Guarda: A web application firewall for WebAuthn transaction authentication

by

Damian Barabonkov

S.B., Computer Science and Engineering, M.I.T. (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by .....
Anish Athalye
Doctoral Candidate
Thesis Supervisor

Certified by .....
M. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Guarda: A web application firewall for WebAuthn transaction authentication

by

Damian Barabonkov

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Transaction authentication is an attractive extension to two-factor authentication. It is proposed in the WebAuthn standard by the World-Wide-Web Consortium (W3C) as a mechanism to secure individual "high-risk" operations of a website via a hardware authenticator device. It defends against a stringent threat model where an adversary can modify or create HTTP requests between the user and the web service. Transaction authentication as defined by WebAuthn is not yet adopted in practice, partially because it requires intrusive web application changes.

This thesis presents Guarda, a firewall for integrating transaction authentication into a new or existing web service with relatively few code changes. The firewall intercepts all HTTP traffic sent to the web service, and based on the configuration, any requests deemed safe are proxied directly to the web service. All other requests are considered high-risk and are held back and validated using transaction authentication. Only if the validation passes are they also permitted to pass through to the web service.

This thesis uses the firewall approach to integrate transaction authentication into three web applications: a blogging site named Conduit, a WordPress admin panel named Calypso and a self-hosted Git service named Gogs. Compared to directly modifying them to support transaction authentication, the firewall approach is close to 8 times more concise. Under heavy load, there is an associated latency of at worst 1.5x slower when using Guarda to secure Gogs versus accessing the web service directly without WebAuthn.

Thesis Supervisor: Anish Athalye
Title: Doctoral Candidate

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor

Acknowledgments

Anish, for his thoughtful expertise and inexhaustible willingness to help when I was stuck.

Frans, for his insightful advice and guidance, which helped steer the direction of this thesis.

MIT, for shaping me into the computer scientist I am today, for presenting me with doors and opportunities I could never have imaged ever existed.

My dearest of friends, for being an integral part of my life, for being a source of limitless happiness and memories that will last a lifetime.

My parents, for their unabating love, for their unconditional support in every endeavor of mine and for being the best parents I could ever wish for. I love you.

Contents

List of Figures

List of Tables

Code Snippets

Chapter 1

Introduction

This thesis presents Guarda, an application-level firewall to add transaction authentication to an existing web application. This chapter describes two core concepts, the threat model and transaction authentication, which are essential to understand before introducing Guarda and the problems it solves. This chapter paints a broad picture of the overall thesis. The sections here serve as a preludes for the chapters that follow in the thesis body.

1.1 Current Authentication Methods

Websites which have user accounts traditionally use a password as their main form of authenticating the user. The security assumption is that if an adversary does not have the user's password, the user is safe from any adversarial attacks. Compromising a password is common [9] because it is easy to target and attack remotely and at scale. Security researchers have been well-aware of the gaping security risks of only using password authentication and thus have developed a security system called two-factor authentication [16]. In one form, upon account registration, the user not only picks a password, but also provides a secondary method for login authentication, usually through a hardware device. Then to login, the user must supply the password as well as physically authenticate the login on their hardware authenticator. However, this commonly used mechanism does not offer protection in a stricter threat model where

an adversary has control over a user's web-browser or operating system.

The World-Wide-Web Consortium (W3C) proposed a specification called *WebAuthn* [8]. It supports extensions to the traditional two-factor authentication. One extension is a mechanism called *transaction authentication*, which provides a number of security guarantees within this new and broader threat model.

1.2 Threat Model

WebAuthn transaction authentication protects against a more stringent threat model than traditional two-factor authentication. In transaction authentication, the threat model assumes that all components except the web service code and hardware authenticator device are vulnerable. This broad capability gives an adversary the ability to create or modify any HTTP requests between the user and web service. Save from defending against denial-of-service, transaction authentication prevents broad unauthorized access to a user's account by the adversary. There is an exception to the threat model: the entire registration event, where cryptographic public key information is sent from the user to the web service, must be assumed secure in order to provide its prescribed security guarantees.

In contrast, the threat model for traditional two-factor authentication assumes a more limited adversary. This adversary has the capacity to launch phishing attacks and steal passwords, but the operating system and web-browser are assumed secure. If any of them were to be compromised, two-factor authentication fails to ensure its security guarantees. For example, a compromised user web-browser can wait for the user to authenticate and login faithfully and then take control of the account. Transaction authentication protects against such failure modes.

1.3 Transaction Authentication

The purpose of transaction authentication is to require "high-risk" operations to be individually authenticated by the user's hardware authenticator, despite the user

being already logged in.

A number of components are at play during transaction authentication. From the user's perspective, they must posses a hardware authenticator device with a text display on which they attest to operations they want to perform. On the other end is the verification code that guarantees that the user's attestations are correct and valid. This code typically runs on the backend, but in the case of this research, it is a part of Guarda.

So when the user tries to issue some high-risk operation, e.g., a monetary transfer, a confirmation message on their hardware authenticator will appear. The message is specific to the operation the user is trying to perform and should contain enough information for the user to validate that it is indeed correct and as intended.

For example, a bank website could require that monetary transfers exceeding $500 must be transaction authenticated. In such a case, a possible authentication message that the user would have to confirm could resemble "Send Alice $750 from account #12345". The user would view the message on their hardware authenticator device as illustrated in Figure 1-1.

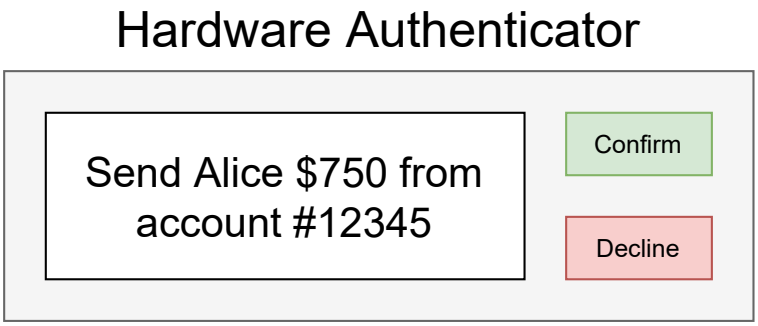

Figure 1-1: User interface of hardware authenticator.

This message contains enough information such that the user is fully informed of the operation they are about to perform. They can then make the educated judgment for whether to confirm the operation or not. In the adversarial event where a modified or unsolicited monetary transfer is attempted, the user should notice a discrepancy in the authentication message and decline the operation on the hardware authenticator.

The goal of transaction authentication is to prevent the adversary from launching

unsolicited operations on behalf of the user and causing fraudulence or damage. The operations that matter will require this additional authentication which the adversary cannot falsify as per the threat model — the hardware device is assumed secure. This is a safe assumption because the hardware device is specialized to only perform this authentication process and nothing else. Ideally, it has a small attack surface that is vetted by security specialists.

When the user authorizes this high-risk operation on their hardware authenticator, the message is cryptographically signed by the authenticator and sent back to the verifying end where it is checked. Only then, upon successful verification is the operation performed.

Which operations should be protected is entirely at the discretion of the administrator of the web service. There is no clear-cut formula on what to protect, but candidate high-risk operations could include, but are not limited to, deleting one's account, transferring money, managing administrative permissions, publishing important software releases, etc.

1.4 Status Quo of Transaction Authentication

Although there are web services and hardware authenticators that support WebAuthn two-factor authentication, no service or authenticator supports the transaction authentication extension. Apart from transaction authentication not being integrated into real-world applications, there also has been no investigation into the implications of where it fits well, where it is inhibitory or unnecessary, what it takes to support transaction authentication in a web service and any software system designs that would help with integrating and configuring WebAuthn for an existing service.

There are several publicly available codebases in the form of demo applets and libraries that work with WebAuthn transaction authentication [18] which illustrate how it can be used in a website. A website typically consists of a frontend and a backend. The frontend is the HTML/CSS/JavaScript that runs in the user's web-browser and is the origin of all of the user's requests when interacting with the website.

The backend consists of the server code and database that execute the user's requests from the frontend.

A web service using WebAuthn transaction authentication would have the frontend issue the WebAuthn requests and have the backend contain the WebAuthn library code to verify those requests and permit the operation if validation passes. Such approach to integrating WebAuthn has a number of practical downsides.

Firstly, the backend architecture might not lend itself well to the control flow of the WebAuthn specification. In order to validate a WebAuthn request, multiple HTTP requests need to be sent back and forth between the frontend and backend. This may not be handled well by the backend if it was built under the assumption that there is only one HTTP request per operation.

Secondly, the integration of WebAuthn transaction authentication into a backend web service tends to be challenging for larger web services. The handler code for each operation is spread throughout the codebase, so it is difficult to keep track of what is secured and where in the code it is. Also, it requires a strong overall understanding of the web service code, which comes with its own set of challenges and slow development speed.

Lastly, in the extreme, albeit unlikely case, the backend may not even be accessible to the software developer if it is, for example, a closed-source API backend. In this case it is impossible to integrate WebAuthn into such a backend.

1.5 Thesis Contributions

There are three major contributions of this thesis:

1. Guarda: A firewall system design for integrating WebAuthn transaction authentication into a new or existing web service.
2. Three distinct case studies showcasing Guarda. Each case study observes a class of web service and is used to test the limits of the usability of Guarda.
3. A discussion on the beneficial use cases for transaction authentication, when it makes sense to be applied and when not.

1.5.1 WebAuthn Firewall

Firstly, it must be emphasized that WebAuthn is simply a protocol specification. Implementation details are not bound to any one way, as long as they comply with the protocol's details. However, the code demos of WebAuthn unanimously integrate it into their codebases in the same intrusive manner. They import a WebAuthn library [15] directly into the codebase and perform the necessary checks and validation in the backend.

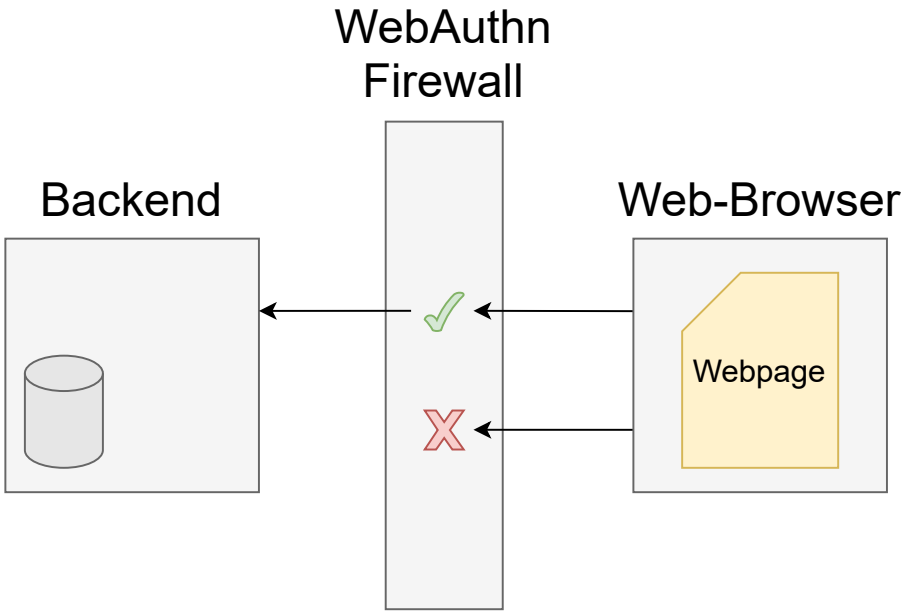

Figure 1-2: Basic depiction of the WebAuthn firewall functionality.

This thesis proposes an alternative method for integrating WebAuthn by incorporating Guarda, a Web Application Firewall (WAF). This firewall monitors and filters HTTP traffic sent from the website webpage to the backend. Any requests that the firewall deems as needing WebAuthn transaction authentication are stopped and processed by it. The firewall validates the request according to the WebAuthn transaction authentication specification. Figure 1-2 depicts how if the validation succeeds, then the firewall lets the request pass on through to the backend. If not, then the request is blocked. As far as the backend is concerned, it is unaware that Guarda exists between it and the website webpage.

With the WebAuthn firewall approach, the backend has to be minimally, if at all,

modified to support WebAuthn transaction authentication. Although the frontend still needs to be modified to produce the WebAuthn transaction authentication requests as it is the origin of all of the user's operations, the firewall approach lends itself better to integrating WebAuthn into a new or existing web service. It is less intrusive than the traditional library-based approach and consolidates all of the WebAuthn related code in one place. As a result, it is less error-prone and easier to configure.

The design of Guarda presented in this thesis goes beyond simply providing a description and base functionality. It supplies the software engineer with useful defaults and a domain specific language (DSL) to configure Guarda. Namely with a short configuration, the engineer can adapt Guarda to parse and understand whatever input request format the web service uses. Requests map to operations in a web service depending on which HTTP route they are sent to. With only a few lines of code per route in the Guarda configuration, the engineer can specify that the route be checked with transaction authentication and what the authentication message should be in order to pass as valid for that operation.

1.5.2 Case Studies

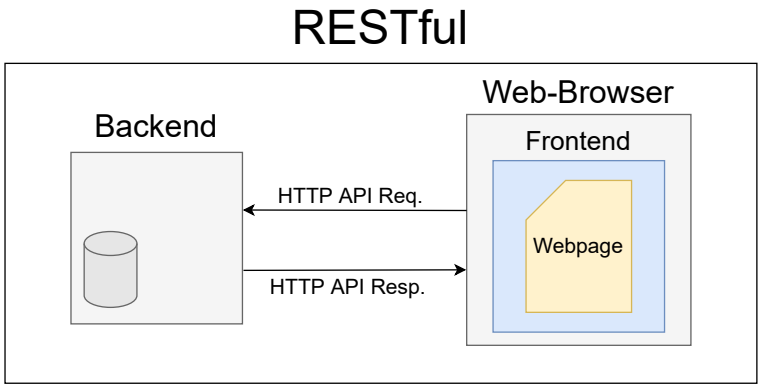

Figure 1-3: The architecture design of a RESTful web application.

Three separate case studies demonstrate how Guarda secures different architectures of web applications with WebAuthn transaction authentication. The web applications studied cover the RESTful design paradigm and the more traditional server-side rendering paradigm. The two RESTful applications studied are Conduit [5], a

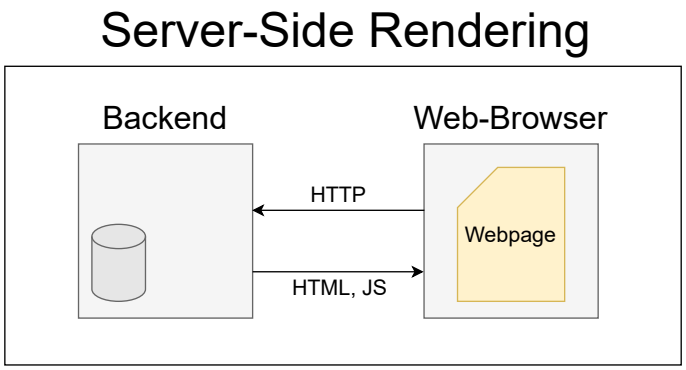

Figure 1-4: The architecture design of a server-side rendering web application.

simple blogging website, and Calypso [7], a frontend admin panel for WordPress. As illustrated in Figure 1-3, the RESTful paradigm is where the frontend runs in the web-browser separately from the web server backend program. The frontend performs the rendering visible to the user, and whenever it needs to fetch information or has to perform some server operation, it communicates via a pre-established API to the web server, which executes those requests.

The server-side rendering application studied is Gogs [2], a self-hosted Git service much like GitHub [1]. Figure 1-4 depicts the server-side rendering paradigm, where every operation the user performs on the webpage is sent over to the server as an HTTP request. In response, the server returns a whole new webpage, complete with all of the HTML and JavaScript code to be displayed on the user's web-browser.

These case studies stress test the flexibility and configurability of Guarda. They are critical in establishing what parameters and configurable knobs Guarda should provide. They are also used to evaluate how tangible the improvements are for using Guarda in a web service rather than integrating WebAuthn transaction authentication intrusively.

1.5.3 Other WebAuthn Possibilities

In this thesis, Chapter 8 gives insights into best practices relating to utilizing WebAuthn transaction authentication to secure various classes of problems. Researching transaction authentication reveals clear use cases where it lends itself well to secure,

some use cases which are possible to secure, but not optimal, and lastly some use cases which cannot be protected at all by transaction authentication.

1.5.4 Source Code

The source code of Guarda and case studies is publicly available at `https://github.com/JSmith-BitFlipper/Guarda-firewall` under the MIT License.

1.6 Thesis Outline

The thesis contains the following chapters. Chapter 2 discusses the related work and background pertinent to this research. Chapter 3 describes the WebAuthn transaction authentication protocol in detail. Chapter 4 outlines the design of Guarda, the WebAuthn firewall. Chapter 5 discusses the implementation details of the novel components of Guarda. Chapter 6 explains the case studies of Guarda. Chapter 7 contains the experiments and evaluation metrics for Guarda. Chapter 8 opens a discussion of supplemental notes uncovered over the course of this research. Finally, Chapter 9 concludes this thesis.

Chapter 2

Related Work

This chapter presents the background and related work that precedes this research. These concepts lay out the foundation on which the rest of this research is conducted.

2.1 Hardware Authenticators for Two-Factor Authentication

The state of the art for using hardware devices for website authentication is two-factor authentication. These devices such as the YubiKey [21] resemble a small USB key which stores a private key on device. Websites that support two-factor authentication simply request the secondary mode of authentication during login. Two-factor authentication solves the security problems for login quite well [9]. Depending on the implementation of the specific hardware device, most provide strong resilience to targeted impersonation, physical observation, internal observation, leakage of data secrets and relying on a trusted third-party. However, as explained previously, these benefits do not pertain beyond the login point of a website and assume a lenient threat model. The adversary with control over the web-browser could simply wait for the user to faithfully log in and then launch their planned attack.

A number of specifications standardize two-factor authentication so that the same hardware authenticator device may be used across platforms and web services. A

popular standard is Universal 2nd Factor (U2F) [20] developed by Google and Yubico, now hosted by the open-authentication industry consortium FIDO ("Fast IDentity Online") Alliance. It is succeeded by FIDO2 [6], which merges WebAuthn and its extensions, including transaction authentication, into one common standard.

2.2 Current Uses of Transaction Authentication

Although some web services like GitHub support WebAuthn two-factor authentication [11], transaction authentication as per the WebAuthn standard is not yet deployed commonly. Cryptocurrency hardware wallets universally support transaction authentication, but for narrow use cases involving crypto transactions. The two most popular manufacturers, Ledger [17] and Trezor [19], sell hardware wallets which hold the private keys and have displays. In order to send any cryptocurrency from the device, the device displays a message detailing the transaction, which the user would need to authorize on the physical device. Otherwise, the device will not sign the transaction which is required for it to proceed and be sent over the network.

In a similar but diminished vain, some banks such as Bank of America recycle two-factor authentication for sensitive or high-value monetary transactions [12]. In order to complete the sensitive transaction, the user must redo two-factor authentication like during login. This is unlike transaction authentication in that no hardware device specific to transaction authentication is involved, and the user does not see a confirmation of the exact transaction about to take place before confirming. But the process of requesting supplemental two-factor authentication on sensitive transactions aims to defend against a similar threat model as transaction authentication.

2.3 Hardware Authenticators for Transaction Authentication

Hardware authenticator devices normally do not support general-purpose transaction authentication. Hardware authenticators for two-factor authentication such as the

YubiKey discussed in Section 2.1 do not have any display. Therefore, they are unable to perform the core function of transaction authentication, displaying a message to the user and signing it upon their confirmation. The cryptocurrency hardware wallets discussed in Section 2.2 provide transaction authentication, but only for a limited subset of functionality relating to crypto transactions.

WebAuthn transaction authentication is not supported by any hardware authenticators. There are hardware authenticators and software emulated authenticators which support WebAuthn, but only for regular two-factor authentication. The software emulated authenticators are intended for development and research purposes. Google has a Chrome extension that poses as a virtual hardware authenticator; it performs all of the signing in-browser [13]. Krypton is a small company dedicated to making cryptographic authentication easy [14]. They provide a browser plugin and phone app, which pair with one another. The plugin interfaces with the web-browser and the phone, which performs the cryptographic signing.

This research must modify one of the software emulated authenticators to support transaction authentication. It uses a modified Krypton browser plugin to display the authentication message and await user consent before performing the signing in the plugin. Figure 2-1 portrays the user interface of the plugin as it awaits consent before returning a signed transaction authentication object. This design deviates from the original Krypton design as it eliminates the phone app entirely and signs in-browser. The security properties of the authenticator device itself is not a focus for this research. It is assumed always secure, so for prototyping and experimentation, this software modified authenticator is sufficient.

2.4 Web Application Firewalls

A Web Application Firewall is a firewall that filters web traffic going to a web service [10]. Most web application firewalls try to block malicious internet traffic from ever reaching the web server. Common attacks defended against by web application firewalls include SQL injection, cross-site scripting and DDoS attacks. Typically, the

firewall inspects incoming GET and POST request HTTP/HTTPS traffic and applies pre-configured rules to identify and filter out the undesired traffic. This thesis describes how to make a web application firewall supporting WebAuthn transaction authentication.

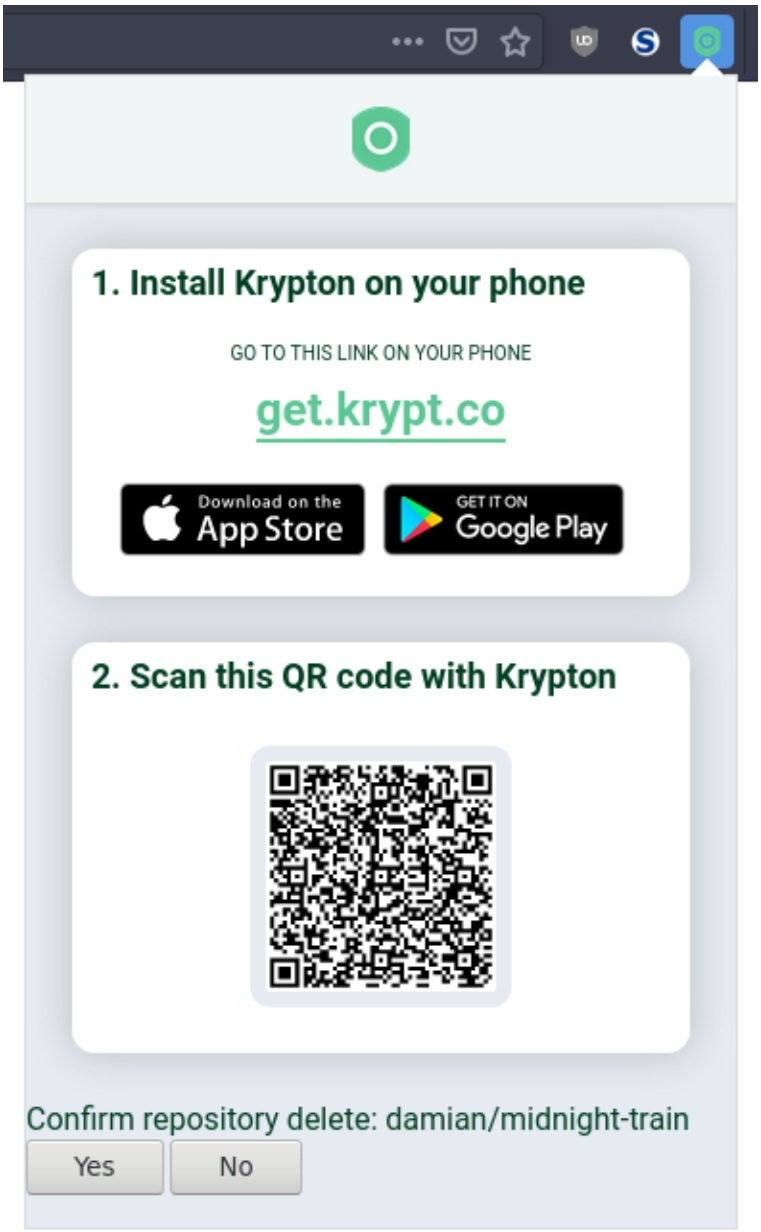

Figure 2-1: User interface of Krypton authenticator in this thesis. It is awaiting user consent before authorizing the deletion of a repository "damian/midnight-train".

Chapter 3

WebAuthn Transaction Authentication

WebAuthn transaction authentication is a protocol specification for authenticating high-risk user operations after login. The specification describes a sequence of steps that must be followed in order to authenticate properly. There is registration and then transaction authentication, which can be split into three stages: the setup, the cryptographic attestation and then the verification. Figure 3-1 gives a simplified overview of the transaction authentication process. It visualizes the order of communication among the hardware authenticator, the web-browser, the WebAuthn verification end and backend server.

1. Registration: Makes a record of the user and their cryptographic credential into a database. Registration is performed only once per user and is assumed secure. The database record is used by the verification step later on.
2. Setup: Initiated by the user's web-browser and involves a priming exchange between it and the WebAuthn verification end.
3. Cryptographic Attestation: Occurs on the hardware authenticator device after the user confirms the operation. The threat model assumes that this attestation is secure.
4. Verification: Validates whether to authorize the high-risk user operation or not. Checks if the requested operation matches its authentication message and that

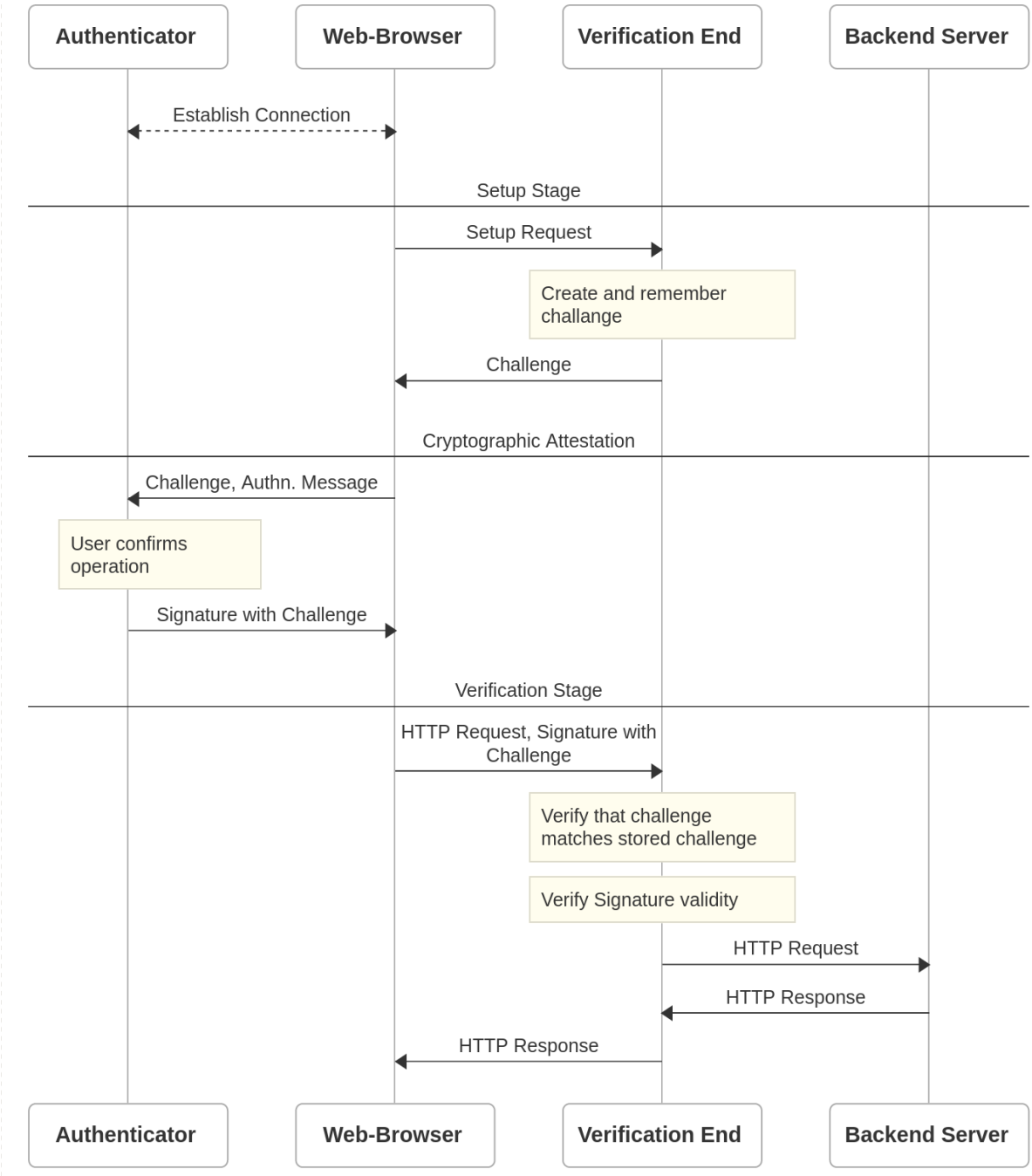

Figure 3-1: A simplified overview of the flow of events during WebAuthn transaction authentication, subdivided into three stages.

the cryptographic signature on it is valid. This stage also is assumed secure under the threat model.

The rest of this chapter describes these steps in more detail and uses Guarda, the WebAuthn firewall, as the verification end. It contains the database of user public key credentials and performs the validation to authorize high-risk operations.

3.1 WebAuthn Registration

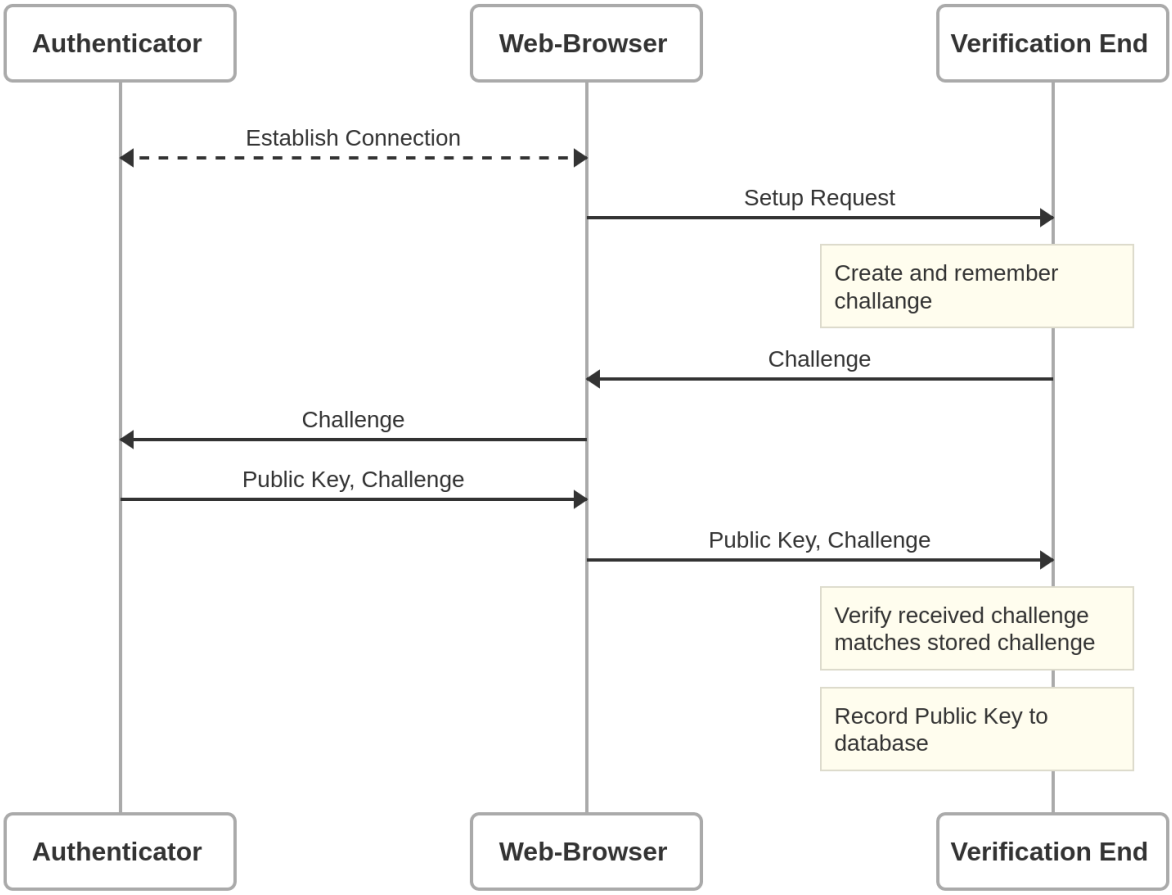

Figure 3-2: The flow of events during WebAuthn registration.

The purpose of registration is for the user's hardware device to transfer its public key credential to Guarda. This process is assumed to be secure in the threat model, with no adversary able to intercept or tamper with any of the communications. Therefore, whatever credential Guarda receives during registration is assumed

to be genuine and the user's own. The firewall later on uses this credential during the verification stage to ensure transaction authentication integrity.

Figure 3-2 illustrates the flow of events during the registration event. The registration process begins with a setup of its own, where the web-browser requests a few parameters from the firewall, most notably a random challenge nonce. Guarda remembers the challenge it sent as a part of the session data associated with the registration setup request. The challenge nonce prevents replay attacks. There are a few other parameters, but they are mainly for the hardware device to know what type of credential the firewall is expecting to receive. The hardware device sends over its public key credential for Guarda to save, with the challenge signed by that credential. Guarda receives an HTTP POST request containing this public key credential. The POST request also contains identifying information of the current user. Guarda verifies that the challenge matches, and upon success, stores the credential and associated user ID into a database row.

3.2 Transaction Authentication Setup

The setup for a transaction authentication event originates from the frontend webpage on the web-browser. There are several ways to setup a WebAuthn transaction event, but the contents being exchanged are all the same. More commonly, setup occurs lazily where the frontend waits for the user to initiate an operation protected by transaction authentication before initiating the setup. Or it may occur eagerly where the frontend preemptively initiates the setup, without knowing whether the user will even perform any secured operation on the webpage.

Figure 3-3 illustrate the setup stage of a transaction authentication event. The setup begins with a POST request sent to Guarda. The payload for the setup POST request is an authentication message that will eventually be displayed to the user on their hardware device. The message is constructed from user input and additional information contained in the HTML of the webpage, but in all three case studies of this thesis, that was always sufficient. In response to the POST request, the frontend

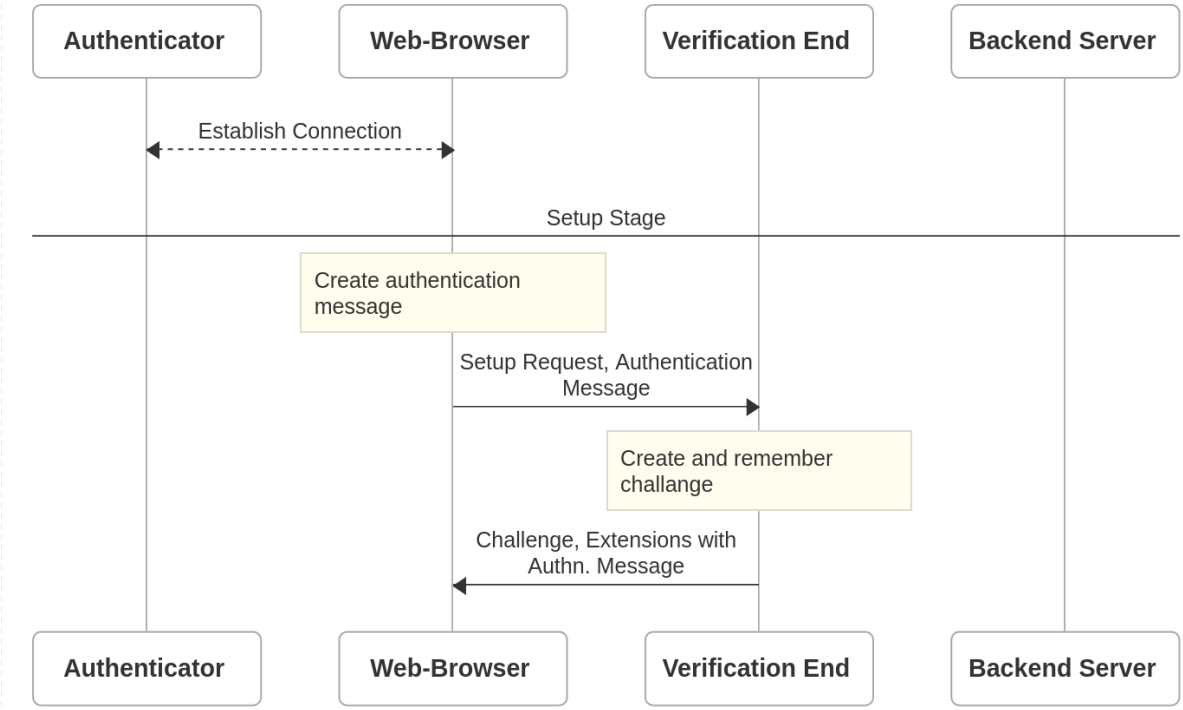

Figure 3-3: The flow of events during the setup stage of WebAuthn transaction authentication.

receives a few parameters:

1. A random `challenge` nonce: The firewall remembers it locally in the session data associated with the request. When the firewall processes the protected request, it will verify that the challenge included in the returned authentication data matches the one previously sent and remembered in the session. An adversary cannot intercept and replay old protected requests since it is exceedingly unlikely that future challenges from the firewall will exactly match the challenge in the intercepted request.
2. An `extensions` field: The firewall transforms the authentication message sent to it into a WebAuthn-compatible form for the hardware authenticator. It then places it into the `extensions` field, which the authenticator reads and handles as a transaction authentication event.

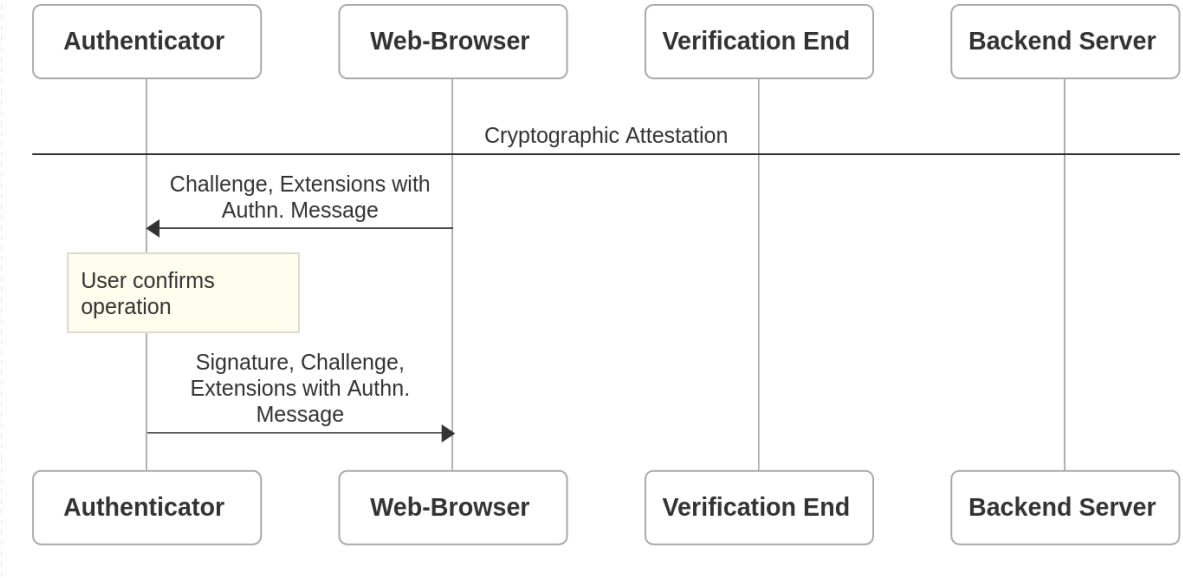

Figure 3-4: The flow of events during the cryptographic attestation stage of WebAuthn transaction authentication.

3.3 Cryptographic Attestation

The request options from Guarda go through the frontend and are passed on to the hardware authenticator device. The threat model assumes that only the firewall, backend and hardware authenticator are secure. At any point, the frontend or web-browser could modify these options, but any tampering will be detected later on and denied authorization.

Figure 3-4 outlines the role of the hardware authenticator. The hardware device parses the request options, extracts from the `extensions` field the authentication message and presents that to the user. The authentication message is in the form of a confirmation for some requested operation and is answered either by "yes" or "no". If the user attests "yes", the hardware device cryptographically signs a data object, which is returned as an additional field within the HTTP request to Guarda for verification.

The response of the hardware authenticator includes a `clientDataJSON` object containing the authentication message displayed to the user a well as the `Challenge` from the setup stage. A cryptographic signature of the `clientDataJSON` is also included. The signature is computed using Elliptic Curve Digital Signature Algorithm (ECDSA)

paired with the SHA-256 hash function. There are other fields, as well, for plumbing to help Guarda know what parameters to use to validate this response.

3.4 WebAuthn Firewall Verification

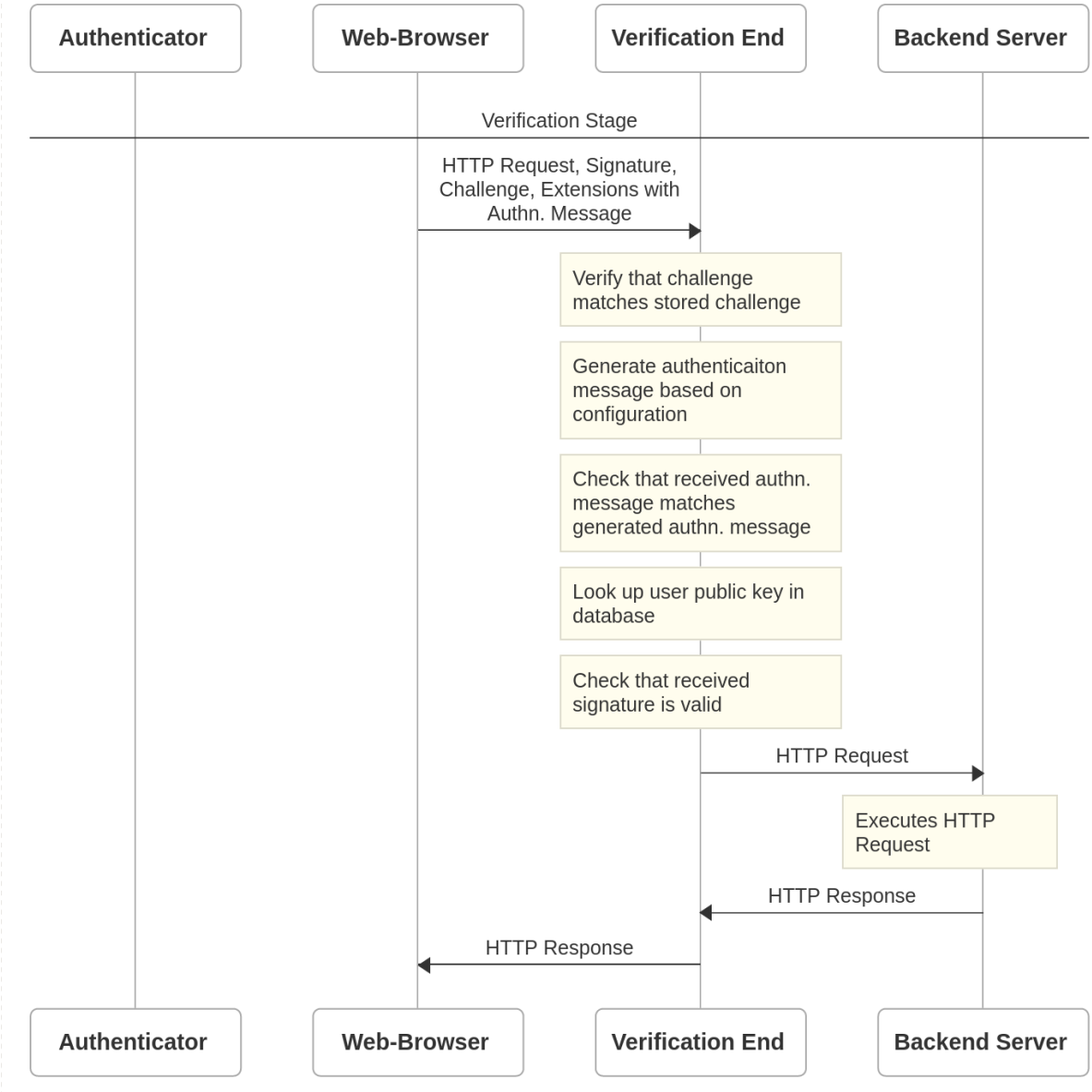

Figure 3-5: The flow of events during the verification stage of WebAuthn transaction authentication.

Guarda receives an HTTP request on a protected route with all of its usual parameters plus the authentication data object. The firewall must verify the integrity of

this object as well that it corresponds with the intent of the HTTP request. In other words, it must detect whether any code not in the trusted computing base, such as the frontend, tampered with the authentication data.

Figure 3-5 illustrates the main steps of the verification stage. The three main steps are to verify the challenge, the authentication message and the authentication data signature:

1. Checking the challenge is a simple comparison between the `challenge` received and the `storedChallenge` in the firewall's session data. This protects against replay attacks.
2. Checking the authentication message is more involved. The firewall is configured per route how to generate an expected authentication message based on the HTTP request parameters. This generated authentication message must unambiguously encapsulate the entire intent of the request. Details are further discussed in Section 4.4. Then it is a simple comparison between the received `clientMessage` and the `generatedMessage`. This makes sure the user authenticated a message that faithfully represents the intent of the HTTP request.
3. Checking the authenticating data signature involves invoking cryptography library utilities. This validates the integrity of the entire authentication object to prove that it was not tampered with. The `clientDataJSON` is signed by the hardware authenticator. The firewall has the public key of the hardware authenticator, so it can see if the `clientDataJSON` indeed corresponds to the signature attributed to it.

Chapter 4

WebAuthn Firewall Design

This chapter describes the high-level design and purpose of Guarda, the WebAuthn firewall, and how it fits within an existing web application. It also describes how a software engineer would use the different configuration options and tools that Guarda provides in order to secure a web service.

4.1 Overview

Guarda acts as a Web Application Firewall. It is situated directly between the frontend and backend, processing all user requests sent between the two. At the highest-level, the purpose of Guarda is to map HTTP requests on protected routes to authentication messages and then verify those requests. To make the mapping and verification easier, Guarda provides configurable options, a domain specific language and a collection of default functions.

4.1.1 Request Life Cycle

The flowchart in Figure 4-1 illustrates the life cycle of an HTTP request passing through the WebAuthn firewall. It begins with Guarda capturing and parsing the request when it is sent from the frontend to the backend. Then Guarda decides whether to verify the request with transaction authentication or not. If not, the

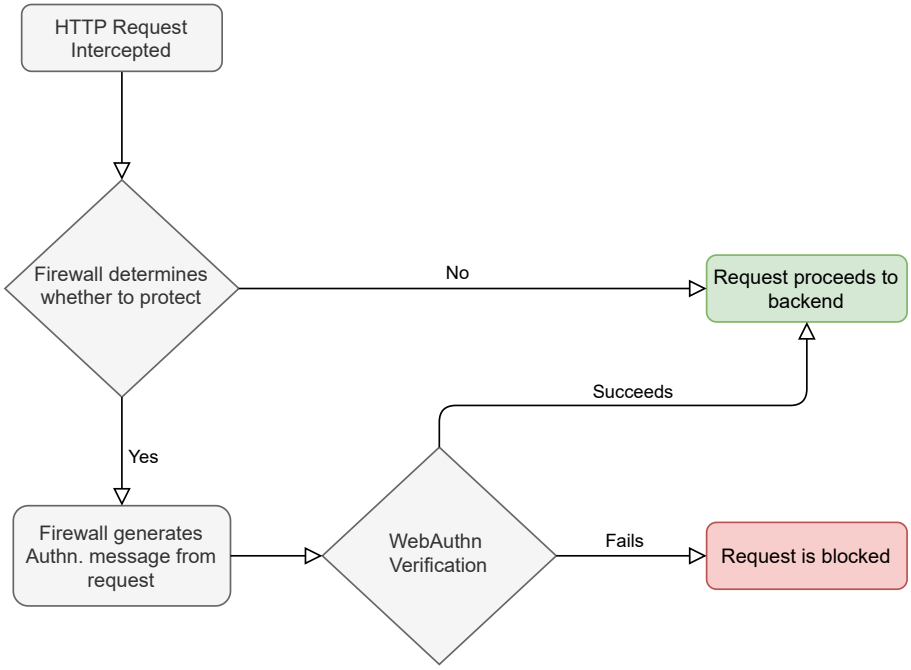

Figure 4-1: The decision process for authorizing an HTTP request or not.

request is simply proxied through to the backend without any extra work. Otherwise, Guarda performs a verification procedure on the request and only upon success does the request pass through the firewall to the backend. Upon failure, the request is blocked.

4.1.2 Securing Sample Operation

To illustrate how an operation is secured, consider the example of an HTTP request to delete an SSH key within Gogs named `"Damian's SSH Key"`. Code Snippet 4.1 is the HTTP request and relevant parts of its payload.

```
1 route: "user/settings/ssh/delete",
2 form-data: {
3   id: 6,
4   assertion: "<assertion data from hardware authenticator>"
5 }
```

Code Snippet 4.1: A sample HTTP request to delete the SSH key with ID 6.

This request should map to a human-readable authentication message, in particular: `"Delete SSH key named: Damian's SSH Key"`. Notice that the HTTP request

contains the ID of the SSH key, but not the name itself. The name is not present anywhere in the request, but can be contextualized from the ID by the backend as described in Section 4.3.3. The domain specific language and default functions make this mapping and context retrieval easier. Code Snippet 4.2 is roughly what manually securing this route in Go would look like.

```
1  func deleteSSHKey(w http.ResponseWriter, r *http.Request) {
2    id := int(r.Form["id"])
3    assertion := r.Form["assertion"]
4
5    // Retrieve the SSH key name of `id`
6    var sshKeyInfo struct {
7      KeyName string `json:"keyname"`
8    }
9    PerformRequestJSON(fmt.Sprintf("server_context/ssh_key/%d", id),
10                      &sshKeyInfo)
11
12   authText := fmt.Sprintf("Delete SSH key named: %s",
13                           sshKeyInfo.KeyName)
14
15   // Check the `assertion` against the `authText`
16   FinishLogin(assertion, authText)
17 }
18
19 router.HandleFunc("/user/settings/ssh/delete",
20                    deleteSSHKey).Methods("POST")
```

Code Snippet 4.2: Route handler which manually secures the delete SSH key operation of Gogs.

First, the code reads the `"id"` and `"assertion"` values from the request form-data. It then contextualizes the `"id"` to retrieve the name of the associated SSH key via the backend's context route. Finally, it creates an authentication message and verifies it against the `"assertion"` data. This handler is attached to the HTTP route accepting SSH key deletion requests.

The domain specific language and default functions simplify this code. The HTTP route for SSH key deletion, `"/user/settings/ssh/delete"`, and request type, `"POST"`, are specified in one function call. A short domain specific program specifies how to generate the authentication message. It begins with a format string `"Delete SSH key named: %v"` where the format tag `"%v"` gets substituted with the SSH key name. Code Snippet 4.3 is the firewall code that secures the SSH key deletion

for Gogs.

```
Secure("POST", "/user/settings/ssh/delete",
  Authn("Delete SSH key named: %v",
    wf.SetContextVar("ssh_key", wf.Get("id")),
    wf.GetVar("ssh_key").SubField("Name"),
))
```

Code Snippet 4.3: Gogs firewall code which incorporates the domain specific language to secures the delete SSH key operation.

As demonstrated above, the direct method of integrating WebAuthn into a web service is bulky and difficult to configure. The supplemental tools provided by Guarda allow an engineer to write less code to achieve the same functionality, thus making development easier and less error-prone.

4.1.3 Configurable Components

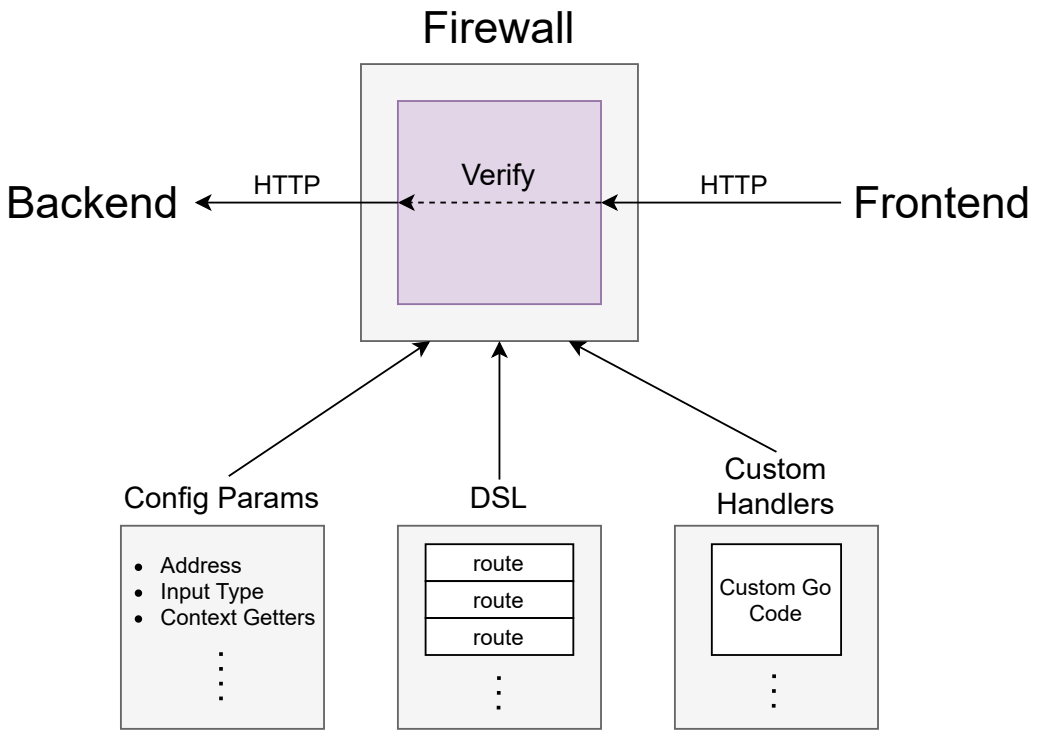

Figure 4-2: The functionality of the WebAuthn firewall fully depends on the configurable components passed into it.

Two sources of configuration enable the Guarda's customizable nature. One source

is the central configuration element depicted as the leftmost input block in Figure 4-2. It contains the general configurable parameters, from which a number of core WebAuthn routes are deduced and secured transparently.

The other two input blocks of Figure 4-2 handle the rest of the application specific routes to be secured. They are secured either using the domain specific language or custom Go code. These configurable options can be easily modified to completely define how Guarda protects a web service.

Apart from configuration ease, the design of a WebAuthn firewall is powerful because it is almost transparent to the web service it is securing. The backend is unaware that the requests it receives are WebAuthn authenticated. The frontend has to interface with the user's hardware authenticator device, so it must be aware of WebAuthn, but only to a minor extent. As a result, deployment of Guarda is simple and seamless.

4.2 Proxying Requests

In three different case studies, Guarda is used to integrate WebAuthn transaction authentication into two different paradigms of web service designs, RESTful and server-side rendered websites. For each, the notion of the firewall being situated between the frontend and backend is slightly different, but the function and role of Guarda is the same, filtering, verifying and proxying requests.

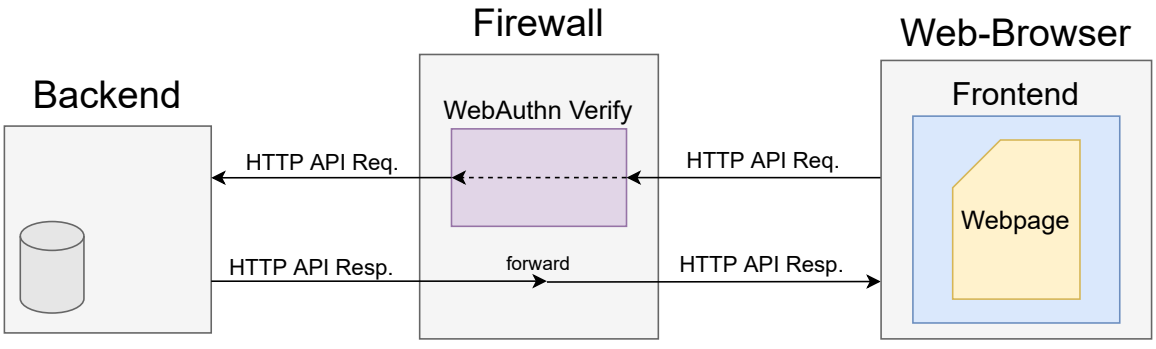

Figure 4-3: The positioning and role of the WebAuthn firewall in a RESTful paradigm web service.

For a RESTful web application, the placement of the firewall is more intuitive. As depicted in Figure 4-3, the firewall sits between the frontend and backend of the web service. In a RESTful design, the frontend runs in the web-browser and renders the webpage visible to the user. Whenever the frontend needs to interact with the backend, it launches HTTP requests to the IP address of the backend. However, since Guarda is situated between the two, the frontend must interact with the firewall instead. So when the frontend needs to issue a backend request, it sends it to Guarda rather than to the backend.

From there, as represented by the "Firewall" box in Figure 4-3, the firewall performs its role and, as necessary, proxies onward to the actual backend. Responses from the backend are returned to the firewall which are automatically forwarded on to the frontend.

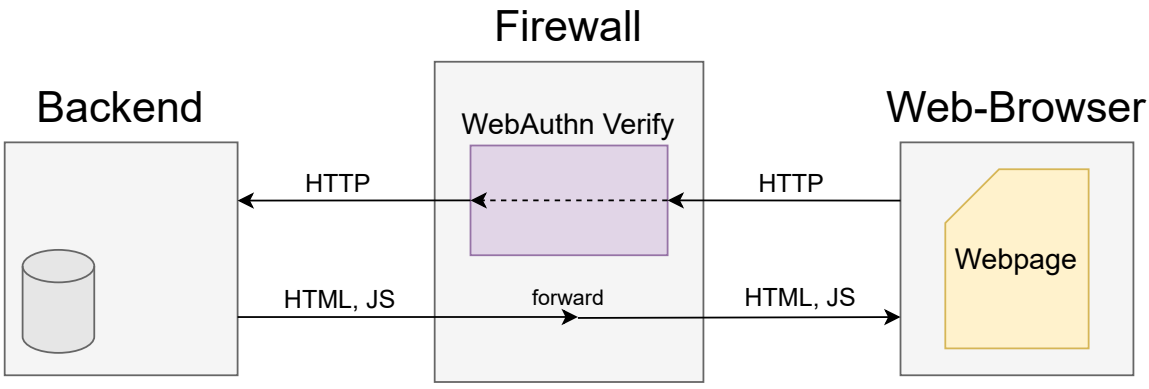

Figure 4-4: The positioning and role of the WebAuthn firewall in a server-side rendering paradigm web service.

In a server-side rendering web application, there is no notion of a frontend rendering the webpage like in the RESTful use case. Rather, every webpage visible to the user is generated by the backend and sent to the web-browser. From the web-browser, the webpage interacts directly with the IP address of the backend. With the WebAuthn firewall in place as shown in Figure 4-4, the endpoint of the webpage is Guarda instead of the backend. This way, all HTTP requests that originate from the webpage are sent to Guarda. The firewall processes these requests and relays them onward to the server-side rendering web application if the WebAuthn verification code

passes. In such a setup, the WebAuthn firewall is situated between the web-browser's webpage and backend.

4.3 WebAuthn Firewall Configuration

Some HTTP requests need to be transaction authenticated first before passing through the WebAuthn firewall. The software engineer configures Guarda to select which requests to verify and what their authentication messages should be. This is done by including the route in Guarda's configuration. Requests to all other routes not specified in the configuration are simply proxied on through to the backend without any checks.

4.3.1 Configuration Parameters

Guarda has a number of configurable parameters that aid and dictate how routes are secured. Code Snippet 4.4 is the firewall configuration for Conduit.

```
1  firewallConfigs := &wf.WebauthnFirewallConfig{
2    FrontendAddress:       frontendAddress,
3    ReverseProxyTargetMap: reverseProxyTargetMap,
4    ReverseProxyAddress:   reverseProxyAddress,
5
6    GetUserID: userIDFromJWT,
7    ContextGetters: wf.ContextGettersType{
8      "comment":      commentFromCommentID,
9      "article":      articleFromArticleSlug,
10     "current_user": getCurrentUser,
11   },
12
13   WebauthnCorePrefix: "/api/webauthn",
14   LoginURL:           "/api/users/login",
15   LoginGetUsername: func(r *wf.ExtendedRequest) (string, error) {
16     return r.Get_WithErr("user", "username")
17   },
18 }
```

Code Snippet 4.4: The firewall configuration for the Conduit web service.

The fields within the configuration are grouped by function. The WebAuthn library uses the first group, `RPDisplayName` and `RPID`, when setting up and verifying

transaction authentication events.

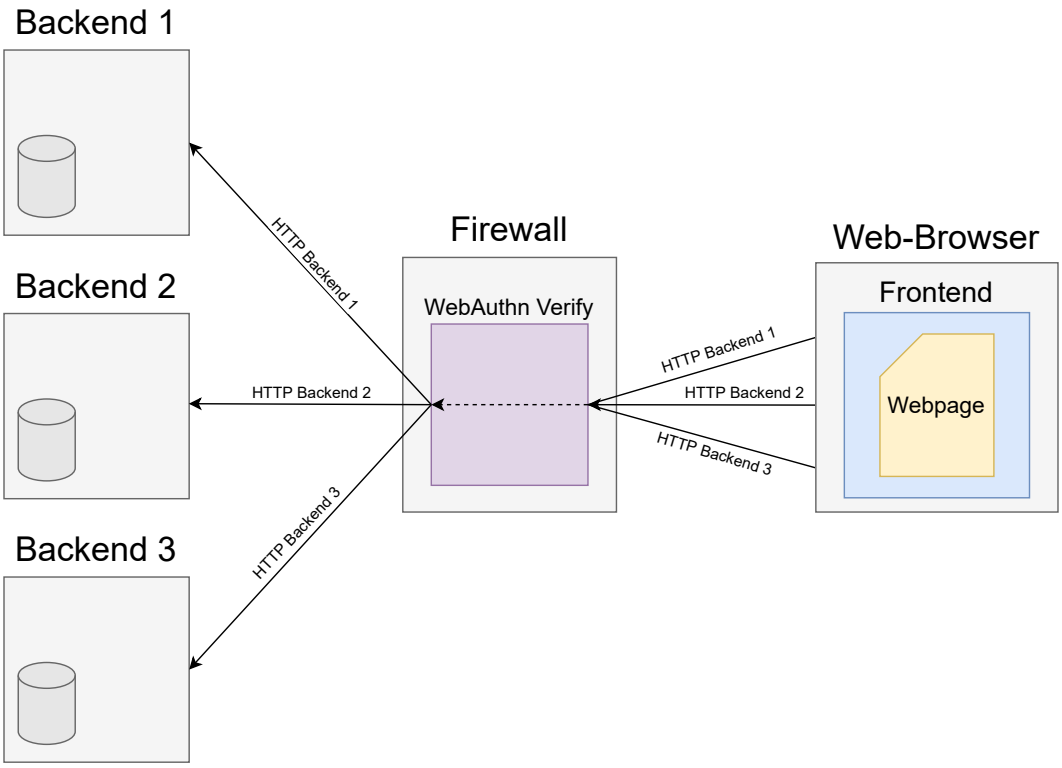

Figure 4-5: A WebAuthn firewall handles multiple backend targets for a single frontend.

The second group, `FrontendAddress`, `ReverseProxyTargetMap` and `ReverseProxyAddress`, contains the proxying address information. They hold the address of the frontend, the target backends and the firewall itself, respectively. It is possible for a web service's frontend to access multiple backends. In this case, the firewall must know which backend to forward incoming requests to after processing them. This is stored as a hash map of hosts and target backends in `ReverseProxyTargetMap`. This field simply configures the firewall, as illustrated in Figure 4-5, to catch requests of a given `host` address and forward them to a specific `target`.

The third group is for context retrieval, explained in greater detail in Section 4.3.3. During normal operation of the firewall, such as verifying incoming HTTP requests, it is often useful to identify the current user issuing that request. For example, the public key credential of a user is stored under their ID in Guarda's database.

Since determining the current user's ID is application specific, the `GetUserID` is left

for the engineer to supply. The example configuration above implements a function that extracts the current user's ID from the JSON Web Token (JWT) included in the HTTP request headers.

Constructing an authentication message oftentimes needs more context than that which is supplied the HTTP request payload. The `ContextGetters` is a collection of context getter functions written by the engineer. Essentially, they retrieve supplemental information needed to generate the authentication message of a protected HTTP request in a human-readable form. When setting up routes to protect, there is a clean way in the domain specific language to invoke these functions to fetch more context as needed.

The fourth group configures the default handlers, explained in more detail in Section 4.3.4. When integrating WebAuthn into a service, there are a number of operations that are a core part of the WebAuthn protocol. They are related to the WebAuthn registration event, the setup event, user login and WebAuthn disable. These fields provide the default handlers with just enough information such that they can be tailored for the given web service.

The final group contains miscellaneous parameters. Sometimes a web service's frontend expects the HTTP OPTIONS for every URL route it interacts with. The `SupplyOptions` flag controls whether Guarda should supply OPTIONS for every protected route.

4.3.2 Default Input Getters

An HTTP request may contain and encode its payload in a variety of ways. Guarda supplies four default input getter functions that extract values from requests in different formats. Each default input getter follows the same API, so new ones can be implemented easily as necessary. The getters include:

- `GetFormInput`: Parses form-data request payloads.
- `GetJSONInput`: Parses JSON request payloads.
- `GetURLInput`: Parses values stored in the HTTP request url. An example would be `"/user/comment/2"` parsing the comment ID `2` from the URL.

- `GetURLParamInput`: Parses parameters passed along with the HTTP request url. An example would be `"/user/comment?id=7"` parsing the comment ID `7` from the URL.

4.3.3 Context Retrieval

When an HTTP request contains non-human friendly identifiers that need to be understood by the user, those identifiers must be translated to their human-readable counterparts. For example, an ID identifies a user's comment to a blog post in the Conduit application. Showing the ID in an authentication message is meaningless to the user. So rather the ID must be translated to the comment's title, which the user can comprehend. This translation is handled by context getter functions programmed by the engineer. Generally, fetching context involves querying the backend for the extra information. This is necessary whenever the contents of an HTTP request payload is not human-readable.

4.3.4 Default Handlers

There are a number of core WebAuthn operations that are constant regardless of the application. Every web application wishing to integrate WebAuthn must support WebAuthn login, registration, disabling (de-registration) and the setup phase of a transaction authentication event. The WebAuthn firewall provides all of these functions transparently. Also, the frontend should be able to query Guarda to see if a user has WebAuthn enabled or not. This is primarily used in the security settings panel of the frontend to determine whether to have an `"Enable WebAuthn"` or `"Disable WebAuthn"` button.

4.3.5 Domain Specific Language

The majority of Guarda's routing configuration is performed using a domain specific language. Small chunks of code containing domain specific programs secure individual routes. Securing a route involves a `Secure` function which takes three parameters. Two

parameters, `url` and `method`, specify which route and which HTTP verb requests (such as POST, GET, etc.) on said route to intercept. The last parameter is a handler function, `handleFn` which verifies the WebAuthn transaction authentication event on that route.

The common case is to write a domain specific program to implement the handler function. The `Authn` function facilitates producing a handler function from a domain specific program. Sometimes there are edge cases where the handler function must manipulate or parse the incoming HTTP request in an atypical way. When the domain specific language cannot capture this behavior, the engineer can write a custom handler directly, described in Section 4.3.6, to utilize the full power of the Go programming language.

The purpose of a domain specific program is to generate the expected authentication message of a transaction authentication event. Section 4.4 explains in more detail how the authentication message should be formatted.

The first argument of the `Authn` function is the authentication message format string with format tags such as `"%v"`. The rest of the arguments make up the domain specific program. The DSL provides an assortment of operations. Some operations replace the format tags in order. Other operations do not affect the format string, but rather facilitate DSL functionality.

The following code snippet is an example of how the `Authn` function is used to write a simple domain specific program. Code Snippet 4.5 transaction authenticates the route for a Gogs user leaving a Gogs repository as a collaborator. Code Snippet 4.6 describes a possible request sent to this route.

```
1 Secure("POST", "/user/settings/repositories/leave",
2     Authn("Leave repository named: %v",
3       wf.SetContextVar("repo", wf.Get("id")),
4       wf.GetVar("repo").SubField("Name"),
5 ))
```

Code Snippet 4.5: A domain specific program to secure the leave Gogs repository operation.

For example, assuming that there exists a repository with ID 9 named “algo-price-

```
route: "/user/settings/repositories/leave",
form-data: {
  id: 9,
  assertion: "<assertion data from hardware authenticator>"
}
```

Code Snippet 4.6: A sample HTTP request to leave the repository with ID 9.

watcher", the authentication message generated to verify the `"assertion"` data of the request would be `"Leave repository named: algo-price-watcher"`.

Domain specific language operations that affect the format string are listed in Table 4.1. Included are the operations that parse values from the HTTP request being verified. Other operations access values from different channels like the context retrieval functions or the scope of the domain specific program. It is important to note that they only affect the authentication message format string if they are invoked at the top level within `Authn`. If they are used as input arguments to other DSL operations, they simply return and pass on their value.

Domain specific language operations that facilitate the DSL, but do not affect the format string are listed in Table 4.2. These operations generally perform some side-effect, but are not directly used to construct the format string. Included are the operations which assign values to variables in the scope of the domain specific program.

The `Authn` example in Code Snippet 4.5 presents a simple, but instructive use-case of the domain specific language. The following is a line-by-line explanation of that example. Its format string is `"Leave repository named: %v"`. Line three retrieves a Gogs repository context based on the `"id"` included in the HTTP request being protected. The repository context is stored within a variable named `"repo"`. Line four retrieves the value of `"repo"` and indexes a sub-field named `"Name"`. Since this `GetVar` call is the first format string modifier operation to appear at the top-level of `Authn`, it replaces the first `"%v"` in the format string, in this case with the name of the repository.

From there, the `Authn` function wraps this domain specific program with all of the boilerplate code needed to verify the WebAuthn transaction. More details on the

verification process are in Section 3.4.

4.3.6 Custom Handlers

Guarda provides a domain specific language that usually can secure most routes of a web service. However, occasionally some routes are more complicated to protect and fall outside of the capabilities of the domain specific language. In which case, the software engineer can implement a custom handler utilizing the full power of the Go programming language to secure those routes.

To support this, the `Secure` function accepts as its third argument a handler function of a predefined type. The custom handler simply must adhere to that type. Common use cases for a custom handler is to perform some control flow decisions in the handler body or sometimes gather external context information to assemble the authentication message in a particular way. Either way, the custom handler usually culminates in calling `Authn` at the end, since it handles all of the boilerplate code to validate the WebAuthn transaction.

A custom handler is used in the Gogs web service. Gogs uses the same route for many different types of POST operations related to a repository's settings. Most of the actions that POST to this route are harmless, except the `"delete"` action. Code Snippet 4.7 describes the core behavior of the custom handler.

```
switch action {
  case "delete":
    // Handle deletion separately
    handlerFn = firewall.Authn(
      "Confirm repository delete: %s/%s",
      wf.Get_URL("username"),
      wf.Get_URL("reponame"),
    )
  default:
    // Proxy all other requests
    handlerFn = firewall.ProxyRequest
}
```

Code Snippet 4.7: The switch/case logic necessary to determine which requests need transaction authentication and which can pass through without any validation. Only requests corresponding to delete repository operations are authenticated.

The custom handler parses the HTTP request on that route and sees what the `"action"` field in the request's payload is. Only if the action is `"delete"` does the handler use `Authn` to validate the request. Otherwise, the custom handler lets the request pass right through without any checks.

4.4 Authentication Message

The engineer must specify the authentication message format for every request route that needs WebAuthn transaction authentication protection. Verifying an incoming HTTP request with transaction authentication begins with Guarda generating an authentication message from the parameters of the request. Then the verification passes only if the generated message is exactly identical to the message signed by the hardware authenticator and the signature is valid.

The message generated by Guarda should encapsulate the entire intent of the request in order to have good security guarantees. All of the parameters in the requests that are considered security sensitive must appear unambiguously in the authentication message in a human-readable format. Unambiguity refers to how no two different requests should share the same authentication message. Human-readability is paramount since the user is the one authenticating the message and sometimes the parameters in the request may be not human intelligible such as item IDs, etc. The context retrieval functions from Section 4.3.3 makes these identifiers into their human-readable counterparts.

4.5 Frontend Modifications

One of the main objectives of the WebAuthn firewall design is to minimize the intrusiveness of integrating WebAuthn into a web service. However, some modifications to the frontend are unavoidable as the frontend is the one to initiate any transaction authentication event. A WebAuthn transaction authentication event has a life-cycle of setup and verification detailed in Chapter 3, which the frontend must support.

In order to minimize frontend intrusiveness, a small WebAuthn JavaScript library included with the firewall handles all of the frontend boilerplate code surrounding this life-cycle. Every endpoint in the frontend that launches transaction authentication requests may use this library to lighten the programming burden of integrating WebAuthn.

4.6 Backend Modifications

Guarda mostly avoids any backend modifications to integrate WebAuthn transaction authentication into a web service. The firewall design intentionally situates itself between the frontend and the backend and requests are proxied between the two such that the backend is unaware it is connected to Guarda rather than the frontend itself.

Of all three case studies, backend modifications are necessary for Gogs. As a server-side rendering application, there is no clean way for the firewall to do its context retrieval. The backend is modified to include a few server context routes, which the firewall may query for supplemental context. Section 7.3 compares the intrusiveness and complexity of integrating WebAuthn directly into Gogs versus utilizing Guarda. The comparison exemplifies that the firewall is the simpler approach to integrating WebAuthn.

Operation	Description
`Get`	Retrieve a value from the HTTP request using the default input getter explained in Section 4.3.1.
`GetInt64`	Similar to `Get`, but returns the value as an `int64` type.
`GetArray`	Similar to `Get`, but returns the value as an `[]interface{}` array type.
`Get_Form`, `Get_URL`, `Get_JSON`, `Get_URLParam`	Get functions that use specific default input getter functions. Each one of these operations has their respective `int64` and `[]interface{}` variants: `GetInt64_Form`, `GetArray_Form`, etc.
`GetUserID`	Retrieve the current user's ID from the HTTP request. Internally performs a call to the `GetUserID` function from the firewall configuration as explained in Section 4.3.1.
`GetContext`	Fetch some extra context by name using the context retrieval functions explained in Section 4.3.3. Example from Gogs: `wf.GetContext("repo", wf.Get("id"))` gets Gogs repository by id.
`GetVar`	Retrieve a store value by variable name within the scope of the domain specific program. Variable are set by a few Set operations explained in Table 4.2.
`Apply`	Applies a Go function to outputs of DSL operations. The result of the function feeds into the format string.

Table 4.1: The domain specific language `Get` type operations. These affect the format string if invoked at the top level within `Authn`.

Operation	Description
`SetVar`	Creates a new variable in the scope of the domain specific program and sets a value to it. Example: `wf.SetVar("id", wf.Get("id"))`.
`SetContextVar`	Shorthand for combining the output of a `GetContext` into `SetVar`. Example: `wf.SetContextVar("repo", wf.Get("id"))` is equivalent to `wf.SetVar("repo", wf.GetContext("repo", wf.Get("id")))`.
`Log`	Logs values to the firewalls console. Useful for debugging.

Table 4.2: The domain specific language `Set` type operations. These do not affect the format string, but generally perform some side-effect.

Chapter 5

WebAuthn Firewall Implementation

The WebAuthn firewall presented in this thesis is implemented in the Go programming language. This chapter describes how some of the critical components of Guarda's design described in Chapter 4 are implemented. The following Table 5.1 outlines the approximate footprint of each component discussed in this chapter.

Firewall Component	Lines of Code
WebAuthn Verification	126
Default Handlers	309
Domain Specific Language	478

Table 5.1: The domain specific language `Get` type operations. These affect the format string if invoked at the top level within `Authn`.

5.1 WebAuthn Verification

The cryptographic verification of a WebAuthn transaction authentication event uses a slightly modified Go WebAuthn library [15]. This library exposes two functions, `BeginLogin` and `FinishLogin`, which set up and validate a WebAuthn two-factor au-

thentication event respectively. The `FinishLogin` function is modified to support the transaction authentication extension. It checks if the extensions object received from the hardware authenticator exactly match an expected extensions object generated by the firewall.

5.2 Default Handlers

As discussed in 4.3.4, every web application with WebAuthn transaction authentication must support a list of core WebAuthn operations. Guarda secures them transparently with a collection of default handlers. They can be grouped in Table 5.2 by their similar implementations.

Function	Description
`webauthnIsEnabled`	Queries the firewall's database to determine if a `username` has WebAuthn enabled or not.
`beginRegister`, `beginLogin`, `beginAttestation`	Functions that setup their respective operations as described in Section 5.1.
`finishRegister`, `finishLogin`	Validates WebAuthn public key credentials. If successful, `finishRegister` saves the credentials in the firewall's database, whereas `finishLogin` allows the login to continue.
`disableWebauthn`	Validates WebAuthn public key credential and authentication message `"Confirm disable WebAuthn for {{ username }}"`. If successful, it deletes the credential from the firewall's database.

Table 5.2: The default handlers included with the WebAuthn firewall.

5.3 Domain Specific Language

Guarda secures most routes using the domain specific language within the `Authn` function described in Section 4.3.5. Any domain specific program has a state and an output container. A `scope` hash table holds the state of the program — all of the

local variables which may be set and accessed. The has table maps strings representing variable names to their respective contained values. The output container is a `formatVars` array. This array stores the values to apply sequentially to the format tags included in the first argument of `Authn`.

The first argument of the `Authn` function is the format string. The second argument and onward are the top-level DSL operations. Only the `Get` type operations listed in Table 4.1 which appear as top-level operations may affect the resulting authentication message. All other occurrences, such as arguments to other domain specific operations, simply return their respective values.

Each DSL operation must implement an `execute` and `retrieve` function. The `Authn` executes the domain specific program line-by-line by calling `execute` of the top-level operations in order. Since `execute` is only called on top-level operations, which may affect the authentication message, it has access to both the `scope` and the output `formatVars` array. The function type of `execute` is produced in Code Snippet 5.1.

```
1 execute(r *ExtendedRequest,
2         scope scopeContainer,
3         formatVars *[]interface{})
```

Code Snippet 5.1: The function type of the `execute` function.

Table 4.1 outlines the `Get` type operations. When `execute` is called on one such operation, it typically appends to the `formatVars` array. The values of `formatVars` substitute the format tags in the format string in order. Table 4.2 outlines the `Set` type operations and calling `execute` typically adds or sets a new variable to the `scope` hash table.

The `retrieve` function of a DSL operation is used to extract a return value from the operation. Whenever a DSL operation is passed as an argument to another operation, the parent operation calls the child's `retrieve` in order to resolve the return value. Since these operations are not top-level, they may not affect the format string. Therefore, they do not receive the `formatVars` array, only the `scope`.

Take the following example from Gogs: `SetContextVar("repo", Get("id"))`. The outer `SetContextVar` calls `retrieve` on the inner `Get("id")` to resolve its return value.

The function type of `retrieve` is produce in Code Snippet 5.2.

```
1 retrieve(r *ExtendedRequest, scope scopeContainer) interface{}
```

Code Snippet 5.2: The function type of the `retrieve` function.

When a DSL operation implements these two functions, it enables a hierarchical domain specific language. A DSL operation may invoke other operations as arguments to it or be an argument to another operation. This chaining enables flexible domain specific programs.

Chapter 6

Case Studies

This chapter presents three case studies used to evaluate Guarda. Each case study involves using Guarda on a web service that is unique in some fundamental way. This variability stress tests Guarda's two main objectives: minimal intrusiveness and ease of configuration. The case studies provide guidance on what aspects of Guarda need configuration and precisely how detailed versus general those configurable options should be to be beneficial without becoming overly burdensome.

6.1 Conduit

Conduit is a simple RESTful blog web service [5]. It is not a production service, rather an educational service to demonstrate the flexibility of RESTful web applications. The project supplies many simple frontend and backend implementations, written in various programming languages and frameworks, but following the same API. The case study focused on a React based frontend [4] and a Golang based backend [3]. Conduit is the best-case application for the WebAuthn firewall, a RESTful web service with basic functionality to protect. No backend modifications are necessary to integrate WebAuthn.

6.1.1 Context Retrieval

A major benefit of a RESTful web application is that the backend exposes many useful context routes. By design, a RESTful frontend performs the necessary rendering on the user's web-browser and must retrieve user specific information from the backend. These routes are useful for WebAuthn context retrieval as well. For example, the backend already implements routes such as getting an article by its ID. And if not, then there certainly will be a tangential route such as requesting all comments of an article to search for a specific one by ID. Since most of the modifications necessary to the backend when integrating the WebAuthn firewall are for context routes, this aspect of RESTful applications make that easy.

6.1.2 Secured Routes

The Conduit case study has three protected routes. Table 6.1 lists the secured operations with a sample authentication message for each. It presents an overview of how Conduit is protected with transaction authentication.

Operation	Authentication Message
Delete Comment	`"Confirm comment delete: I love WebAuthn"`
Delete Article	`"Confirm article delete: Cat Memes"`

Update User Settings	`"Confirm new user details:`
	`  username damian`
	`  email damianb@mit.edu"`

Table 6.1: The operations of Conduit secured by transaction authentication.

The simplicity of the Conduit application does not challenge the domain specific language much. The domain specific programs simply retrieve some context to complete the format strings. The "Update User Settings" route requires a custom handler, which is not exceptionally complicated. The custom handler transaction authenticates requests only if the username, email or password are modified. Otherwise,

HTTP requests that only modify non-sensitive fields like the bio or profile pictures pass through without any verification.

6.2 Calypso

Calypso is a RESTful frontend for a WordPress admin panel [7]. This is a production service, with far greater complexity than the Conduit application. Also, unlike Conduit with a backend running locally, which can be modified, Calypso accesses the official WordPress backend servers. These servers are closed-source and their modification is out of question. Whereas avoiding backend modifications for Conduit is a favorable result, with Calypso it is a necessity. Nonetheless, integrating WebAuthn is possible because the RESTful API of WordPress was complete enough to satisfy the needs of Guarda.

6.2.1 Multi-Target Proxying

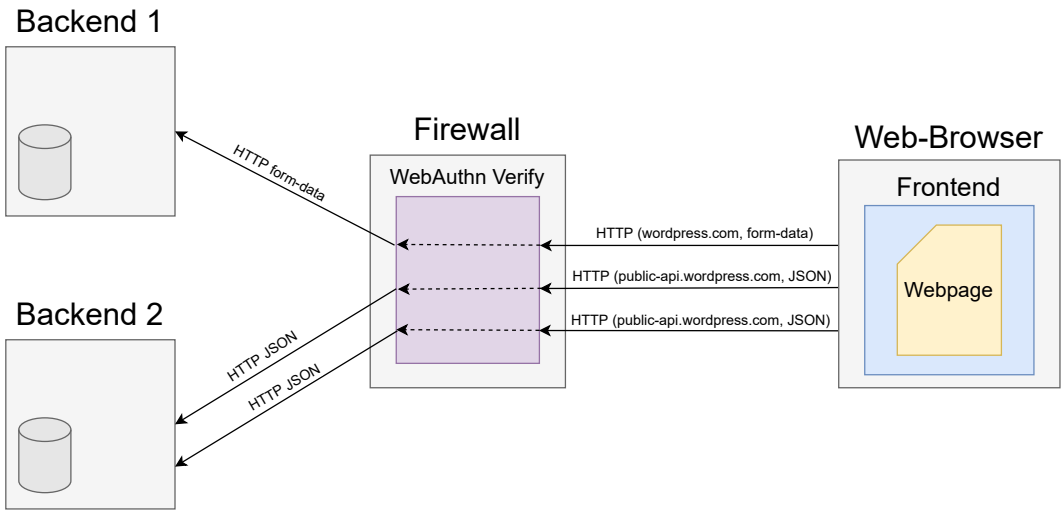

Figure 6-1: The WebAuthn firewall must support two backend targets for Calypso.

Unlike the other two case studies, a point of complexity that Calypso has is that the frontend accesses multiple backends. Typically, the frontend requests to a single backend source. However, as shown in Figure 6-1, the Calypso frontend interfaces with two backend targets. Guarda receives requests with target addresses

"public-api.wordpress.com" and "wordpress.com" in their headers. It must proxy them to their respective backends accordingly. One WordPress backend is for API requests located at "public-api.wordpress.com" and accepts JSON payloads. Most of Calypso interacts with this backend. However, login requests go to a backend at "wordpress.com" which uses form-data payloads. This architecture design requires that Guarda support multi-target proxying with separate default input getter functions per target. The firewall configuration for Calypso lists both of those target domains along with `GetJSONInput` and `GetFormInput` as their default input handlers respectively.

6.2.2 Login

Guarda transparently protects a number of core WebAuthn routes, including the route for login, as described in Section 4.3.4. However, the Calypso project approaches login in an atypical fashion. Normally a service has a dedicated route for login. Calypso shares a login route with other operations, so HTTP requests are identified by an "action" field in their payload, similarly as described in Section 4.3.6. As a result, the default login handler cannot support Calypso directly. Extending the default login handler to be customizable for this one specific case would make Guarda's configuration too complicated. Rather, a custom login handler implements the Calypso login event. The login handler roughly resembles the Go pseudo-code in Code Snippet 6.1.

```
1  func finishLogin(w http.ResponseWriter, req *wf.ExtendedRequest) {
2    action := req.Get("action")
3    switch action {
4      case "login-endpoint":
5        // WebAuthn verify the login request
6        VerifyLogin(w, req)
7      default:
8        // Other requests go right through
9        ProxyRequest(w, req)
10   }
11 }
```

Code Snippet 6.1: Go pseudo-code for the custom login handler for Calypso.

The handler listens on the "wordpress.com/wp-login.php" route. It extracts the "action" field in the request payload and validates the login attempt only if the action

is `"login-endpoint"`. The `VerifyLogin` function performs the WebAuthn authentication and only upon successful verification does it allow the request to proceed through the firewall. Otherwise, all other requests are proxied through with no additional checks.

6.2.3 Secured Routes

Table 6.2 lists the secured operations of Calypso with a sample authentication message for each. The domain specific language supports Calypso well; every operation listed is secured using the domain specific language. This array of protected operations demonstrates the flexibility of the domain specific language.

Operation	Authentication Message
Update Profile Settings	`"Save the profile settings: Espanol Public"`
Invite New Users to Site	`"Invite new users: damian, durian, john-smithson"`

Change Site Address	`"Change site address`
	`  from: calypso.localhost`
	`  to: mytravels.blog.com"`

Change Site Theme	`"Change theme to: Tangerine Orange"`

Table 6.2: The operations of Calypso secured by transaction authentication.

Certain domain specific programs have multiple context retrievals. Others have multiple format tags to fill, and one program requires a special formatting function. Apart from the custom login handler, there is no case where a custom handler is necessary. The "Invite New Users to Site" route receives HTTP request containing an array of elements that must be comma separated in the authentication message. Rather than implementing a custom handler for this minor inconvenience, the `Apply` domain specific operation described in Table 4.1 resolves this problem. It enables Go code to be used within a domain specific program to a limited extent. Code Snippet 6.2 is a domain specific program for inviting new users to administer a WordPress blog.

```
Secure("POST", "/rest/{version}/sites/{site_id}/invites/new",
  Authn("Invite new user(s): %v",
    wf.Apply(func(args ...interface{}) (interface{}, error) {
      invitees := args[0].([]string)
      return strings.Join(invitees, ","), nil
    }, wf.GetArray("invitees")),
))
```

Code Snippet 6.2: A domain specific program to secure the Calypso operation for inviting new users to administer a WordPress blog.

The `Authn` operation string formats the argument of `wf.GetArray("invitees"))` with a Go closure passed to `Apply`.

6.3 Gogs

Gogs is a server-side rendered self-hosted Git web service [2]. A server-side rendered web application presents its own set of challenges to the WebAuthn firewall. Section 4.2 explains how Guarda has to be the end-point interacting with the user's web-browser. The firewall must obtain the context from the Gogs backend. The backend does not, however, conveniently expose context routes like RESTful backends as discussed in Section 6.1.1.

6.3.1 Intrusive WebAuthn

The Gogs web service was the first of the three case-studies. Before Guarda became a mature idea, part of Gogs was secured in the traditional, intrusive fashion described in Section 1.4. This work is redone using the WebAuthn firewall approach once it proved itself as a prospective design approach.

The two main challenges with integrating WebAuthn into Gogs intrusively are the database adapter and organizational difficulties. WebAuthn needs a database table to record entries of users' public key credentials. Simply creating a new table in Gogs requires writing a custom database model for the table as well as modifying the codebase in a number of separate locations. Furthermore, the handlers for various Gogs routes are spread throughout the code. Keeping track of which routes are

WebAuthn secured becomes increasingly more difficult as more routes are protected.

6.3.2 Context Retrieval

A downside to server-side rendering web services when compared to RESTful services when it comes to WebAuthn firewall integration is the lack of pre-built context routes. Effectively, the API routes in the RESTful backend services act as the context routes. Gogs, being a server-side rendering backend, must be modified to include those context routes. Some Gogs routes need additional context to retrieve objects such as a `"repository"` or `"webhook"`. Gogs serves this context information out of a `"server_context/<type>/<args>"` route. The `<type>` refers to what type of object is being requested (e.g., repository, webhook, etc.). Each context type has its own handler in Gogs, which interprets the `<args>` to retrieve the correct context object.

6.3.3 Secured Routes

While integrating WebAuthn into Gogs during the case study, every HTTP route of the web service is categorized whether it should be protected by transaction authentication or not. This judgment is made by weighing the cost versus the benefit of protecting that route. Protecting a route is not free, most heavily affecting the user experience. Routes are protected if the harm due to malicious hijacking justifies the burden to the user's experience.

Table 6.3 lists the secured operations with a sample authentication message for each. It presents an overview for the types of operations that could warrant WebAuthn transaction authentication.

6.3.4 Custom Handlers

As with the other case-studies, most of the routes are simple and easy to secure using the domain specific language. At most they require a context retrieval to fill a single format tag. Within the Gogs service, the "Delete Repository" and "Set Primary Email" operations in Table 6.3 are handled by HTTP routes that also handle

Operation	Authentication Message
Delete Repository	`"Confirm repository delete: damian/JS-OS"`
Add SSH Key	`"Add SSH key named: Damian's Laptop"`
Delete SSH Key	`"Delete SSH key named: Damian's Laptop"`

Update Profile Settings	`"Confirm profile details:`
	`  username damian`
	`  email damianb@mit.edu"`

Set Primary Email	`"Confirm new primary email: damianb@alum.mit.edu"`
Change Password	`"Confirm password change"`
Leave Repository	`"Leave repository named: JS-OS"`
Delete App Access Token	`"Delete App named: Gogs-Watcher-App"`
Publish New Release	`"Publish release named: Version 4.20` `  File names: release.tar.gz, release"`
Delete Web-Hook	`"Delete webhook for: URL gogswatcher.app.com"`

Table 6.3: The operations of Gogs secured by transaction authentication.

multiple other operation types. Similarly to the login route of Calypso discussed in Section 6.2.2, requests have an `"action"` field in their payload that delineates the operation type. Only certain actions warrant being transaction authenticated. Custom handlers are used in these cases, to parse the `"action"` field and WebAuthn secure only when necessary.

Gogs requires a more involved custom handler for protecting the "Publish New Release" operation. The files associated with a new release are contained as UUIDs in the HTTP request. So each file needs context retrieval to retrieve the file name, which then must be comma separated in the format string. Rather than expressing this series of operations and formatting in an `Apply` operation like with Calypso in

Section 6.2.3, it is easier to drop down to a custom handler function. The custom handler function written in Go pseudo-code is produced in Code Snippet 6.3.

```
func publishNewRelease(w http.ResponseWriter,
                       r *wf.ExtendedRequest) {
  title := r.Get("title")
  uuids := r.Request.Form["files"]

  names := []string{}
  for _, uuid := range uuids {
    append(names, r.GetContext("attachment", uuid)["Name"].(string))
  }

  authText := fmt.Sprintf("Publish release named: %v", title)
  authText += fmt.Sprintf("\nFile names: %s",
                          strings.Join(names, ", "))

  handlerFn := firewall.Authn(authText)
  handlerFn(w, r)
}
```

Code Snippet 6.3: A custom handler in Go pseudo-code to secure the Gogs operation for publishing a new release.

This handler constructs an authentication message from the `"title"` of the release and all of the included attachment file names. For the attachment file names, the handler must translate the UUIDs into contextualized attachment objects and then look up the `"Name"` fields.

Chapter 7

Evaluation

This chapter presents evaluation metrics for the three case studies of Guarda. The principal evaluation criteria focus on simplicity, organization, ease-of-use and performance overhead of Guarda by answering the following questions:

- What is the complexity of using Guarda? (Section 7.1.1)
  - What is the breakdown of the configuration file? (Table 7.1)
  - How much incremental work does it take to secure a new route? (Figure 7-1)
- How much does a web service's frontend code need to be modified? (Section 7.2)
- How do the changes to a web service's backend compare when integrating WebAuthn with Guarda versus intrusively? (Section 7.3)
- What is the performance overhead of Guarda? (Section 7.4)

7.1 WebAuthn Firewall Configuration Metrics

Each of the three case studies has its associated WebAuthn firewall configuration file. The number of lines of code of this file is a proxy measurement to evaluate the complexity and ease-of-use of Guarda.

7.1.1 Overall Complexity

The total configuration file size evaluates the overall complexity of configuring Guarda and is shown in Table 7.1, sub-divided into four categories:

- Configuration parameters as discussed in Section 4.3.1
- Context retrieval functions as discussed in Section 4.3.3
- Domain specific language and custom route handlers as discussed in Sections 4.3.5 and 4.3.6
- Miscellaneous code such as extraneous syntax and boilerplate code

The firewall configuration file is the sole source of customization that dictates how the firewall interacts with a specific web service. Both Calypso and Gogs are production web services. Considering that they are secured by Guarda within approximately 250 lines of code, this supports the claimed advantages of the firewall's simplicity and organization.

7.1.2 Incremental Complexity to Secure a Route

The configuration parameters, context retrieval and miscellaneous code are more or less fixed costs to the configuration file's complexity. Only the route handlers scales with the size of the service being secured. Figure 7-1 plots the frequencies of lines of code needed to secure a new route among all three case studies. The bar chart can be partitioned into two clear clusters at approximately the 20 lines of code per route handler boundary. The bars on the left side with fewer than 20 lines correspond to uses of the domain specific language. The handlers greater than 20 lines of code are solely the custom handlers. It is natural to expect this divide since the custom handlers by nature perform more involved processing of a request than the DSL handlers and thus use more lines of code.

This chart also evidently demonstrates that most of the routes may be secured using only the domain specific language. Of all 20 routes secured among all three case studies, only 5 required a custom handler. This means that 75% of the routes could be captured by the domain specific language. The average domain specific program

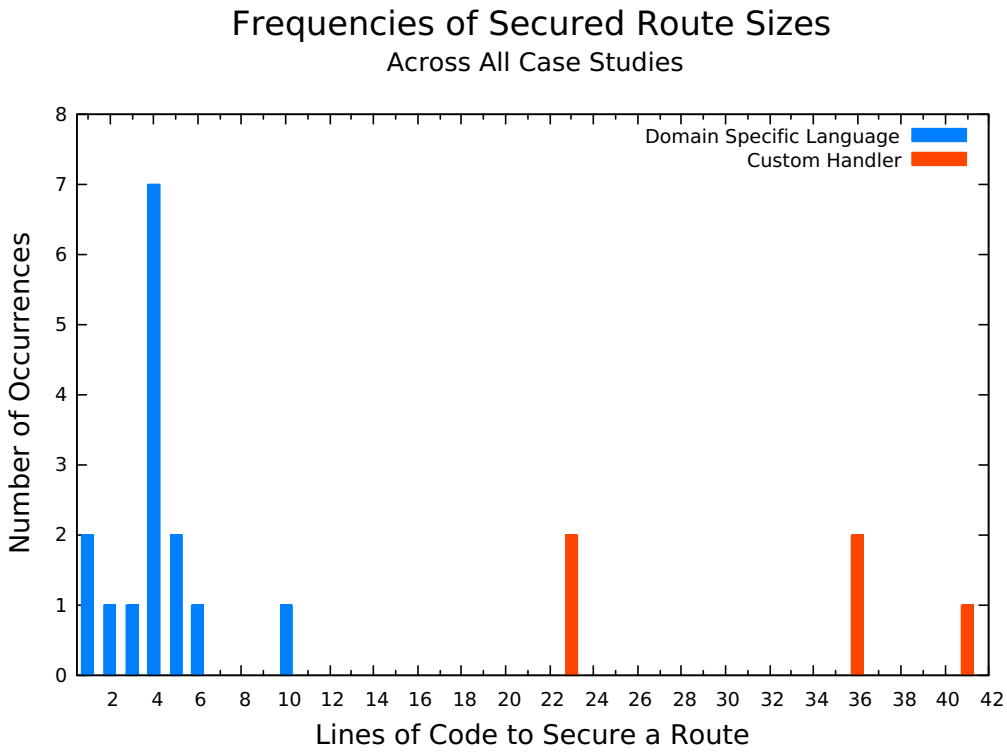

Figure 7-1: The frequencies of lines of code to secure a route with transaction authentication. The majority of the routes can be secured in 10 lines or less.

to secure a route is 4 lines of code, and the average custom handler is 32 lines of code. The total weighted average to secure a route is 11 lines of code.

7.2 Frontend Modifications

The frontend of a web service must be modified slightly when integrating WebAuthn, regardless of whether the WebAuthn firewall is in use or not. The frontend issuing the WebAuthn operation must adhere to the protocol specification life cycle as described in Chapter 3. Table 7.2 lists the total number of code changes for each frontend of the case studies. For every case, the number of changes all exceed the sizes of the WebAuthn firewall configuration files.

These code changes are not complex. There is boilerplate code delegated to a JavaScript library. Its contents are not included in the lines of code measurements, but its import and usage within the frontend is. The nature of protecting an operation

with WebAuthn on the frontend involves making a few library calls, error handling and piecing together the authentication message from the HTML data present. For example, each Gogs operation secured by WebAuthn involves on average 54 lines of code changes to the frontend.

7.3 Backend Modifications

A significant advantage to using Guarda over integrating WebAuthn intrusively into a web service is the lack of modifications needed to the backend. Table 7.3 demonstrates this fact — using Guarda is not invasive to the backend.

The RESTful case studies, Conduit and Calypso, require no backend modifications. The Gogs case study, which is server-side rendered, only needs modifications to supply context.

Before Guarda became a mature idea, Gogs was partly WebAuthn secured intrusively. The intrusive Gogs case study protects 5 routes whereas the WebAuthn firewall study of Gogs covers 11 routes. Nevertheless, even with fewer routes protected, the invasive nature of securing Gogs is unmistakable. Table 7.4 backs this claim up plainly. The intrusive Gogs implementation is significantly more complex and more spread throughout Gogs than the WebAuthn firewall approach.

Many of the additional lines of code for the intrusive WebAuthn integration of Gogs come from all of the plumbing code needed in to support and run WebAuthn. WebAuthn needs its own database table with entries to record registered users. The backend must support WebAuthn registration and login. It must also include the WebAuthn code that verifies transaction authentication events. All of these functionalities come with Guarda for free, which explains the large reduction in backend complexity when using Guarda.

Guarda also reduces the amount of boilerplate code required to secure a new route. Gogs secured intrusively takes on average 87 lines of code per new route. The firewall has on average 11 lines per new route.

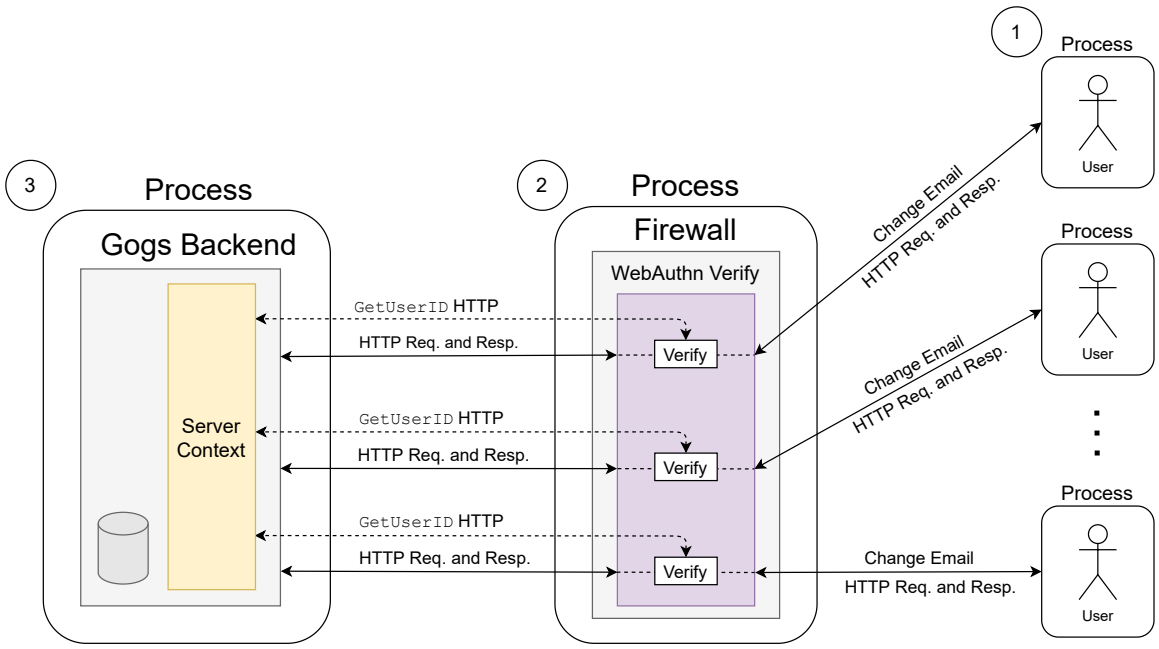

Figure 7-2: The experiment setup to measure the performance overhead of Guarda under load.

7.4 Performance Overhead

Guarda acts as a gatekeeper between the frontend and backend. It naturally adds some performance overhead to the whole system. This evaluation experiment measures the performance overhead for securing Gogs with Guarda under load to see how well it scales.

The hardware for this experiment is a quad-core Intel i7-6600U CPU @ 2.60GHz laptop machine. Figure 7-2 illustrates the setup used to collect the performance data. The firewall and Gogs web server run in separate operating system processes. Each trial to collect performance run times initializes a given number of users running in their own processes as well. Each user proceeds to sequentially send 300 POST requests to the "Change Email" route as quickly as possible. This workload generation is performed as a closed-loop system, where each user sends a new HTTP request only when their previous request comes back with an HTTP response. The latency of each request is the elapsed time measured for the request to return a response. The data point for the trial is the 95^{th} percentile of the running times tail. When there are many users sending requests simultaneously, there is contention among them which

accounts for the increase in latency.

Three different scenarios are tested and the results reported in Figure 7-3. The blue line measures the request latencies for using Guarda on users that have WebAuthn enabled. The orange line measures the request latencies for Guarda on users without WebAuthn enabled. The difference between these two lines is the overhead of validating a WebAuthn transaction versus simply letting the request pass through. The red line measures the latencies of sending these requests directly to the Gogs server.

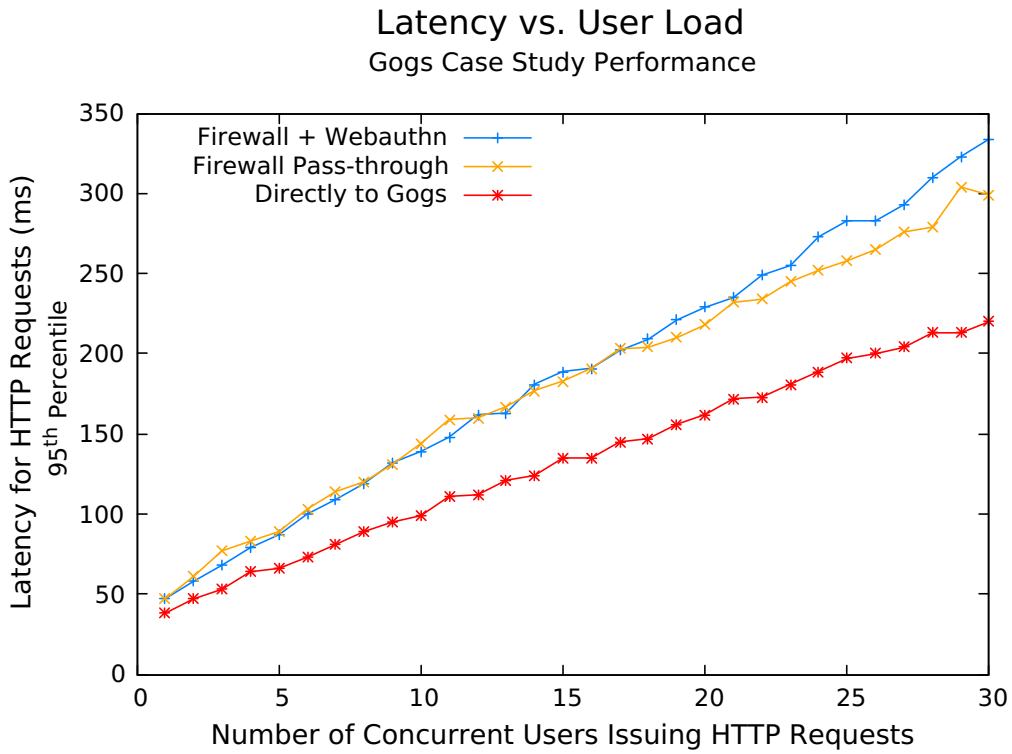

Figure 7-3: The 95th percentile run times of three different Gogs setups under load. The x-axis is the number of users issuing requests concurrently, and the y-axis is the run time latency in milliseconds. The three experiments are: using Guarda with WebAuthn enabled, using Guarda with WebAuthn disabled, issuing requests directly to the Gogs server without WebAuthn transaction authentication. There is a latency penalty to using the WebAuthn firewall.

As seen in Figure 7-3, the blue and orange line remain tightly together meaning that there is little overhead from the actual validation of a WebAuthn transaction. Rather there is a significant gap between the lower red line representing the direct

connection and the upper blue and orange lines representing the firewall setups. This discrepancy arises from the implementation of the `GetUserID` function within the Gogs firewall. The `GetUserID` is a configurable function described in Section 4.3.1 which identifies the current user of a request received by Guarda.

Guarda uses the current user information to determine, for every incoming HTTP request, whether that user has WebAuthn enabled and, if so, verify the transaction accordingly. In Gogs, the user of a request is identified by a session ID, which only the backend can translate to a user ID. Therefore, the `GetUserID` for the firewall must send an HTTP GET request to the backend to retrieve the user ID. As depicted by the dashed request lines in Figure 7-2, every request that passes through the firewall invokes an additional HTTP request, accounting for the noticeable overhead when using the firewall. Under a single user load, the additional overhead of this `GetUserID` HTTP request contributes approximately 4 milliseconds to the overall latency. Under the 30 user load when contention is high, the overhead from `GetUserID` is approximately 70 milliseconds. This is visible in the non-negligible gap between the lower red line and the upper blue and orange lines.

The `GetUserID` function is specific to the web service, and it is possible to design the software to avoid the additional HTTP call. The Conduit web service, for example, uses JSON Web Tokens (JWT) in order to identify the current user of an HTTP request. In this setup, the user ID is simply included in a signed JSON object with the request. Guarda can simply parse this data object and extract the identifier directly without any additional HTTP requests. Under such a setup, the latency discrepancy between using Guarda and the direct scenarios should diminish significantly.

Case Study	Configuration File Lines of Code
Conduit	245
Config Parameters	17
Context Retrieval	117
Route Handlers	49
Miscellaneous	62
Calypso	253
Config Parameters	22
Context Retrieval	92
Route Handlers	81
Miscellaneous	58
Gogs	257
Config Parameters	22
Context Retrieval	51
Route Handlers	126
Miscellaneous	58

Table 7.1: A breakdown of the configuration file size for each case study. Each total is broken down into four categories with their respective lines of code contributions.

Case Study	Frontend Lines of Code Changes
Conduit	289
Calypso	402
Gogs	570

Table 7.2: The number of code changes performed on each frontend of the case studies in order to support WebAuthn transaction authentication.

Case Study	Backend Lines of Code Changes
Conduit	0
Calypso	0
Gogs	169

Table 7.3: The number of code changes performed on each backend of the case studies in order to support the WebAuthn firewall.

	Backend Lines Changed	Backend Files Modified
Intrusive Gogs	1293	18
WebAuthn Firewall Gogs	169	5

Table 7.4: Comparing the complexity differences between intrusive and WebAuthn firewall integration.

Chapter 8

Discussion and Future Work

This chapter makes a number of observations on transaction authentication discovered over the course of this research.

8.1 Applications of Transaction Authentication

Transaction authentication is not without its limitations. There are use cases that make perfect sense for this authentication extension, which fit nicely within its specification and capabilities. Other cases may not lend themselves well to transaction authentication. The complexity of the authentication message displayed is a key determining factor for what makes a good versus poor use case.

8.1.1 Good Use Cases

Good use cases of transaction authentication are those where the authentication message is short, simple and easy to comprehend by a user. The contents being displayed should also be human-readable in nature. Names and titles are easily recognizable by a human. A cryptographic key, albeit displayable, is much less human-friendly.

An example from Gogs is the delete repository action. It is secured with the authentication message: `"Confirm repository delete: {username}/{reponame}`. From this message, it should be immediately evident to a user what the operation to be per-

formed is.

A short authentication message is less likely to be misinterpreted by the user. Additionally, a hardware authenticator device is likely to have a relatively small display. So shorter authentication messages are better fit for this restrictive display medium.

8.1.2 Poor Use Cases

A poor use case is one where it is possible to use transaction authentication, but it is rather clunky and disruptive to the user experience. Three general classes of such problems are as follows:

- There is a lot of context to authenticate. One example is a big form that needs transaction authentication. This form is unlikely to all fit on the hardware authenticator's display. Also, even when only one aspect of the form is modified, all of the entries must be displayed to the user since they all are sent over in the HTTP request.
- The contents being displayed are not friendly for human readability. Examples include SSH keys, cryptographic hashes, and Git hooks code blocks. These can be displayed, but are burdensome for the user to verify.
- The context media is difficult or impossible to meaningfully be displayed. Rich media such as images or audio clips fully depend on the hardware authenticator's physical capabilities. Binary uploads have no good way of being displayed to the user at all. For example, the Gogs route to publish a new release as discussed in Section 6.3.4 has no way to validate the binary contents being uploaded.

8.1.3 Inapplicable Use Cases

Transaction authentication defends against unauthorized and unwanted operations on a user's account. However, under the threat model defined in Section 1.2, it is incapable of performing any form of secure input. That means that transaction authentication cannot prevent a malicious keyboard sniffer from recording a credit card number being input during checkout or a new password being set in the account

settings. Transaction authentication may prevent these events from being maliciously initiated by an adversary, but it has no means of preventing any snooping on sensitive data as it is entered into a website.

8.2 RPC Isolation

Transaction authentication is often understood as a mechanism to protect the user by defending against unauthorized operations. Intrusive integration of transaction authentication into a web service has a collateral benefit of being able to shrink the trusted computing base on the server-side. The original threat model assumes that all server code is secure. However, if a web service RPC isolates its various components internally, then only the component actually executing the given operation must be trusted.

For example, the Gogs web sever has code and database tables pertaining to repository actions. They are independent of any other server operations and can be RPC isolated into a separate operating system process. This process is the only one with the privileges to execute repository actions, from renaming to deleting a repository. Consequently, the web server must interface with this process in order to operate on any repository. If this RPC isolated process also contains WebAuthn verification code, it can perform the transaction authentication validation on its own. Even if any other aspect of the web server were compromised, the WebAuthn protected repository requests sent to this process cannot be tampered with since they would fail the verification.

8.3 Tracing Transaction Authentication Subversion Opportunities

Securing a route with transaction authentication requires careful planning to avoid any vulnerabilities of directly or indirectly subverting the protection. Direct subversion actively evades the transaction authentication protection. Indirect subversion

tricks the user into transaction authenticating some operation that actually is undesired.

8.3.1 Direct Subversion

If the disable WebAuthn event were not transaction authentication secured, a trivial example of direct subversion is possible. The adversary could disable WebAuthn before performing any sensitive operation. To avoid this, the disable WebAuthn operation is one of the default operations protected by Guarda as discussed in Section 4.3.1.

Less obvious direct subversion attacks are also possible. In Gogs, an adversary could create a new dummy user without WebAuthn enabled. Then they could add the dummy user as an owner of a target repository and perform the delete operation from the unsecured user. If not carefully protected, such as requiring transaction authentication also for adding new repository owners, this sequence of operations could subvert the protected route of repository deletion.

Direct subversion opportunities might be detectable by performing route search and analysis similar to symbolic execution. Such a system could scan the web service's code and trace all possible chains of operations that could get around the transaction authentication protection of a specific route.

8.3.2 Indirect Subversion

Indirect subversion opportunities are harder to detect since they involve a human element. These attacks trick the user into transaction authenticating a seemingly innocuous operation when actually they are authorizing something sensitive.

An example within Gogs is the following scenario. A user has two repositories A and B. The repository A is important whereas B is garbage. In this scenario, repositories may be deleted or renamed. Deletion is WebAuthn protected whereas renaming is not.

When the user decides to delete the garbage repository B, the adversary quickly

and quietly renames A to B and B to A. So by the time the user authorizes the deletion of B, repository B is actually the important one. In doing so, the adversary hijacked the user's faithful deletion of the original garbage repository B in order to delete the important repository A.

The user believes that they are doing one operation, but in reality, they are doing a completely different and destructive other operation. Indirect subversion opportunities are difficult to notice and detect since, unlike the direct subversion opportunities, there need not be any causal links between operations. In the aforementioned example, there is no codified link between renaming a repository and its deletion. This attack purely relies on silently duping the user.

Chapter 9

Conclusion

This thesis presents Guarda, a WebAuthn firewall architecture with the central aim of reducing the burden for integrating transaction authentication into a new or existing web service. Before this research, WebAuthn would be integrated directly into the web service's codebase, which is time consuming and difficult to maintain. The firewall approach enables an engineer to configure a ruleset for the firewall using the domain specific language and, if necessary, custom handlers. This configuration determines which HTTP requests are high-risk and which are not. The firewall processes those that are deemed high-risk and only allows them to pass through if the transaction authentication validates successfully.

Three case studies with evaluations justify the advantages of this system over integrating WebAuthn intrusively.

- Complexity: The average firewall configuration file size among the case studies is approximately 250 lines of code. The average secured route for Gogs done intrusively is 87 lines of code. Considering all of the boilerplate code necessary to support WebAuthn, the firewall approach is almost 8 times more concise than the intrusive approach.
- Configurability: Among the case studies, 75% of all routes can be secured using the domain specific language in 20 lines of code or less. The average secured route requires 11 lines of code, so making adjustments to the configuration is simple and painless.

- Intrusiveness: In both RESTful case studies using Guarda, no modifications are necessary to the backend. The Gogs case study with Guarda requires 169 lines of code changes. Contrasted to the almost 1300 lines of code change over 18 different files for the intrusive Gogs case study, the firewall is the significantly less invasive approach.

The firewall does incur a performance penalty, which varies depending on the implementation details of the web service. The session ID implementation of Gogs requires that Guarda make an HTTP request in order to identify the current user issuing a request. This identification procedure slows down the entire latency of the request being authenticated. Under heavy load, the request latency may be up to 1.5x slower when using the firewall than accessing the web service directly without Guarda.

In conclusion, the WebAuthn firewall architecture is flexible and extensible. With relatively little effort, it can equip even a production level web service with WebAuthn transaction authentication. When well tuned for performance, such a WebAuthn firewall architecture design is an appealing option for greater web security.

Bibliography

[1] Github: Where the world builds software. https://github.com.

[2] Gogs — a painless self-hosted git service. https://github.com/gogs/gogs, 2021.

[3] Golang echo realworld example app. https://github.com/xesina/golang-echo-realworld-example-app, 2021.

[4] React redux realworld example app. https://github.com/gothinkster/react-redux-realworld-example-app, 2021.

[5] Realworld example apps. https://github.com/gothinkster/realworld, 2021.

[6] FIDO Alliance. Fido2: Webauthn & ctap. https://fidoalliance.org/fido2/.

[7] Automattic. Calypso. https://github.com/Automattic/wp-calypso, 2021.

[8] Dirk Balfanz, Alexei Czeskis, Jeff Hodges, J.C. Jones, Michael B. Jones, Akshay Kumar, Angelo Liao, Rolf Lindemann, and Emil Lundberg. Web authentication: An api for accessing public key credentials level 1. https://www.w3.org/TR/webauthn/#sctn-simple-txauth-extension, 2019.

[9] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. *IEEE Symp. Security and Privacy*, pages 9, 11, 2012.

[10] Kim Crawley. Web application firewalls explained: what is waf? https://cybersecurity.att.com/blogs/security-essentials/explain-how-a-web-application-firewall-works, 2020.

[11] Lucas Garron. Github supports web authentication (webauthn) for security keys. https://github.blog/2019-08-21-github-supports-webauthn-for-security-keys/, 2019.

[12] Gennie Gebhart. How to enable two-factor authentication on bank of america. https://www.eff.org/deeplinks/2016/12/how-enable-two-factor-authentication-bank-america, 2016.

[13] Google. Virtual authenticators tab. https://github.com/google/virtual-authenticators-tab.

[14] Krypt.co. Krypton. https://krypt.co/.

[15] Duo Labs. Webauthn library. https://github.com/duo-labs/webauthn, 2020.

[16] Juan Lang, Alexei Czeskis, Dirk Balfanz, Marius Schilder, and Sampath Srinivas. Security keys: Practical cryptographic secondfactors for the modern web. *Financial Cryptography and Data Security*, 20, 2016.

[17] Ledger. Ledger cryptocurrency wallet. https://www.ledger.com/.

[18] Nick Steele. Webauthn.io: A demo of the webauthn specification. https://webauthn.io/.

[19] Trezor. Trezor cryptocurrency wallet. https://trezor.io/.

[20] yubico. What is fido u2f? https://www.yubico.com/authentication-standards/fido-u2f/.

[21] yubico. The yubikey. https://www.yubico.com/products/.