Ghostrail: Ad Hoc Control-Flow Integrity for Web Applications

Bastian Braun, Caspar Gries, Benedikt Petschkuhn, Joachim Posegga

Ghostrail: Ad Hoc Control-Flow Integrity for Web Applications

Bastian Braun, Caspar Gries, Benedikt Petschkuhn, and Joachim Posegga

Institute of IT Security and Security Law (ISL), University of Passau, Germany
{bb,jp}@sec.uni-passau.de, cgries@in-doc.de, petschku@fim.uni-passau.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 enforcement of allowed next user requests. Based on this result, we provide our approach, named *Ghostrail*, a control-flow monitor that is applicable to legacy as well as newly developed web applications. It observes incoming requests and lets only those pass that were provided as next steps in the last web page. Ghostrail protects the web application against race condition exploits, the manipulation of HTTP parameters, unsolicited request sequences, and forceful browsing. We evaluate the approach and show that it neither needs a training phase nor a manual policy definition while it is suitable for a broad range of web technologies.

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, i.e., to first access the website on an entry page and then proceed by clicking on links and buttons and fill provided forms. 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. Our previous work showed that common web application frameworks provide hardly any protection means a developer could rely on [1], meaning that developers must ensure control-flow integrity manually.

Several known attacks have exploited missing protection in the past. The attacks' impact ranges from sending more free SMS text messages than actually allowed [2], and unauthorized access to user accounts [3–5], to shopping for expensive goods with arbitrarily low payments [6]. This paper presents a novel approach for avoiding problems related to control-flow integrity in web applications. Based on the assumption that all client-side requests are potentially compromised, the approach replicates a user's mouse clicks and form input in a server-side sandbox. Requests triggered by the sandbox are trustworthy because they adhere to the assumed user interaction with the web application. We show that this approach provides ad hoc protection against attacks on the web application's control flow without the need for a learning phase or a manual policy definition. It functions as a reverse proxy and is thus independent of the server-side technology. No adaptations are required on the web application making the approach applicable to new and legacy applications. The induced load can be outsourced to scalable, e.g. cloud-based, platforms if necessary.

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 our novel approach for controlling flow integrity at the server-side, and Section 4 gives details about the implementation. We evaluate the approach in Section 5, discuss related work in Section 6, and finally conclude in Section 7.

2 Control-Flow Integrity

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. 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 of the HTTP request, the request parameters, and the server-side session record.

2.2 The Attacks

Several kinds of attacks on web applications 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 forceful browsing.

Race Conditions In order to exploit race conditions [7] 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 [2]. 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. [6] 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.

File Inclusion Attacks are a special kind of HTTP parameter manipulation. The successful attacker gains access to protected resources be it static documents or application functions. For instance, an application might offer the URL `http://example.com/?view=welcome.html` as a hyperlink. In this case, the `view` parameter holds the name of a file that is supposed to be included in the output. An attacker can change the parameter value to `/etc/passwd`. He succeeds if the application fails to detect the manipulation. A successful file inclusion attack allows to access all files that are readable to the web server process.

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. [6], 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 shipped but not invoiced.

Forceful Browsing A forceful browsing attacker exploits predictable naming schemes in combination with insufficient access control. For instance, left installation scripts for PHP-based web applications are popular targets for forceful browsing attacks. An administrator uploads such a script to the web server and calls it via her browser in order to install a web application. Attackers can call the installation script and reconfigure the application if the administrator forgets to delete it. In 2010, a group of attackers gained access to 114,000 user records of iPad owners by requesting a server-side script that was supposed to embed user details into a web page [5]. The attackers could easily guess the naming scheme and access restricted application functions.

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 client. 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.

Forceful browsing attacks and some cases of HTTP parameter manipulation can be overcome with access control. The other attack vectors, however, 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.

3 Preserving Control-Flow Integrity Ad Hoc

In this section, we present *Ghostrail*, our approach to overcome the attacks described in Sec. 2. We give details on the implementation in Sec. 4.

3.1 High-Level Overview

The idea behind Ghostrail is the ad hoc enforcement of control-flow integrity based on the developer's assumptions phrased in Sec. 2.3: Users first request one entry point of the web application, e.g. `www.example.com`, and then click on links and buttons or fill in forms. We assume that the attacker is a web user that controls all client-side data and applications within his domain. However, he can not bypass reverse proxies. He can send messages to the server but does not control the server-side platform.

Ghostrail operates as a traffic monitor on the server side. It protects a web application against the attacks described in Sec. 2.2 by filtering out incoming requests that are not generated by user clicks and form entries. In order to determine whether a request arises from regular interaction between the user and the current web page, Ghostrail analyzes the last web page delivered by the web application. A request is accepted if the web page contained the respective link. Otherwise – the page did not contain the requested URL – the request is considered crafted and, thus, possibly malicious because it violates the assumption that the user only interacts with the web application using clicks and filling in forms. Ghostrail has a three-tier whitelisting approach to derive regular requests:

- First, Ghostrail queries an application-wide whitelist of always allowed requests. The list contains the application's entry points, e.g. the start page, and possibly all requests to public resources that do not change the application's state.
- Second, Ghostrail parses the last web page delivered by the web application. It compiles a list of static references found in HTML and CSS documents. Those references denote hyperlinks, i.e., possible next user clicks, or embedded resources that are needed by the browser to render the web page, e.g. images. We give more details on static reference extraction in Sec. 3.2.
- Third, Ghostrail renders web pages in server-side sandboxes to determine dynamically generated requests, e.g. using AJAX and JavaScript. Those requests can not be determined by the static parser in step 2. Ghostrail accepts requests from client side if the sandbox triggered the same request. We describe the sandbox-based request detection in Sec. 3.3.

Ghostrail lets only requests found in any of these three lists pass. This way, it enforces the assumption that users do only interact with the web application using mouse clicks and form input. Due to the fact that the whitelists in step 2 and step 3 are compiled ad hoc, Ghostrail neither needs a pre-release learning phase to generate its control-flow policy nor a hand-crafted control-flow definition. However, as Ghostrail operates on automatically generated whitelists of references, every reference it fails to extract may degrade the usability of the web application. It is therefore crucial to extract as many references as possible. By extraction we mean the analysis, classification and storage of reference information that is embedded in content delivered by the web application. In the remainder of this section, we provide details on how Ghostrail extracts references statically and dynamically.

3.2 Extraction of Static References

In this section, we give details on how Ghostrail extracts static URLs from delivered web pages. Static references usually occur in HTML and CSS files. Other web resources like JavaScript and Flash, i.e., ActionScript, mainly utilize dynamic URL generation. They assemble the requests based on user input or client-side state. We explain dynamic URL tracking in the next section. Finally, media files like images are ignored because they do not contain links for subsequent requests.

HTML is a tag-based language, i.e., the elements of a web page are described as tags. There is a limited number of HTML tags that may contain URLs: `<a>`, `<link>`, `<iframe>`, `<script>`, `<img>`, `<area>`, `<embed>`, `<form>`, `<base>`, and `<meta>`. Ghostrail parses these tags and extracts URLs found. However, not all HTML content is trustworthy. Web applications often allow users to provide own content, e.g. in the form of comments that are embedded in the HTML output. An attacker could easily abuse such a feature by posting the URLs he wants to request next. It is therefore possible and necessary to configure Ghostrail so that it excludes user-provided content from reference extraction within HTML documents. The second type of static content that may contain references is CSS data. As this is limited to only one syntax element (`url()`), an adequate regular expression performs the reference extraction.

There can be different URLs that are semantically identical, e.g. `http://example.org/?par1=foo&par2=bar` and `http://example.org/?par2=bar&par1=foo`. Ghostrail normalizes URLs in order to prevent misclassification.

During reference extraction, Ghostrail tags whitelisted URLs either as a *transition* or as an *extension*. Transitions make the browser replace the current web page while extensions only update a part of the page, e.g. in a iframe or by an AJAX request, or load an additional page in a new browser window (or tab) without modifying the parent window. A transition invalidates all previous whitelisted URLs whereas an extension adds new URLs to the whitelist.

3.3 Replication of Client-Side Execution

Beside the static references, dynamically generated requests play an important role within modern web applications. Applications that used to be installed and executed on a local machine, for instance office apps, move to the Web and become accessible via a web browser. The synchronization of client and server state as well as seamless interface updates require dynamic request generation and response processing, known as AJAX. The same is true for the search-as-you-type feature during user input. Such dynamically generated requests can not be determined using the static reference extraction described in Sec. 3.2 because the static analysis of JavaScript code is fault-prone and requires manual code annotations [8]. Ghostrail, however, aims to protect web applications without the need to change the application code. Instead, we equipped Ghostrail with a server-side replica of the user's browser to track the execution of JavaScript and derive the respective requests. Ghostrail maintains one replica for each user

session. Each replica runs in a sandbox and virtually performs the same actions that happen in the user's browser.

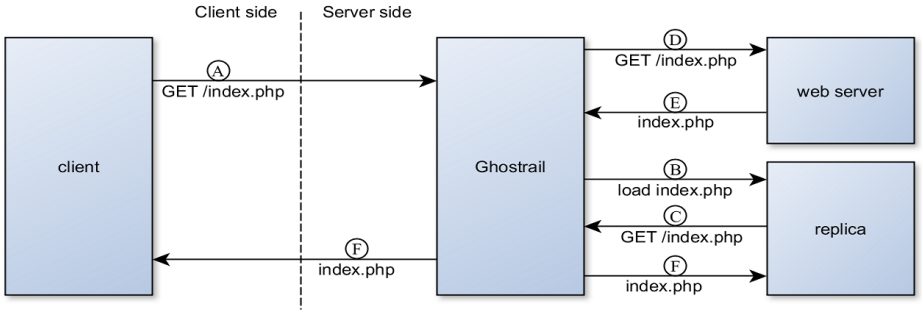

Fig. 1. Initial loading of a web page in the sandbox.

In order to monitor a user's actions, Ghostrail injects a few lines of JavaScript code into every delivered web page. This code monitors all user actions that can trigger JavaScript events. This is necessary because JavaScript has an event-driven execution paradigm: Code is not executed linearly but triggered by user actions, timing, or state changes of the web page. While timing and state changes also happen in the server-side replica without further ado, user actions must be transmitted to keep track. Interesting user actions include mouse movements, mouse clicks, and keystrokes.

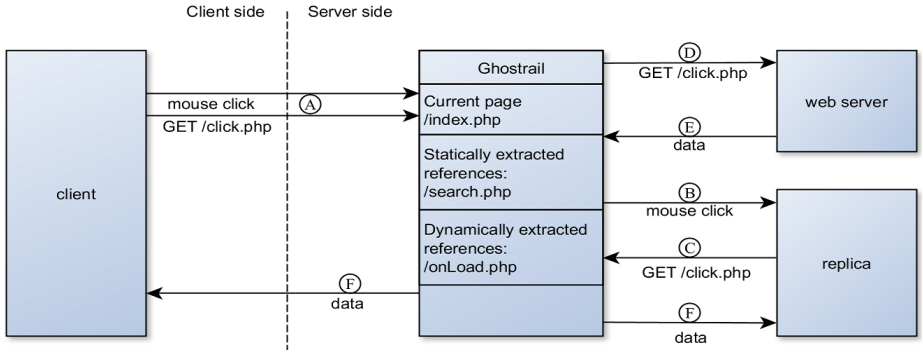

Fig. 2. Dynamic reference extraction by replicating an user's action.

The server-side replica virtually renders the same web page as the user, it executes the same JavaScript code, and it simulates the user's mouse movements, clicks, and form input, i.e., only expected – thus benign – user actions. The requests from the replica are the condensed set of expected user requests.

So, Ghostrail adds them to the user's whitelist. It is important to stress that Ghostrail only receives user actions from client side but not the respective state change in the user's browser. This is a crucial point because it limits a malicious user's scope: transmitting state changes allows an attacker to modify his browser such that it finally generates an attack request. For instance, a hash function may compute a URL parameter. The modified browser would always output `/etc/passwd`, independent of the input. Transmitting the output would allow the attacker to inject crafted requests into Ghostrail's whitelist. Limited to user actions, he can only spoof mouse clicks, movements and keystrokes on the web page. However, all these actions are within the scope of expected user interaction so there is no attack even if these actions are spoofed.

Fig. 1 shows the initial loading of a web page in the sandbox. The replica is initiated with the start of the user session (steps A/B). After the page has been fetched from the web application (steps D/E), it is send as a response to the client and the replica (step F). The replica does not interact directly with the web application to prevent double impact on the application state. Also, the duplication of the application's response (step F) ensures that both the client and the replica share the same content. After the page load, the user can interact with the web page. Fig. 2 shows an example how Ghostrail classifies dynamic references. The user clicks on an element (step A). The details are transmitted to Ghostrail and forwarded to the replica (step B) which simulates the same mouse click. The subsequent request to `click.php` (step C) is recorded by Ghostrail as legitimate because it is the result of intended user interaction. So, Ghostrail accepts and forwards the user request (step D).

4 Implementation

In this section, we describe the implementation of Ghostrail and the sandboxed replica. We implemented Ghostrail as a reverse proxy using Node.js (version 0.10.0) [9]. This reverse proxy manages all requests and responses and directs the replicas (see Sec. 3.3) as well as the static reference parser (see Sec. 3.2). This design allows to outsource CPU- or memory-intensive processes to other machines. The reverse proxy buffers incoming requests, queries the three-tiered list of regular requests (see Sec. 3.1) and finally accepts or rejects the request. It forwards the web application's responses to the static reference parser and the respective replica for further analysis. In the remainder of this section, we focus on the implementation of the replicas and the handling of client-side data.

4.1 The Sandboxed Replica

Ghostrail initiates a fresh replica for each user session – and destroys replicas when the session ends. We implemented the replicas using PhantomJS [10], a fully-fledged, GUI-less, and WebKit-based browser. Due to the WebKit basis, the replicas support all major web technologies like JavaScript, AJAX, CSS, JSON and SVG. Replicas do not need to run on the same machine as Ghostrail. They

can be distributed to cloud-based computing platforms to scale with varying load. The communication between the replicas and Ghostrail is based on HTTP and WebSockets.

Ghostrail injects a small piece of JavaScript code into every web page that is delivered to the user. This code establishes a WebSocket connection with Ghostrail to transmit user actions. Ghostrail records three kinds of user actions:

- **Mouse clicks:** Ghostrail records mouse clicks by injecting the `onclick` event handler for the whole web page. In order to simulate the click in the right page area, the (x, y) coordinates relative to the browser window are appended. Also, the dimensions of the browser window must be transmitted to configure the replica with the same size.
- **Mouse movements:** Mouse movements can trigger `onmouseover` events on a web page. Hence, Ghostrail records mouse movements above a configurable threshold using the `onmousemove` event handler. The threshold is necessary to avoid an overload of Ghostrail.
- **Key strokes:** Requests can contain user-defined data from an HTML form. Ghostrail must record every key stroke (using `onkeypress` and `onkeyup` event handlers) to simulate the user input. This is the only option to relate the resulting request to regular user behavior, i.e., entering data into form fields.

While Ghostrail injects new event handlers into the web page, there could be other event handlers that fire first and bypass Ghostrail's events. For instance, an event handler that redirects the browser to another page is executed first such that Ghostrail loses track and forbids future regular requests. We overcome this issue the following way: There are two options for the order of cascading event handlers, namely *event bubbling* and *event capturing* [11]. Event bubbling triggers the innermost event of the DOM tree first, i.e., for nested elements where each has an `onclick` event handler, the event of the inner element fires first when the user clicks. Event capturing has the reverse execution order. Ghostrail enforces event capturing and assigns its event handlers to the outermost element of the DOM tree, i.e., `document`.

4.2 Handling Browser Cache and History

In order to improve performance, browsers cache web content locally. Upon the next page access, they first query their local cache and restore the page without the need to request it again from the website. This, however, poses a problem to Ghostrail if it can not observe the local page load and extract the references from the cached page. We implemented a twofold cache management in Ghostrail to overcome this issue: First, Ghostrail adds the `Cache-Control: no-cache` HTTP header [12] to each response to prevent caching on client side. More precisely, the browser may cache the respective content but must revalidate every usage with the server. However, we found that the Chrome and the Firefox browser still cache at least the last visited page and reuse it without revalidation. Second, the client-side code detects the click that loads the cached resource. Then, the replica performs the same click and loads the same content from

the local cache provided by PhantomJS. Beside the local cache, browsers also maintain a local browsing history. This history allows users to navigate back and forth. PhantomJS supports the browsing history since version 1.8 such that the replica can emulate the navigation through the browsing history.

5 Evaluation

We evaluate the security gain by Ghostrail, investigate a possible impact of Ghostrail on the protected web application's availability, and give results of our performance measurements.

5.1 The Security Gain by Ghostrail

In order to evaluate the security gain by Ghostrail, we first describe how Ghostrail protects web applications against the attacks described in Sec. 2.2. Then, we explain our practical evaluation using the intentionally vulnerable web application Google Gruyere [13].

An attacker who exploits race conditions in web applications must send the same request many times in parallel. If the request is compiled dynamically, he must prepare the respective input in his browser and send the form using a mouse click to make the request be also sent by the replica and thus be whitelisted. In any case, as soon as Ghostrail accepts the attacker's first request, it immediately discards the whitelist of requests and waits for the application response to refill the list of expected requests. So, Ghostrail rejects the second request unless it is extracted from the subsequent page.

HTTP parameter manipulation attacks rely on the user's ability to freely change the parameters of a given request, e.g. to change the given account ID or message ID. While an attacker can still craft arbitrary requests in his browser's address line, Ghostrail rejects all requests that do not match an extracted request from the current page.

Unsolicited request sequences occur if an attacker can assemble a request to call application functions when they are not supposed to be called. In the example given in Sec. 2.2, the attacker knows the request that adds items to his shopping cart. As Ghostrail discards previously allowed requests, the crafted request is not whitelisted after checkout and thus rejected.

A forceful browsing attacker also needs to craft a targeted request that is not part of the current web page. So, Ghostrail rejects forceful browsing attempts by design.

In order to evaluate Ghostrail's protection in practice, we set up Google Gruyere that is vulnerable to forceful browsing, file inclusion, and reflected cross-site scripting (XSS) attacks. With Ghostrail in place, none of the attacks on Gruyere worked. However, we want to emphasize that injection attacks like XSS and SQL injection are out of scope for Ghostrail. If the attacker enters the payload into a form field, Ghostrail regards the resulting request as benign. So, protected applications still need to sanitize user input from free text form fields.

Nevertheless, Ghostrail limits possible user input via drop-down menus or radio buttons to the given options.

As Ghostrail plays the role of a traffic monitor, it must be the server-side endpoint of SSL connections with the client side. Given that it runs in the same domain as the protected web application, we do not consider this point a serious issue. If needed, the communication between Ghostrail and the web application, as well as between Ghostrail and the replicas, can be encrypted again.

5.2 The Impact of Ghostrail on the Availability of the Protected Web Application

We evaluated whether Ghostrail has a negative impact on the web application's availability. A negative side effect can occur if Ghostrail fails to extract a regular reference that is accessed by the user in the next step. In that case, Ghostrail mistakenly classifies the user request as unexpected (false negative). The evaluation is threefold: First, we set up a demo web application that implements a broad range of modern web technologies, i.e., redirects, CSS, jQuery as a representative JavaScript library, dynamic page updates via AJAX, and the navigation to dynamically generated URLs. This approach is meant to find out whether there are general compatibility issues of Ghostrail with any web technology. We used Selenium [14] to direct an instance of Firefox and Chromium respectively. Each browser performed virtually 1,000 user actions, resulting in 20,648 requests overall (each user action can trigger several requests). Afterwards, we analyzed Ghostrail's log files and did not find any blocked request. Please note that every blocked request would be a false negative because the virtual users only clicked on links on filled forms – what we defined as compliant behavior.

Second, we set up Ghostrail as a reverse proxy for the Alexa Top 20 websites. We used Selenium again to make Firefox perform 200 user actions on each website. Overall, we recorded 18,319 requests with a false rejection rate of 17.57%. Our analysis showed that blocked requests contained customized elements. Some websites perform a kind of client fingerprinting, i.e., they read browser and system features that differ for Firefox and the replica and add such information to the requests. The replica proved able to simulate the more common `User-Agent:` string for client classification. Another source for blocked requests are client-side timestamps and random numbers generated by JavaScript and appended to requests. In these cases, the outcome of code execution differs for the browser and the replica. While we had to consider each web application as a black box, the application provider can configure Ghostrail more appropriately to avoid most of the false rejections, e.g. by adding a rule that ignores differing parameters if they match the expected pattern and if their processing may not cause harm. We avoid transmitting the random numbers and timestamps from client side to the replica for synchronization because this would allow a malicious user to inject arbitrary HTTP parameters.

Third, we accessed three websites manually to learn the perceivable impact of Ghostrail. This is important because the raw number of blocked requests does not make a point concerning the impact on the web application's availability.

– *Google search:* The search function was usable without interference. Only the auto completion did not work due to differing request parameters.
– *Amazon:* We were able to search items, add them to our cart and checkout. However, we did not see product recommendations. The almost complete functionality of Amazon is particularly interesting because we experienced the highest number of falsely rejected requests ($\approx 60\%$). This result calls the significance of the raw number of false rejections into question.
– *Wikipedia:* We did not experience any issues on the availability.

We found that Ghostrail is able to allow workflows which span several domains. For instance, Ghostrail may protect an online shopping web application. When the user is redirected to a third-party cashier like PayPal, Ghostrail can not track the payment process (however, the cashier may run another instance of Ghostrail). Hence, the first request that leads the user back to the shopping application must be whitelisted as an application entry point. Instead of redirecting, a web application may include third-party content in its own pages. Then, the replica fetches the same content but does not provide cookies or authenticating HTTP parameters to avoid requests on behalf of the user.

5.3 Performance

Finally, we evaluated the performance impact of Ghostrail. We measured the HTTP round trip time between sending a request and receiving the response. We used the testbed with our demo application described above in order to avoid independent factors on the performance. The size of the served web pages ranges from 225KB to 450KB, and the round-trip overhead was between 250ms and 360ms. For applications with real-time requirements, e.g. online games, Ghostrail may only be an intermediate solution. For other applications, it is possible to scale the number of Ghostrail and replica instances with the load.

6 Related Work

The *Open Web Application Security Project (OWASP)* coined the term *Failure to Restrict URL Access* [15] to describe a similar vulnerability as our control-flow weakness. However, it is more focused on access control flaws. Workflows and control-flow integrity play a tangential role in the description.

With existing approaches, every change on the web application either needs a manual change on the policy definition [16, 17] or a new pre-release learning phase to derive the policy automatically [18–21]. Their policies can never be sound because training phases always miss unusual scenarios and manual policy definition requires expert knowledge and is prone to human faults. Also, all such approaches must be fuzzy by design because they neglect the actual request context, e.g. HTTP parameters like an ID that change case-by-case but must not be changed by the user. Ghostrail is able to enforce exact parameter matching without a need for policy updates.

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 statically analyze JavaScript to determine the expected sequence of requests [8] or check the web application for exploitable HTTP parameter pollution vulnerabilities [22]. Two approaches replicate client-side computation on server side to detect deviations: *NoTamper* [23] focuses on input validation of HTML forms, while *Ripley* [24] follows a similar approach to ours. It replicates client-side JavaScript events in a server-side replica. However, Ripley is only applicable during development but not for legacy applications. Also, it relies on a distributing compiler thus excludes non-fitting technologies. Technically, Ripley ignores mouse movements on client side and can not track respective events. As it uses event bubbling, it misses client-side events that redirect the browser. Ripley can not handle JavaScript code from different domains like common JavaScript libraries, mash-ups, and the `postMessage` API for communication between iframes. In that sense, Ghostrail is the consequent next step after Ripley because it covers modern application scenarios and all relevant user actions, thus makes less assumptions. Ripley and Ghostrail still share the same issues with randomness and timestamps.

An attacker exploiting a race condition vulnerability [7] can execute one function more often than intended by the application developer. Paleari et al. [2] describe an approach to detect race condition vulnerabilities in web applications.

7 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 attacker's possibility to craft arbitrary requests at any time together with the developer's assumption that users only follow provided links. Ghostrail overcomes this problem by the ad hoc generation of next step policies. It is the first approach that neither needs a repeated training phase nor a manual policy definition and covers the whole bandwidth of related vulnerabilities, including race conditions, HTTP parameter manipulation, unsolicited request sequences, and forceful browsing. Ghostrail is compatible with all modern web technologies including mash-up's and JavaScript libraries while it is applicable to all new and legacy web applications without any changes on the application code. For high-traffic applications, the induced load can be moved to any appropriate platform. In sum, we provided a thorough approach that provides guarantees to the developer concerning the sequences of incoming requests including the values of parameters. As a side effect, Ghostrail mitigates *Cross-Site Request Forgery (CSRF)* and *injection (XSS, SQLi)* attacks in most cases.

References

1. Braun, B., v. Pollak, C., Posegga, J.: A Survey on Control-Flow Integrity Means in Web Application Frameworks. In: NordSec 2013. (2013)

2. Paleari, R., Marrone, D., Bruschi, D., Monga, M.: On Race Vulnerabilities in Web Applications. In: DIMVA. (2008)
3. Grossman, J.: Seven Business Logic Flaws That Put Your Website At Risk. [White Paper], https://www.whitehatsec.com/assets/WP_bizlogic092407.pdf, last accessed 01/23/14
4. The New York Times: Thieves Found Citigroup Site an Easy Entry. [online], http://www.nytimes.com/2011/06/14/technology/14security.html, last accessed 01/23/14
5. Tate, R.: Apple's Worst Security Breach: 114,000 iPad Owners Exposed. [online], http://gawker.com/5559346/, last accessed 01/19/14
6. 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 S&P. (2011)
7. OWASP: Race Conditions. [online], https://www.owasp.org/index.php/Race_Conditions, last accessed 01/23/14
8. Guha, A., Krishnamurthi, S., Jim, T.: Using Static Analysis for Ajax Intrusion Detection. In: WWW. (2009)
9. Joyent, Inc: Node.js. [online], http://nodejs.org, last accessed 01/22/14
10. Hidayat, A.: PhantomJS. [online], http://phantomjs.org, last accessed 01/22/14
11. Ilya Kantor: JavaScript Tutorial - Bubbling and capturing. [online], http://javascript.info/tutorial/bubbling-and-capturing, last accessed 01/22/14
12. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: HTTP/1.1. RFC 2616
13. Google: Gruyere. [online], http://google-gruyere.appspot.com, last accessed 01/23/14
14. SeleniumHQ: Browser Automation. [online], http://docs.seleniumhq.org, last accessed 01/23/14
15. OWASP: Failure to Restrict URL Access. [online], https://www.owasp.org/index.php/Top_10_2010-A8-Failure_to_Restrict_URL_Access, last accessed 01/23/14
16. Jayaraman, K., Lewandowski, G., Talaga, P.G., Chapin, S.J.: Enforcing Request Integrity in Web Applications. In: DBSec. (2010)
17. Braun, B., Gemein, P., Reiser, H.P., Posegga, J.: Control-Flow Integrity in Web Applications. In: ESSoS 2013. (2013)
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. Balduzzi, M., Gimenez, C.T., Balzarotti, D., Kirda, E.: Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In: NDSS. (2011)
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. Vikram, K., Prateek, A., Livshits, B.: Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution. In: CCS. (2009)