Amazon checkout process in Section 4.

Preventing Race Conditions in SMS Delivery In the first example [1], attackers managed to bypass the delivery limit of an SMS portal by exploiting a race condition vulnerability. We assume the following control flow to send an SMS: First, the user requests the SMS input form. Then, after entering all necessary information, the user submits the form. The related control-flow definition ensures that, first, the input form has to be accessed before the submission, and, second, the submission must be protected against race condition attacks, see Listing 1.2.

```
SMS.showForm -> !SMS.validateAndSendForm
```

Listing 1.2. Control-Flow Definition to Protect SMS Delivery from Race Conditions

The control-flow definition allows access to the method `validateAndSendForm` only after requesting `showForm`. This prevents the attacker from sending the message information directly to the delivery gateway. Of course, a capable attacker might send the requests to the `showForm` method in an automated fashion. However, as the `validateAndSendForm` method is protected against race condition attempts, e.g. on user level, the attacker's requests will only be processed sequently. This avoids sending more messages than actually allowed.

Prevent Adding Items to the Shopping Cart Between Checkout and Payment A more complex example is given by Wang et al. [5]. After requesting the checkout, the user was able to add more items to her shopping cart. These items were not charged. In order to prevent this sequence of requests, the checkout workflow has to be properly defined. The method that adds goods to the cart must not be accessible during this workflow. The respective control flow definition is given in Listing 1.3.

```
Checkout.logIn
-> Payment.chooseMethod
-> Payment.validateStatus
-> Checkout.completeOrder
```

Listing 1.3. Control-Flow Definition to Prevent Adding Items after Checkout

After the authentication, i.e. login, the user chooses her favorite payment method and is redirected to a payment service provider. The actions on the payment service provider's site are not part of the definition because they happen on a different domain that is not controlled by the same control-flow monitor. The next request within the scope of the definition is the payment status validation after the user's return. Finally, the order is completed, the goods are shipped to the user, and the cart is reset. During the whole process, no addition of items to the cart is granted.

4 Discussion & Evaluation

In this section, we discuss the properties of our control-flow monitor. We show that it produces a negligible overhead and evaluate the protection goals defined in Section 3.2. Finally, we explain its possible application scenarios and limits.

4.1 Performance Evaluation

As described in Section 3.4, the control-flow monitor is applied between the router and the controller of the web application. It examines the received HTTP request with respect to the requested method and the parameters and checks these against a given policy. This application overhead is independent of the web application's execution time. The delay relates to the complexity of the given policy, though.

We used Xdebug (version 2.1.2) [12] to determine the control-flow monitor's overhead in a virtual machine with Debian 6 as the operating system and Apache2 as the web server with PHP 5.5.3.3-7 on an Intel Core-i7-2600 (Intel-VT activated) with 3.4 GHz and 2 GB RAM. For evaluation purposes, we implemented the checkout process of Amazon. Therefore, we analyzed the control flow on `amazon.com` by hand and derived the control-flow definition given in Listing 1.4. Note that the controller and method names are simplified for readability reasons. The control-flow definition does not allow usage of the back button because Amazon prohibits it, too.

```
1 Login.index
2 -> (Address.chooseExisting|Address.addNew)
3 -> Shipping.preferences
4 -> ((Payment.chooseExisting
   |Payment.addNewCreditCard)
   |Payment.addNewDebitCard)
5 -> (Billing.chooseExisting|Billing.addNew)
6 -> Order.placeOrder
```

Listing 1.4. Definition of Amazon's Checkout Control Flow

	Runs										
Step	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	avg
1	8.9	8.2	8.4	10.2	11.0	8.7	8.2	9.4	8.3	7.7	8.9
2	10.2	9.9	9.3	9.8	10.1	9.8	9.0	8.2	9.5	9.1	9.5
3	10.1	9.2	10.9	8.6	9.6	8.2	9.5	9.0	9.0	8.4	9.2
4	8.3	10.1	10.0	10.2	9.4	9.8	10.3	7.8	10.6	9.0	9.6
5	8.8	11.0	10.1	8.3	8.4	10.0	8.6	7.9	7.7	9.8	9.1
6	10.0	8.5	8.1	8.5	8.4	8.4	9.6	9.7	8.0	10.4	9.0

Table 1. Overhead caused by the control flow monitor in ms

We measured the runtime overhead ten times and computed the average for each step in the control flow (see Table 1). The respective graph shows a peak in the fourth state due to the triple branching (see Figure 3). Branches, namely alternative paths through the control flow, cause most of the overhead, the earlier a branch occurs the bigger its overhead. This is the reason why step 2 causes more overhead than step 5. We assume that some overhead can be saved by a more efficient policy parsing algorithm. Overall, the induced delay ranges between 8.9 and 9.6 milliseconds per request. We consider this an acceptable effort with respect to the security gain. In order to determine the monitor's scalability to several user sessions, we set up 100 parallel user sessions

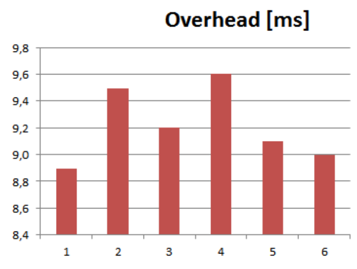

Fig. 3. Performance Evaluation of the Amazon Checkout Process

and repeated the measurement. While the overall response time increased, we found out that there is no measurable difference to the scenario with only one user in terms of the monitor's overhead.

There is a one-time overhead for the generation of the temporary controller class file (see Section 3.4). This overhead occurs once whenever a new controller class is added or an existing class is modified. The first call on this class takes 60% to 90% more time than the subsequent calls. For usability concerns, this overhead can be neglected because the web application provider could easily initiate an appropriate request, thus, preventing all users from facing the mentioned delay.

4.2 Discussion

In this section, we evaluate our findings with respect to the protective goals defined in Section 3.2. We have to note that the monitor is responsible for control-flow integrity while other tasks like session management and user authentication are handled by the framework in place.

Every incoming HTTP request has to pass the router in the assumed MVC architecture. So, all requests are finally processed by our control-flow monitor. Our security evaluation showed that in fact all requests are treated by the monitor and accepted or rejected appropriately. The control-flow monitor achieves complete protection against maliciously crafted requests as well as erroneous navigation attempts.

However, the protection level depends on the sound definition of control flows. The definition has to be provided by the application developer. The implications from this fact are twofold. First, the definition requires a deep knowledge of the web application and its methods. The knowledge and understanding of the web application must already exist to implement and maintain the web application. This allows developers to provide more accurate control flow policies than automatic approaches. So, we consider this a feasible task for an expert. Second, the necessary policy definition efforts stay within reasonable bounds. The `class.method`-based policy language abstracts from the implementation of functional modules but is still close to the web application's architecture. In order to estimate the complexity in real world use cases, we crafted the control flow policy for the checkout of the open source shop Magento [13] (see Listing 1.5).

```
Mage_Checkout_CartController.indexAction ->
Mage_Checkout_OnepageController.indexAction ->
Mage_Checkout_OnepageController.saveBillingAction ->
Mage_Checkout_OnepageController.saveShippingAction  ->
Mage_Checkout_OnepageController.saveShippingMethodAction  ->
Mage_Checkout_OnepageController.savePaymentAction ->
Mage_Checkout_OnepageController.reviewAction  ->
Mage_Checkout_OnepageController.successAction
```

Listing 1.5. Control-Flow Definition of the Magento [13] Online Shop

Our control-flow monitor provides multi-tabbing and back button support, thus proves usable with modern browser features. This increases usability and ensures acceptance by end users. This way, security is not achieved at the expense of a limited user experience.

To the best of our knowledge, our approach is the first to effectively protect against race condition exploits. The control-flow monitor allows the flexible definition of the protection level, ranging from control flow-based over user-level up to system-wide protection.

Policies on HTTP parameters can be defined including both GET and POST parameters. Policy rules can apply in terms of data type, the limitation on a single value assignment, and the exclusion of parameters for given workflows. Our parameter control means are suitable to prevent the attacks described in Section 2.2.

The definition of uncritical methods allows control-flow integrity to focus on a comprehensible set of relevant method calls. For instance, there can be unhindered access to pictures because they are not part of the business logic. Confidential data can be protected by access control means. AJAX requests can be divided into state-changing and other requests. The state-changing requests can be covered by the control-flow definition, the others are excluded and pass the control-flow monitor. As AJAX requests also call server-side methods, their control-flow definition is straightforward with respect to the web application's control flow.

Our approach is easily applicable at development phase though one of its most advantageous features is its usability with legacy web applications. We implemented a PHP-based proof of concept. Nevertheless, a Java-based implementation can be achieved with acceptable effort, e.g. by a J2EE filter. Even non-MVC-based web applications can be equipped with the monitor. However, the integration causes more overhead if a central request handler is missing. Then, all calls on server-side actions have to be intercepted separately.

The control-flow monitor does not aim at replacing the web application's business logic. As a matter of fact, it provides reasonable and reliable guarantees concerning the sequence of requests and properties of provided parameters. The web application still has to make sure that user generated content fits the expected information. For instance, a sequence of requests containing semantic garbage but matching the defined control flow will still succeed to finally request the intended method.

5 Related Work

The *Open Web Application Security Project (OWASP)* coined the term *Failure to Restrict URL Access* [14] 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* [15] to find a *deep link* [16] to a high privilege web page. Workflows and control-flow integrity play a tangential role in the description.

To the best of our knowledge, there is no work with a similar scope of protection and a comparable feature set. The approaches that restrict the web application's request surface towards the user are considered most similar to our approach. They limit the accepted requests to a predefined set and prevent arbitrary navigation by users. This is either achieved by issuing tickets to access server-side resources [17] which, by design, inhibits multitabbing and back button support as well as page reloads, or by defining pairs of steps that can be executed in order [9] where the first step serves as a gatekeeper to the second step. The latter approach does not allow to define complete workflows explicitly what we identified as a crucial point.

Other approaches aim at detecting unintended or unusual server states [18], combinations of such server states and code execution points [19] or code execution paths that lead to the violation of application invariants [20] or input/output invariants [21]. These approches infer the intended application states during a training phase or by static code analysis. They do not intend to make workflows explicit and control the interactions with users.

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. Existing approaches replicate client-side computation on server-side to detect deviations [22, 23], statically analyze JavaScript to determine the expected sequence of requests [24], or check the web application for exploitable HTTP parameter pollution vulnerabilities [25].

An attacker exploiting a race condition vulnerability [8] 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 web applications.

6 Conclusion

We explained the complex problem of control-flow vulnerabilities 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 solution overcomes this problem by the explicit definition and enforcement of intended workflows. To the best of our knowledge, it is the first approach that covers the whole bandwidth of related vulnerabilities, including race conditions, HTTP parameter manipulation, unsolicited request sequences, and the compromising use of the back button. Moreover, it is the first approach that properly handles client-side features like back button usage and multitabbing. We showed that this approach can prevent all described attacks and causes negligible overhead.

In sum, we provided a thorough approach that is applicable to existing and newly developed web applications and provides guarantees to the developer concerning the sequences of incoming requests as well as the format and values of parameters. This allows to separate web application semantics and control-flow integrity. As a side effect, the presented approach mitigates *Cross-Site Request Forgery (CSRF)* and *injection (XSS, SQLi)* attacks.

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. OWASP: Race Conditions. [online], https://www.owasp.org/index.php/Race_Conditions,(05/23/12)
9. 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)
10. Kristol, D., Montulli, L.: HTTP State Management Mechanism. RFC 2109, http://www.ietf.org/rfc/rfc2109.txt (February 1997)
11. ExpressionEngine Dev Team: CodeIgniter - Open Source PHP Web Application Framework. [online], http://www.codeigniter.com/,(05/24/12)
12. : Xdebug. [online], http://xdebug.org/,(06/05/12)
13. : Magento Commerce. [online], http://demo.magentocommerce.com/,(09/24/12)
14. 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)
15. OWASP: Forced Browsing. [online], https://www.owasp.org/index.php/Forced_browsing,(05/04/12)
16. Bray, T.: Deep Linking in the World Wide Web. [online], http://www.w3.org/2001/tag/doc/deeplinking.html (05/29/12)
17. Jayaraman, K., Lewandowski, G., Talaga, P.G., Chapin, S.J.: Enforcing Request Integrity in Web Applications. In: DBSec. (2010)
18. Balzarotti, D., Cova, M., Felmetsger, V., Vigna, G.: Multi-Module Vulnerability Analysis of Web-based Applications. In: CCS. (2007)

19. 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)
20. Felmetsger, V., Cavedon, L., Kruegel, C., Vigna, G.: Toward Automated Detection of Logic Vulnerabilities in Web Applications. In: USENIX Security. (2010)
21. Li, X., Xue, Y.: BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In: ACSAC. (2011)
22. Vikram, K., Prateek, A., Livshits, B.: Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution. In: CCS. (2009)
23. Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., Venkatakrishnan, V.N.: NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In: CCS. (2010)
24. Guha, A., Krishnamurthi, S., Jim, T.: Using Static Analysis for Ajax Intrusion Detection. In: WWW. (2009)
25. Balduzzi, M., Gimenez, C.T., Balzarotti, D., Kirda, E.: Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In: NDSS. (2011)