WebAppAuth: An Architecture to Protect From Compromised First-Party Web Servers

Pascal Wichmann[1][a], Sam Ansari[1], Hannes Federrath[1] and Jens Lindemann[1][b]

[1] *Universität Hambrug*

pascal.wichmann@uni-hamburg.de, sam.ansari@studium.uni-hamburg.de, {hannes.federrath, jens.lindemann}@uni-hamburg.de

Keywords: Code Authenticity, Web Applications, Client-Side Security, Web Security.

Abstract: We present the WebAppAuth architecture for protecting client-side web applications even from attackers who fully control the web server. WebAppAuth signs all files sent to the client on a secure offline device or a hardware security module never accessible by the web server. Public keys are propagated through a key registry that is maintained by two independent key registration authorities, thus protecting users even on their first visit to the web application. Our threat model covers attackers who gain full control over the targeted domain and its DNS and DNSSEC configuration.

1 Introduction

Web applications can implement security mechanisms such as encryption or signature validation in client-side code. However, the architecture of the web relies on web servers to provide the client-side code, thus requiring trust into the authenticity of the code provided by the web server.

Solutions exist to protect the authenticity of web assets, such as Transport Layer Security (TLS) for verification of the server identity and ensuring the confidentiality of connections, or Subresource Integrity (SRI), which allows to pin external resources to fixed contents via hashes. However, most of these solutions do not consider compromised first-party web servers where an attacker is able to use TLS certificates valid for the targeted domain or modify resources served by the first-party server.

While such an attacker would have very strong capabilities, protecting from them is important for several types of applications that can still provide sufficient protection if the browser does not execute manipulated script assets. Examples are end-to-end-encrypted applications supposed to conceal contents from the web server where – without protection – such an attacker could turn off the end-to-end encryption, or web-based crypto currency wallets, where the attacker could extract the private keys. Thus, we propose WebAppAuth, an architecture that provides protection even if the first-party web server is compromised. To provide this protection, we rely on signing all assets using private keys that are stored and used at a location inaccessible to the web server, e. g., on a device never connected to the Internet.

We assume the party responsible for the application *("web application operator")* to be trustworthy. However, *any* servers involved in the operation of the application, including those of the first party, may be under full control of an attacker.

The remainder of this paper is structured as follows: In Section 2, we present our architecture, which we evaluate in Section 3. In Section 4, we discuss the practicability and limitations of our architecture. Section 5 presents related work on the authenticity of web application code. We conclude our work in Section 6.

2 Architecture

In this section, we describe the architecture of WebAppAuth. We start by discussing our threat model. Subsequently, we describe the components of our architecture, followed by the deployment and retrieval of web applications. Lastly, we describe operational tasks, such as key revocation.

[a] https://orcid.org/0000-0002-8969-4277

[b] https://orcid.org/0000-0003-0103-2461

This Paper was presented at SECRYPT 2023. The publisher's version is available at https://doi.org/10.5220/0012141700003555.

2.1 Threat Model

We assume a very strong attacker with full control over all servers of the targeted web application, including first-party servers. Hence, the attacker can also manipulate all communication between any web server of the attacked web application and its users. Alternatively, the attacker may manage to control the DNS and DNSSEC data of the targeted domain, obtain a valid TLS certificate for it, and redirect traffic to an own server.

WEBAPPAUTH relies on key registration authorities to manage published public key material. These authorities are assumed to be trustworthy and outside of the control of the attacker. The authorities follow the protocol and do not tamper with the integrity of the data they are responsible for.

2.2 Components of WEBAPPAUTH

Figure 1 gives an overview of the components of WEBAPPAUTH. The architecture relies on a web application key registry (Section 2.2.1), which is checked by the browser to establish whether a domain uses WEBAPPAUTH. The registry is administered and published by two key registration authorities (Section 2.2.2). To check a concrete web application, the browser further processes a web application manifest (Section 2.2.3) and individual file signatures (Section 2.2.4) published by the web server alongside the application.

2.2.1 Web Application Key Registry

The key registry contains a SHA-256 hash of the public key for all web applications that employ WEBAPPAUTH. Every registry entry is associated with exactly one such key. Applications are identified by their domain name. The registry entry defines the subdomains that use WEBAPPAUTH. The entry may be overwritten by a revocation message and can contain a flag that marks the entry for deletion at a specific time.

The domain name should be an eTLD+1 name. It is possible to deploy WEBAPPAUTH for a non-eTLD+1 subdomain. However, the set of subdomains covered by two entries may not overlap. This ensures a single definitive trust source for each application and prevents operators from keeping old keys in the registry.

All browsers that support WEBAPPAUTH keep a verified local copy of the key registry updated at least daily. This prevents downgrade attacks, i. e., attackers who manage to impersonate a web server cannot disable WEBAPPAUTH. On every update, the authorities provide deltas to every previous registry version of the past week. Consequently, if browsers have a current version of the registry that is less than a week old, they can retrieve a single delta file and the signature of the new version. The delta files can be applied in a deterministic way, i. e., always produce a consistent registry file.

Every version of the registry contains a timestamp that is covered by the signatures of the authorities. To allow public auditing of the registry, a log of all changes between every published version of the key registry is published using a Merkle hash tree equivalently to certificate transparency logs (Laurie et al., 2013).

In addition to the registry hosting provided by the authorities, a copy of the key registry including the signatures from the authorities is hosted by additional mirror servers, improving the availability of the registry.

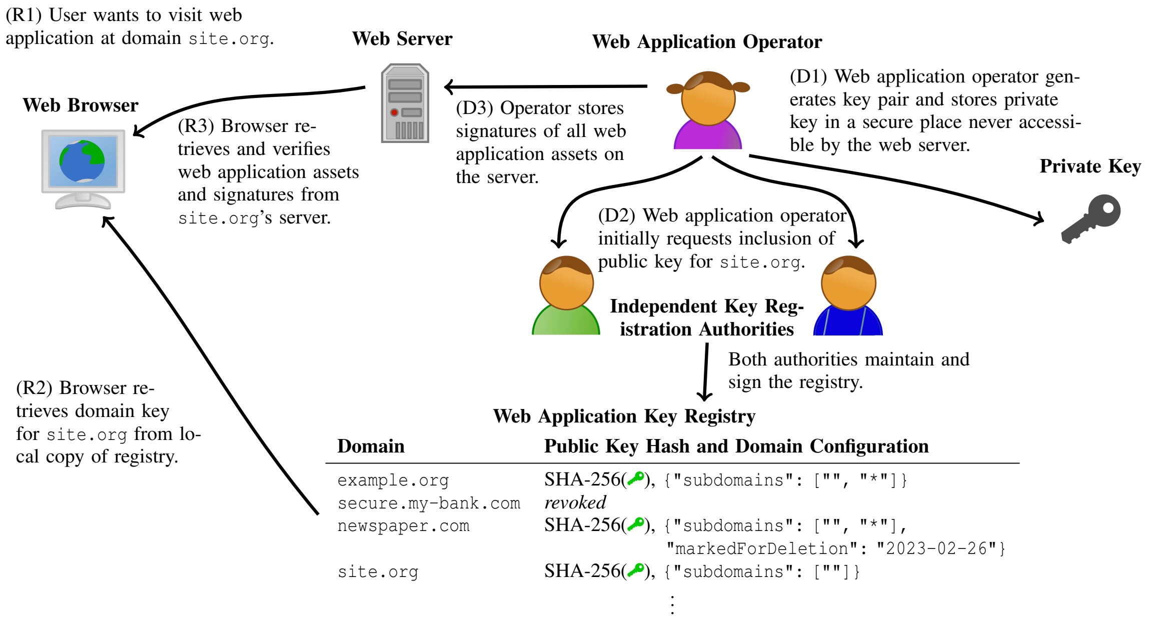
2.2.2 Key Registration Authorities

The key registration authorities are responsible for maintaining the key registry. To prevent an individual registration authority from manipulating, we rely on two independent authorities.

Both key registration authorities independently verify that requests to add a domain to the key registry are authentic, i. e., the requestor has control over the domain. Each authority sends a challenge that consists of a randomly generated string to the requestor who then deploys a DNS TXT record containing the string for the domain. Additionally, the TXT record contains a SHA-256 hash of the requested registry entry to prevent manipulations of the requests, e. g., by a man-in-the-middle attacker.

Requests must always be sent to both authorities. The authorities coordinate submission attempts and block requests that do not fulfill this requirement. To prevent malicious deployments when an attacker has temporarily gained control over a domain, an email is sent to the addresses `hostmaster`, `postmaster`, `security`, and `abuse` of the domain every hour throughout the 72 hours following the successful ownership verification. The high email frequency is chosen to mitigate attacks temporarily manipulating the DNS email configuration of the domain. After the 72 hours, the domain is added to the key registry unless disputed by the legitimate domain owner. Within the first 14 days after addition, an immediate removal can be requested by the legitimate domain owner after ownership validation (late dispute). Afterwards, the entry is permanent and the regular removal procedure (cf. Section 2.6) must be used for removal.

The authenticity of the key registry is verified us-


(R1) User wants to visit web application at domain site.org.
Web Server
Web Application Operator
(D1) Web application operator generates key pair and stores private key in a secure place never accessible by the web server.
Web Browser
(R3) Browser retrieves and verifies web application assets and signatures from site.org's server.
(D3) Operator stores signatures of all web application assets on the server.
Private Key
(D2) Web application operator initially requests inclusion of public key for site.org.
Independent Key Registration Authorities
Both authorities maintain and sign the registry.
(R2) Browser retrieves domain key for site.org from local copy of registry.
Web Application Key Registry

Domain	Public Key Hash and Domain Configuration
`example.org`	SHA-256(🔑), `{"subdomains": ["", "*"]}`
`secure.my-bank.com`	*revoked*
`newspaper.com`	SHA-256(🔑), `{"subdomains": ["", "*"],` `"markedForDeletion": "2023-02-26"}`
`site.org`	SHA-256(🔑), `{"subdomains": [""]}`
	⋮

Figure 1: Overview of the architecture of WEBAPPAUTH. The steps (D1) to (D3) are required to deploy the mechanism to a web application at a new domain `site.org`. The steps (R1) to (R3) are performed upon every retrieval of a web application using WEBAPPAUTH.

ing signatures from both authorities. Both have an identical copy of the key registry as they enforce that all modifications are registered with both authorities.

The public key of both authorities is included in the browser and used to verify the authenticity of the registry. Each authority is responsible for its private key's security and must store it on a hardware security module or a device not connected to the Internet.

The communication between the web application operators and the key registration authorities is protected through TLS. The TLS certificates of the authorities are signed with the private key used for the key registry for ensuring their authenticity, but use a separate key pair for the communication itself. This separate key pair cannot be used to sign registry versions, implying lower security requirements for the protection of the private keys. In particular, key registries can store their key pair for signing the registry on a dedicated secure device not connected to the Internet while still being able to deploy the separate communication key pair on their server connected to the Internet.

Both authorities use independent implementations of WEBAPPAUTH using different software stacks. This reduces the likelihood that both have identical vulnerabilities or bugs.

2.2.3 Web Application Manifest

Every web application that utilizes WEBAPPAUTH has to provide a manifest as a JSON file at the path `/.well-known/auth-protection/manifest.json`. A signature of the manifest created with the private key of the web application operator is provided and verified by the browser (cf. Section 2.2.4). If no valid manifest file exists or the signature verification fails, the browser refuses to load any part of the web application. If WEBAPPAUTH is enabled for more than one subdomain, each subdomain has to provide its own manifest file.

An example of a manifest file is shown in Listing 1. Every manifest contains the following information:

- The public key used for asset validation. Its hash has to match the entry in the key registry.
- The creation time of the manifest file. It is used to prevent a malicious rollback to a previous version.
- A list of file types excluded from the authenticity protection, i. e., these file types do not require signatures. This can be useful where files are generated automatically, e. g., images, documents or HTML pages. Signing such files would be impractical without allowing the web server to have direct access to the private key.

  WEBAPPAUTH suppresses the sniffing of (MIME) media types by the browser to prevent

```
{"publicKey": "d2ljaG1hbm5wYXNjYWwtcGF...",
 "time": "2022-12-19T13:39:52.362346",
 "excludeMediaTypes":
        ["application/json"],
 "excludePaths": ["/staging/"],
 "CSP": {"/": "default-src 'self';"}}
```

Listing 1: Example web application manifest file

possible exploitation (Barth et al., 2009). It enforces that only scripts can be executed.

- The `excludePath` option disables protection for specific paths. The browser does not verify signatures underneath these paths. However, it is not possible to load any files from excluded paths from origins that are not excluded. For example, if `/staging/` is excluded, a file at path `/app/foo.html` cannot load a resource `/staging/script.js` from an unprotected location.
- The Content Security Policy (CSP) that must be set for all protected resources. It must disallow the inclusion of scripts from third-party origins. Different CSPs can be set for different path prefixes. If a resource uses a CSP that does not match the configuration in the manifest, the request is blocked by WebAppAuth. The manifest must set a CSP for every path, i. e., requests for resources for which the manifest does not specify a value are blocked.

  The restriction of the CSP prevents attacks that exploit media types that are excluded from protection. For example, without a restrictive CSP and with the `text/html` media type excluded, an attacker could use inline scripts to bypass protection of script files.

2.2.4 File Signatures

For every asset of the web application that is not excluded through the manifest file, a signature is provided by the web server validating both the media type and the content. Two ways are supported to provide the signatures:

- Through an HTTP header `Asset-Signature` containing the base64-encoded signature value.
- As a separate file at the same path with an additional `.sig` extension. This does not require modifications to the web server configuration, as they can be deployed like regular web application files.

  If the web server supports it, this approach can be combined with HTTP/2 server push to provide the signature files before the browser explicitly requests them.

The browser always checks for the existence of the `Asset-Signature` HTTP header first. Only if the header is not present, it tries to retrieve the separate signature file.

2.3 Deployment for a New Web Application

To deploy the mechanism for a web application at a domain that has not previously been used with WebAppAuth, the following steps are performed (cf. Figure 1):

(D1) The web application operator generates an asymmetric key pair consisting of a public and a private key. The key generation is performed in a secure environment. The private key is always stored in a secure location inaccessible to the web server, e. g., on an offline device that is never connected to the Internet, or a hardware security device.

In addition, the application operator generates a revocation message that is signed with the private key. It can be used to revoke the key registry entry in case the private key is lost (cf. Section 2.7). A secure storage of the revocation message is essential to the availability of the domain, as any attacker gaining access to the revocation message can disable the web application.

(D2) The application operator sends a request for inclusion in the web application key registry to the authorities, consisting of the domain name, the SHA-256 hash of the public key, and the list of subdomains that should enforce WebAppAuth. Before the registries accept the request, both independently verify the domain ownership via DNS-based authentication (cf. Section 2.2.2).

The application operator also provides a contact email address and optionally a postal address. The email address should (but does not have to) belong to a separate domain that is independent from the registered domain. This contact information is used to inform the private key holder of security-relevant information about their domain, such as a removal request or a revocation. The authorities verify that they received identical contact information. The contact information is not published, however.

(D3) The web application operator signs the manifest and all assets of the web application and uploads the signatures to the web server. On every web application update, signatures for the new and changed assets are uploaded.

Deploying WebAppAuth for a domain is a permanent action. Consequently, domain owners must carefully consider whether to add their domain to the key registry. If the key is lost or it becomes inconvenient to continue the support of WEBAPPAUTH for a web application (e. g., due to changed application requirements), the protected domain cannot be further used. While in the latter case, it is possible to use the manifest to disable WEBAPPAUTH for most parts of the application, in the case of a lost key, it is not possible to perform *any* further modifications, including to the protected file types. However, WEBAPPAUTH provides a removal procedure (cf. Section 2.6) that allows to remove a domain from the key registry with a delay of two months.

2.4 Retrieval of a Protected Web Application

A browser with support for WEBAPPAUTH performs the following steps for every retrieval of a web application and any other file received from a web server (cf. Figure 1):

(R1) The browser receives the request to load a specific URL.

(R2) The browser uses its verified local copy of the key registry to check whether the respective domain uses WEBAPPAUTH. If no matching entry is found, the browser continues to handle the instruction as usual. If an entry comprises the domain and contains a revocation flag, the browser refuses the request. If a matching entry is found that marks the domain for deletion in the past, the entry is ignored and the browser continues to handle the user instruction as usual. Otherwise, the public key hash from the entry is used to verify the public key.

Next, the browser retrieves the manifest file for the web application from the `/.well-known/auth-protection.json` path of the domain the request is sent to and verifies its signature. To verify the signature, the browser first checks the `Asset-Signature` HTTP header of the response. If missing, the signature at the path with `.sig` suffix is requested. If no valid signature is found, the request is blocked.

The manifest is cached locally according to the cache configuration in the HTTP headers. If a cached version not yet expired exists, the browser does not request the manifest again. If the browser previously retrieved a manifest file from this domain, it verifies that the timestamp of the new manifest is more recent than the previous, blocking all requests to this domain otherwise.

(R3) Concurrently to the previous step, the browser requests the URL from the server. Unless the manifest excludes this media type or path, the browser verifies its signature using the public key from the manifest. The browser verifies the signature of the manifest as explained in the previous step. The last performed check verifies that the CSP of the response matches the one configured in the WEBAPPAUTH manifest for the respective path. If the CSP does not match, the request is blocked.

When a request is blocked, the user is warned. However, the user is *not* provided with the ability to suppress such errors.

2.5 Update of a Key Registry Entry

The list of included subdomains of an entry can be updated. The process is identical to the deployment of a new web application (cf. Section 2.3) but the requestor must use the existing key pair. After successful domain ownership verification and 72 hours dispute time, the existing entry is updated.

This process can only be used to *extend* the list of subdomains. It is not possible to remove any subdomains this way. Removing a subdomain is subject to the key registry entry removal process (cf. Section 2.6).

However, it is possible to *split* an entry, i. e. create a new, separate entry for a subdomain. To prevent an attacker from exploiting this for publishing a (compromised) key for a subdomain that would then be used instead of the legitimate key from the original entry, a waiting period of two months, similar to the removal of an entry (cf. Section 2.6) has to be observed. After this period has passed, the subdomain is removed from the original entry. Simultaneously, the requested new entry for this subdomain is published.

2.6 Removal of a Key Registry Entry

WEBAPPAUTH provides a removal procedure to allow domain owners to revert a deployment, for example because they do not want to use it any longer or the domain ownership has changed. However, the removal enforces a delay of two months before taking effect to provide legitimate domain owners a sufficient time to dispute an illegitimate removal request.

To initiate the removal of a domain, the domain owner sends a removal request to both key registration authorities. Both authorities independently verify the domain ownership of the requestor via DNS-based authentication analogously to the process of domain inclusion (cf. Section 2.2.2). If both authorities have verified the ownership of the requestor, they update the specific key registry entry to mark it for deletion after the delay of two months. On the marked date, the authorities completely remove the respective entry.

As an alternative to removing a full entry, subdomains can also be removed from an entry. This will leave the rest of an entry intact, covering the remaining subdomains. This is subject to the same process and delay as removing an entry altogether.

2.7 Revocation of a Key Registry Entry

A revocation procedure exists that allows to mark a key registry entry as revoked in case the private key of a domain is leaked. When the browsers have updated their registry copy to contain the revocation flag of the entry, they refuse any requests to its configured (sub)domains, effectively disabling the domain for all browsers supporting WEBAPPAUTH.

To perform the revocation, the operator sends their domain's signed revocation message they have prepared during key generation (cf. Section 2.3) to both key registries. The registries replace the domain configuration with a revoked flag and publish a new version of the key registry. The revocation procedure does not trigger a removal. If desired, the operator needs to follow the regular removal procedure (cf. Section 2.6) to fully remove the entry.

3 Evaluation

In this section, we evaluate the overhead of WEBAPPAUTH for users and servers. Browsers contain a copy of the key registry. Each entry has a size of approximately 150 B depending on the number of subdomains. Assuming 10 000 entries, this results in a total size of 1.5 MB stored locally. For comparison, the HSTS preload list in Chromium contained approximately 198 000 entries as of March 30, 2023. Due to the nature of WEBAPPAUTH, we assume a much lower number of deployments than HSTS preload, given primarily web applications with client-side security operations benefit from WEBAPPAUTH.

Browsers update the registry at least daily if the browser is used. As long as the last update is less than a week ago, updates transfer only the delta to the previous version. Depending on the number of changes, this adds a small amount of traffic for every daily update, e. g. just under 5 kB for 33 updates. If the browser has not been used for more than a week, this requires another full download of the registry, causing another 1.5 MB of traffic for a registry of 10 000 entries.

To retrieve a web application, browsers first check whether the key registry contains an entry for the domain. Only domains included produce further overhead: per protected domain, the browser retrieves the manifest file, causing on average 500 B traffic for the request-response pair. In addition, each request for a file that is not excluded from protection adds a signature that is less than 80 B long if retrieved as a header. If deployed as a separate file, a somewhat larger amount of traffic of less than 400 B is generated, as a full request-response pair is needed. If we assume a web application requires retrieval of 50 protected files, this causes a total of 4 kB (header) or 20 kB (separate signature files) of additional traffic.

No additional delay is caused by checking the manifest, as it is retrieved concurrently to the other files. Checking the signatures also causes no additional delay if they are delivered in an HTTP header. If the server does not deliver signatures through the header, a signature file will be retrieved after the corresponding file, however. This induces a delay of one round-trip time.

4 Discussion

In this section, we discuss the practicability and limitations of WEBAPPAUTH.

Relevance of the Threat Model Our threat model considers a very strong attacker who has full control over all servers of a web application. Consequently, the attacker can manipulate all server-side processing of the application and access all data. However, client-side code can use cryptography to conceal plaintext data from the web server and verify data authenticity. Thus, the protection of client-side code of such applications is essential even when the web servers are malicious.

Number of Key Registration Authorities The number of authorities is a trade-off between trustworthiness of the key registry (more independent parties and thus higher trust) and overhead (more parties and thus more peers to coordinate with). WEBAPPAUTH uses two to keep the coordination overhead acceptable while still preventing a single party from manipulating the registry.

Choice of Key Registration Authorities The authorities need to be trustworthy. Possible options for this role may be browser vendors or well-known nonprofit associations.

Attacks Prior to Initial WebAppAuth Deployment We assume that the initial deployment of WebAppAuth, i. e., the addition of the domain to the key registry, happens *before* any attack attempts, for example right at the initial deployment of the web application. That is, WebAppAuth cannot provide its protection if attacks occur before the deployment. Such attacks may include attacker-initiated attempts to register a new domain that did not exist previously, e. g., for phishing.

Attackers With Full Control of the Domain Attackers fully controlling DNS(SEC) data of a domain can break its availability. However, confidentiality and authenticity of data remain protected on the client side. An attacker must control the domain for more than two months to undergo the complete removal procedure in order to impair the confidentiality and authenticity of the client-side protected data.

Attacking Browsers Attackers not covered by our threat model may attempt to attack browsers to disable WebAppAuth or to manipulate the public keys of the authorities. However, the capability to manipulate browsers also gives the attacker the ability to manipulate the client-side web application execution independently of any authenticity protection.

Application Vulnerabilities While WebAppAuth can protect the authenticity of the client-side code, it cannot protect from vulnerabilities that are present in the authentic code provided by the web application operator.

Risk of Temporary Domain Loss If operators of a WebAppAuth-protected web application lose their private key, they can no longer deploy updates to the client-side code, making the domain practically unusable for web applications. This risk cannot be avoided without losing protection from attackers who manage to temporarily control the attacked domain, as such attackers temporarily mock the role of the operator. A removal procedure is available to re-gain control to such a domain after a delay of two months.

Single Key per Entry Our architecture does not allow to use more than one key pair per registry entry. Consequently, domains for example maintained by larger organizations need to sign their assets in a secure way without providing additional keys for organizational units. This way, the responsibility for the key security and trust management lies with a central entity within the organization. On a subdomain level, separate non-overlapping key registry entries can be used with individual keys to split responsibility within the organization.

Required Changes for Deployment To utilize the protections of WebAppAuth, some changes are required on the user and server side. Firstly, the browser needs to support the mechanism. This support can be added as a native browser functionality in the future. Until then, a browser extension can be used to add WebAppAuth support to the browser, which does not require implementation by browser vendors. Secondly, the web server needs to provide the signatures for the protected assets. For file-based delivery, the web application operator only needs to upload the signature files to the server. For the more efficient header-based delivery, the web server software needs to be configured to read the signatures from their corresponding files and add them to the header. Depending on the used software, this may require changes to its code. However, if a servers hosts only a small number of infrequently updated files, it is sufficient to statically configure the signature headers for the individual paths, which is possible in all popular web server software without further changes.

5 Related Work

In this section, we discuss prior work on web authenticity.

Established Web Security Mechanisms Several security mechanisms are supported in all major browsers. TLS (Rescorla, 2018) can protect the confidentiality and authenticity of the communication between the web server and the browser using transport encryption and validation of server certificates. HTTP Strict Transport Security (HSTS) (Hodges et al., 2012) allows to enforce the use of TLS for a domain and is activated through an HTTP header. With HSTS preloading (Chromium Project, 2023), domains can be added to a list in browsers to protect the first visit.

SRI (Akhawe et al., 2016) allows to protect included subresources by specifying a cryptographic hash of their contents. If a resources' hash does not match, the browser refuses to load it. SRI is intended to protect from third-party servers hosting referenced assets.

Signature-Based Authenticity Sutter et al. (2021) consider a compromised TLS connection between a browser and a web server, e. g., a man-in-the-middle attack where the attacker has obtained a valid TLS certificate. The authors use service workers to verify the integrity of all HTTP responses from the web server, allowing deployment in web applications without changes to the browser. All responses from the web application contain a signature in an HTTP header which is verified by the service worker. The signatures are generated on the web server or optionally on a separate server. The client must have visited the domain before for the mechanism to provide protection. If an attacker manages to compromise the first connection of a user, they can suppress the corresponding header to prevent the browser from enforcing it, voiding all protection.

Levy et al. (2016) propose Stickler as an alternative to SRI that can be used to provide authenticity to assets that are served through a content delivery network (CDN). The initial request is sent to the first-party web server, returning a script that contains the Stickler code and the public key of the first party. This script retrieves a signed manifest file from the CDN, allowing to retrieve and verify all further assets from the CDN. Mignerey et al. (2020) propose another browser extension that uses signatures to verify the authenticity of assets. Their solution assumes the public key to be shared on an out-of-band channel before the users visit the web application.

Cavage and Sporny (2019) propose a standardization for end-to-end signing HTTP messages, i. e., preventing undetected tampering of messages during their transmission. Their draft adds a `Signature` header specifying the key used for signing as well as the resulting signature. The other party can then verify the signature. However, Cavage and Sporny do not cover the key exchange and key authenticity verification processes.

Transparency Logging WAIT (Meißner et al., 2021) provides transparency of client-side code through a verifiable public log in which the web application operators publish their code modifications. This prevents web servers from tampering with the client-side code sent to individual clients without publishing the specific modification to the log.

Salvador et al. (2018) focus on digital elections and propose a solution that monitors changes of client-side web application code introduced by the operators. They perform periodic scans of the web application to analyze its code and prevent web server operators from delivering malicious code to a wide range of their users. However, this monitoring-based approach cannot prevent targeted attacks, e. g., if the server operator provides malicious code only to users at specific IP addresses.

Code Reviews Cap and Leiding (2018) propose an architecture that relies on manual reviews of web application code. All reviews are published in a public log. A browser extension verifies whether a valid review for a retrieved version of a web application exists in this log. It then provides the user with information about the security of the application based on this.

Jansen et al. (2017) implement a browser extension that verifies client-side web application code that may be served from untrustworthy sources. They consider a secure multi-party computation scenario where multiple mutually distrusting parties want to perform computations together through a web application. All code is audited by multiple trusted parties before it is executed on the clients.

Additional Trusted Servers The required trust can be split across additional parties or trusted servers. Karapanos et al. (2016) propose Verena, which adds an additional server to the deployment of a web application, allowing clients to receive cryptographic proofs of the application integrity. All requests requiring end-to-end authenticity are passed through this additional Verena server which generates a proof verifiable by the browser. The developer defines which application parts require such proofs. Verena allows to protect from attackers who compromise the application server and the database server as long as the Verena server is not compromised.

Popa et al. (2014) consider a threat model in which the attacker has full access to the web server. They provide a web application framework that provides client-side encryption, allowing keyword searches on the server without decryption. Their architecture assumes that attackers are unable to tamper with the initial user's request to a first origin, while manipulations to all communication with a second origin can be detected.

DNS-based Authenticity Protection Multiple solutions rely on DNS records to provide protection. The SecureBrowse project (Chuk and Shapiro, 2019) provides protection if the connection between the client and the web server is compromised or a third-party web server is malicious. It relies on DNS TXT records that contain Subresource Integrity (SRI) hashes of the web resources, including all first-party resources and the main page. To ensure the authenticity of the DNS records, DNS-based Authentication of Named Entities (DANE) is used. However, unlike WEBAPPAUTH, SecureBrowse does not require the key to be kept outside of the web server's control and thus does not consider a compromised first-party web server. Beyond DANE, SecureBrowse does not protect from a compromised DNS, for example if an attacker manages to remove or replace an existing DNSSEC key and set an own DNS nameserver for the domain. Furthermore, unlike WEBAPPAUTH, SecureBrowse is unable to protect from attackers who are able to control the DNS for multiple days, including the capability to manipulate DNSSEC keys. In addition, SecureBrowse does not allow to exclude assets from protection, which makes it unsuitable in some scenarios that are covered by WEBAPPAUTH, such as dynamic generation of HTML or media files.

Varshney and Shah (2021) analyze the threat of client-side manipulation of web application code through browser extensions. They propose a DNS-based security policy framework that enables the browser to detect such manipulation. Their architecture provides hashes of important pages of the web application via DNS TXT records, which the browser can compare with the hashes of the actual pages.

6 Conclusion

In this paper, we proposed WEBAPPAUTH that can protect from very strong attackers who have full control over all web servers and the domain DNS. It relies on signing client-side code with a private cryptographic key, which the web application operator must store in a secure location, such as on air-gapped devices. Two independent key registration authorities verify domain ownership and maintain a public registry containing the public keys of all domains. These public keys can then be used by clients to verify the authenticity of the web application. WEBAPPAUTH requires the transmission of only a relatively low amount of extra data and can be deployed in a way that does not cause additional delays when loading a web application. It is robust to attackers fully controlling an attacked domain for a limited time.

As future work, we intend to research server-side authenticity within our threat model, e. g., using trusted hardware.

References

Akhawe, Devdatta et al. (2016). *Subresource Integrity*. W3C Recommendation. URL: https://www.w3.org/TR/2016/REC-SRI-20160623/.

Barth, Adam et al. (2009). *Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves*. In: *30th IEEE S&P*, pp. 360–371.

Cap, Clemens and Benjamin Leiding (2018). *Ensuring resource trust and integrity in web browsers using blockchain technology*. In: *International Conference on Advanced Information Systems Engineering*, pp. 115–125.

Cavage, Mark and Manu Sporny (2019). *Signing HTTP Messages*. Internet-Draft draft-cavage-http-signatures-12. URL: https://www.ietf.org/archive/id/draft-cavage-http-signatures-12.txt.

Chromium Project (2023). *HSTS Preload List Submission*. Accessed May 5th, 2023. URL: https://hstspreload.org.

Chuk, Brian and Paul Shapiro (2019). *SecureBrowse Project*. URL: https://gitlab.com/securebrowse/securebrowse/-/wikis/The-SecureBrowse-RFC.

Hodges, J. et al. (2012). *HTTP Strict Transport Security (HSTS)*. RFC 6797.

Jansen, Frederick et al. (2017). *Brief Announcement: Federated Code Auditing and Delivery for MPC*. In: *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS*, pp. 298–302.

Karapanos, Nikolaos et al. (2016). *Verena: End-to-End Integrity Protection for Web Applications*. In: *37th IEEE S&P*, pp. 895–913.

Laurie, B. et al. (2013). *Certificate Transparency*. RFC 6962.

Levy, Amit et al. (2016). *Stickler: Defending against Malicious Content Distribution Networks in an Unmodified Browser*. In: *IEEE Security & Privacy* 14.2, pp. 22–28.

Meißner, Dominik et al. (2021). *WAIT: protecting the integrity of web applications with binary-equivalent transparency*. In: *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing*, pp. 1950–1953.

Mignerey, Josselin et al. (2020). *Ensuring the Integrity of Outsourced Web Scripts*. In: *17th International Joint Conference on e-Business and Telecommunications, ICETE 2020 - Volume 2: SECRYPT*, pp. 155–166.

Popa, Raluca Ada et al. (2014). *Building Web Applications on Top of Encrypted Data Using Mylar*. In: *11th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, pp. 157–172.

Rescorla, E. (2018). *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446.

Salvador, David et al. (2018). *wraudit: A Tool to Transparently Monitor Web Resources' Integrity*. In: *Mining Intelligence and Knowledge Exploration - 6th International Conference, MIKE*, pp. 239–247.

Sutter, Thomas et al. (2021). *Web Content Signing with Service Workers*. In: *arXiv:2105.05551*.

Varshney, Gaurav and Naman Shah (2021). *A DNS Security Policy for Timely Detection of Malicious Modification on Webpages*. In: *28th International Conference on Telecommunications (ICT)*.