Ensuring the Integrity of Outsourced Web Scripts

Josselin Mignerey, Cyrille Mucchietto and Jean-Baptiste Orfila
WALLIX, Paris, France

Keywords: Data Integrity, Network Security, Security and Privacy in the Cloud, Security and Privacy in Web Services, Security Protocols.

Abstract: Dynamic web browsing, supported by web scripting languages such as JavaScript, has quickly conquered the Internet. In spite of the obvious advantages they offer, they have also opened many security flaws for the user browsing. The browser starts by retrieving some external scripts, potentially distributed over many servers. In terms of security, this process is extremely sensitive, therefore many solutions have been introduced to secure web browsing. Unfortunately, they mostly rely on server side actions. Hence, a malicious server is able to compromise the client by modifying the security policy and the scripts sent. We propose an efficient solution, which does not require any trust in the servers, to ensure the integrity of distributed web scripts. Our protocols rely on simple cryptographic tools, such as digital signature schemes and hash functions. In the end, we provide a proven secure, user-friendly and easy-to-deploy solution which only adds a small latency in the end-user browsing.

1 INTRODUCTION

Over the last decades, web content has become more and more dynamic. In practice, dynamic web pages are provided to the client by scripts, written in Javascript for 95% of them (W3C, 2018). They provide many kind of services to the end-users, but are also used to track and advertise them. The browser is in charge of executing those scripts generally located between the HTML tags `<script>...</script>`. Two main methods exist to retrieve scripts: either they are directly included into the web server response or the browser has to make additional requests. In the first method, called *inline scripting*, a script is represented as a plaintext program between the tags. Whereas the second method only provides the browser with a pointer materialized as a URL in order to retrieve the scripts. The external method is massively implemented since it facilitates the use of external libraries such as jQuery (De Volder, 2006) and allows cache management. In order to avoid end-user actions, the libraries are retrieved on-demand: when the user requests a web page including a script requiring a library, the latter is also requested. In practice, these libraries are widely used: more than 87% of the Alexa top websites used at least one of them (Lauinger et al., 2017). Consequently, the library distribution is performed by dedicated servers, called *Content Delivery Networks* (CDN) (Farber et al., 2003), and possibly by some cloud services. The purpose of CDNs is twofold: to limit the network stream of the original server by hosting libraries and to ensure the versioning of these libraries (even if using outdated libraries is still a problem, see (Lauinger et al., 2017) again).

In terms of security, web scripts have opened many flaws (Bielova, 2013). Among the previous methods, inline scripts are however safer from the client viewpoint. Assuming that there is a secure connection between the client and the server (e.g., with HTTPS), only the scripts directly sent by the server are executed on the client side. Obviously, this also supposes that the client trusts the server enough to execute the aforementioned scripts. In the case of external scripts, either the *URLs* indicate the same server or a different one. The second method has many security flaws (Nikiforakis et al., 2012): neither of the end-user privacy nor the script integrity is guaranteed since the third party is allowed to inject malicious code. A recent example of code injection has been realized using end-user resources to mine cryptocurrencies (Eskandari et al., 2018). To counteract the numerous security issues brought by malicious script inclusions, many solutions have been deployed (see (Nadji et al., 2009) for a more exhaustive listing).

One of them is the *Same-Origin Policy* (SOP),

Mignerey, J., Mucchietto, C. and Orfila, J.
Ensuring the Integrity of Outsourced Web Scripts.
DOI: 10.5220/0009829401550166
In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) - SECRYPT*, pages 155-166
ISBN: 978-989-758-446-6

which is a kind of access control policy for scripts. This checks if the origin (defined as a protocol, a domain and a port) of the scripts is the same as the original server. In theory, all three of them must verify the equality with the web server. This method only provides a partial protection, and some breaches have been found (Saiedian and Broyle, 2011). In practice, the origin is only partially verified so that the port is deduced from the protocols (Ruohonen et al., 2018). Moreover, cross-origin embedding is, despite the SOP, often allowed (Ruderman, 2018). SOP can also be relaxed through the *Cross-Origin Resource Sharing* (CORS) *via* a whitelisting of authorized cross-origin resources declared in the headers of the original web pages. This method has also recently been attacked (Chen et al., 2018). Another widely deployed counter-measure, called *Content Security Policy* (CSP) (West, 2018), provides to the web administrator a way to define a security policy on the resources that the end-user is allowed to retrieve or execute. In practice, the CSP is represented as a set of rules located in the headers of the HTTP response. The *SubResourceIntegrity* (SRI) (Weinberger et al., 2016) mechanism allows the client to verify the script integrity beforehand. This works as follow: along with the received scripts, a list of hashes is associated. Before executing a script, the browser computes the hash of a script, and verifies that the obtained digest matches the given one. This solution prevents against eventual script modifications by an external resource (such as a CDN), but not against the server itself. This clearly leads to a security flaw.

1.1 Contributions

Previous countermeasures are mainly rules set by the administrator on the server-side. A client such as the browser must follow these rules to block potential attacks. Nevertheless, if the server is corrupt and the rules are removed then the client won't be able to block any potential attacks on his own. This raises two issues: firstly, the current trend is towards decentralizing the script sources so that server-based solutions offer security guarantees inversely proportional to the deployment difficulties. Secondly, this assumes that the server is trusted. Otherwise, a malicious server has then the ability to modify the scripts sent to the client, to retrieve private information from it or to reduce the security policy from the initial configuration. We propose a different approach: here, the web server and potential external resources such as *CDNs* are seen as untrusted ways of storing resources. We choose to differentiate the web storage from the script editor i.e., the entity which develops the program. The latter goal is to put these scripts online, so that end-users retrieve and execute them into their browser. The scripts can be stored on a classic web server potentially administrated by the editor itself, but they can especially be stored into an external web server, or a cloud. The editor has no control on the content that the web server really distributes.

As a concrete use case, one can imagine a security software editor whose clients need to update their configuration. The editor provides some JavaScript applications that should be retrieved by the client. Using cloud-based services to ensure the availability of these scripts, neither the client nor the editor can be sure about their integrity. This is a concern for both of them: the former risks a potential data leakage, whereas the latter runs the risk of ruining its reputation. This example is inspired by the request of *ProtonMail* (Team, 2019): *"we are closely following the work being done in the web standards community to introduce some form of code signing for web"*. From this scenario, we extract the following problem, that has also been highlighted in (Ruohonen et al., 2018): *How can we ensure the integrity of outsourced (web) scripts without trusting the delivering servers ?*

We address this question by proposing protocols to ensure the integrity of outsourced scripts, in particular the JavaScript ones. Unlike existing solutions, our goal is to protect the client from a malicious server, which holds the web page including the scripts. We propose an externalized signature-based solution ensuring the integrity of scripts with respect to the initial developer code publication. Moreover, our solution also plays a part in the attacks against the JavaScript versions (NPM) (Lauinger et al., 2017): we let the developers choose the library version that should (and will) be executed by the client. Our solution proposes a way to protect the user from a website that is running on an untrusted provider. It may seem unnatural to run a website on such host but it can respond to specific needs or scenarios. Our main focus is to ensure and prove that a website, providing cryptographic services (such as mail encryption or data storage encryption) has not been compromised. Then the client's secrets are not leaked since we provide a possibility for the client to verify that all scripts came from a trusted source. When the developer has updated the scripts, the client can also be notified to eventually audit them.

1.2 Related Work

The problem of ensuring the integrity of web scripts has been addressed several times. Historically, BEEP (Jim et al., 2007) has been one the first.

Their proposal is a kind of CSP, with a client-side whitelisting approach. In the same vein, fine-grained client-side policy have been studied, for example in Conscript (Meyerovich and Livshits, 2010). In (Mitropoulos et al., 2016), authors propose a fingerprint-based solution as a countermeasure against XSS attacks. Their approach consists in checking the fingerprints for all JavaScripts that might be executed by the browser. Unfortunately, none of these approaches remain secure assuming that the server is malicious. A closer solution which also relies on external signatures has been developped by Mozilla (Mozilla, 2008). The idea is to use a signed JAR file, which contains the entire HTML structure along with the scripts references. Besides being related to a specific browser, this solution has been abandoned due to the risks associated to the JAR files (Mozilla, 2019). We overcome this issue by providing an astute use of hash lists, so that the integrity proof is small enough to not require any compression without degrading performances. In (Soni et al., 2015), the authors propose a solution based on signatures and some JavaScript specific isomorphisms defined to reduce the number of signature updates. However, their proposal is server-based, centralized and requires trust on first use. In (West, 2019), their idea is to replace the digest field from SRI solution by a digital signature by the web server. Then the client checks the signature using the server public key before executing the scripts. From this work, which seems to be close to our solution, we insist on the fact that the two security models differ: in our case, adding a signature into the SRI field does not provide any security enhancement if the web server is malicious. For instance, it is sufficient for the server to simply updates the original field containing the script signature and add a malicious one computed with the server private key. Another example consists in changing the headers used to define the security policy: the server removes the one indicating that the client must verify signatures beforehand. In our case, by addressing a malicious server, we handle both of these attacks. In (Nakhaei et al., 2018), the author presents a method in which scripts are first viewed as plaintexts containing signatures. Specific JavaScript files act as proxies to retrieve other subresources and check their signature. If the signature is valid, the original plaintext is *converted* to a JavaScript object and then loaded. Their approach eliminates the need for modification in web servers or browsers. As before, if the initial web server is malicious, the proxy script can be modified to bypass signature verification in order to allow malicious content to be executed by the browser.

1.3 Organisation

We start with a general description of our solution and its security model in Section 2. Then in Section 3, we present the protocols, in the form of algorithms. In the following section (Section 4), we formally define the security model and guarantees that these protocols provide. Next, (in Section 5), we detail an implementation of a plugin dedicated to the integrity of JavaScripts based on our solution. Finally, we conclude and present some future works in Section 6.

2 OVERVIEW AND DEFINITIONS

2.1 Preliminaries and Notations

We denote a cryptographic hash function as $\mathsf{H} : \{0,1\}^* \to \{0,1\}^l$ with $l \in \mathbb{N}$ the size of the digest. The concatenation of two messages m_1, m_2 is denoted $m_1||m_2$. The cardinal of a set A is denoted $\#A$. The security parameter is written 1^k. We denote with ε a negligible function. Next, we recall the definitions of the digital signature schemes and hash functions.

Definition 2.1 (Digital Signature Scheme)**.** *Let* $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verif})$ *be a digital signature scheme, composed with three probabilistic polynomial time (p.p.t.) algorithms such that:*
$\underline{\mathsf{KeyGen}(1^k)} \to (\mathsf{pk}, \mathsf{sk})$ *with* pk *a public key and* sk *the associated private key ;*
$\underline{\mathsf{Sign}(\mathsf{sk}, M)} \to \sigma$*, with* sk *the private key and* $M \in \{0,1\}^*$ *a message ;*
$\underline{\mathsf{Verif}(\mathsf{pk}, M, \sigma)} \to b \in \{0,1\}$*.* σ *is a valid signature of* M *with* sk *if* $\mathsf{Verif}(\mathsf{pk}, M, \sigma) = 1$*.*
A signature scheme is correct if $\forall k \in \mathbb{N}, \forall M \in \{0,1\}^*$,

$$Pr\left[\begin{matrix}(\mathsf{pk},\mathsf{sk}) \overset{\$}{\leftarrow} \mathsf{KeyGen}(1^k) \\ \mathsf{Verif}(\mathsf{pk}, M, \mathsf{Sign}(\mathsf{sk}, M)) = 1.\end{matrix}\right] = 1$$

We recall the security definition in the case of an existential forgery under the chosen message attack.

Definition 2.2 (Security of digital signature schemes)**.** *Let* $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verif})$ *be a digital signature scheme. An adversary* $\mathcal{A}$ *is a p.p.t. algorithm that is given as input a public key* pk *and an access to a signing oracle* $O_\Sigma(\mathsf{sk}, \cdot)$*, where* $(\mathsf{pk}, \mathsf{sk}) = \mathsf{KeyGen}(1^k)$*. The oracle takes as input a message* M *and returns a signature* $\sigma = \mathsf{Sign}(\mathsf{sk}, M)$*. The adversary is able to ask a signature for the messages of its choice, and eventually outputs a forgery* (M, σ)*. The advantage of* $\mathcal{A}$ *against the signature scheme, denoted as* $\mathbf{Adv}_{\mathcal{A}}^{\mathrm{UF-CMA}}(1^k)$ *is the probability that the adversary outputs a valid pair* (M^*, σ^*) *such that the signature* *of the message M^* has not been previously requested. We say that a signature scheme Σ is secure against existential forgery under adaptive chosen message attacks if the advantage for any $\mathcal{A}$ is negligible with respect to the security parameter k.*

Definition 2.3 (Second pre-image resistance of a hash function H (adapted from (Rogaway and Shrimpton, 2004))**.** *Let* $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^l$ *with* $l \in \mathbb{N}, l > 0$ *be a hash function.* H *is second pre-image resistant if*

$$Pr\left[\begin{matrix}(M) \xleftarrow{\$} \{0,1\}^*, M' \xleftarrow{\$} \mathcal{A} : \\ (M \neq M') \wedge (H(M) = H(M'))\end{matrix}\right] \leq \varepsilon$$

The advantage for an adversary $\mathcal{A}$ to find a second pre-image attacks is denoted $\mathbf{Adv}_{\mathcal{A}}^{2^{\text{nd}}\text{Pre}}()$.

2.2 Entities and Security Model

We propose a tripartite model composed of a client $\mathcal{C}$, a server $\mathcal{S}$ and the program editor $\mathcal{PE}$. In real-life, the client is represented by the end-user's browser. The latter communicates with the web server and potentially a small number of *CDNs* (around one or two). We represent all of these as a generic server. Finally, the program editor is the direct provider of the scripts, which deploys them *via* the server services.

The idea is to capture the cases in which attacks might be induced because of a bad script retrieval i.e., the scripts executed by the client's browser differ from the initial developer will. Two points are then particularly sensitive: first, the script source code has been altered. This obviously means that at least one part of the program does not match the initial source, but it could also mean that some of the libraries are not exactly the expected ones. The second point concerns the security policy, that could be altered when sent to the client. In this case, external sources or inline scripts might be added by a malicious server.

The security model stems from these observations. First, *the client is trusted*: in our model, it is the attacks target. However, we make some assumptions on its behavior in order to free ourselves from some collateral attacks. We suppose that its browser is safe i.e., it implements the common security requirements and is up-to-date. In particular, we require that the browser is able to deploy a security policy from the headers of an HTTP response, that it is able to check digital signatures and computes cryptographic hashes. Currently, this means that the browser could verify these assumptions if it implements the *CSP* and *SRI* mechanisms correctly. *The program editor is trusted*: in the sense that the provided scripts are not supposed to endanger the client security. Then, attacks related to the privacy of the client are not considered as attacks: if the client has chosen to execute the program for this editor, then we consider that the client agreed on giving its data. *The server is malicious* i.e., it might modify the web pages before sending them to the client. In particular, it has the possibility to alter the headers of its HTTP responses or the script content in order to damage the client security or privacy. As previously precised, the server might also represent also the set of entities delivering sources related content. Considering a malicious server encompasses the case where several servers collude, since they are viewed as one malicious entity (i.e., a *n* among *n* security). Thus, CDNs are not trusted to correctly deliver libraries and their behavior is associated with the server attacks. Moreover, considering a malicious server captures also potential attacks by an external adversary, since a server can be seen as an external adversary with additional possibilities.

2.3 Overview

When the browser needs to display dynamic pages, first it analyses the rules provided by the web server response. Generally, they are described with a list of headers, in which belongs the security policy. Then, we represent the security policy as a set of headers, denoted $\{\mathsf{hd}\}$. Next, the browser retrieves the script sources accordingly to the security policy. The complete program might require several scripts (such as external libraries), potentially downloaded by the browser from different providers. In the end, we only have interest into these scripts, since the providers should not have any impact on the security. Thus, we model these scripts with a simple set, denoted $\{\mathsf{script}\}$. Each element $\mathsf{script}_i \in \{\mathsf{script}\}$ refers to a finite set of instructions belonging to a common specific language, such as JavaScript. We abstract a complete web page as the concatenation of the set of headers and scripts i.e., $WP = \{\mathsf{hd}\}||\{\mathsf{script}\}$. In order to ensure that the policy and the scripts have not been modified by the server, we propose protocols divided into two parts. The first one is about the generation of the reference file labeled *IntFile*, for integrity file. Namely, its main property is to be unalterable. The second is about the comparison between the *IntFile* and the actual web page sent by the untrusted server. In the Figure 1, an overview of a protocol execution is schemed. The main idea is to externalize the signature of both the security policy and the scripts. This leads to an integrity file (*IntFile*), which is then published by the developer on any outsourced web services. The generation of this *IntFile* only requires that the developer signs its program and then uploads it. No more action or trust towards the outsourced service are required from its side. At this point, the client retrieves the web page along with the *IntFile*. In practice, the manner of retrieving this file is decided by the client: it could get it from the same web server or on any services of its choice. Next, the client checks that the web page is indeed in accordance with the integrity file. This solution is simple and efficient: it only requires one signature verification and the computation of some hashes from the client.

2.4 Key Management

The digital signature schemes are the core of the proposed solution. Although providing strong cryptographic guarantees, they assume an efficient and secure key management. On one hand, the signer uses its private key to sign. In general, it has generated the key pair himself, so this is not an issue. On the other hand, the verifier checks the signature using the public key of the signer, so that retrieving the key *a priori* is mandatory. We now detail some appropriate solutions addressing the key management problem. From now on, an underlying secure key management is assumed to be used.

Firstly, the program editor distributes, out of the band, its public key. Although simplistic, this solves the problem for the security software editor willing to distribute its script, as in our B2C use case. Secondly, public key infrastructures with X.509 certificates are also conceivable: the program editor requests a certificate to a well known certification authority (i.e., included into the trust anchor store of the browser). Then, the certificate is sent within the web pages. A more decentralized way will be to use the web-of-trust, namely Pretty Good Privacy based solutions. Each editor manages its own keys and gives its trust to other editors' public key. Obtaining a consensus seems however hardly achievable unless a private trust evaluation mechanism like (Dumas et al., 2017b) is also deployed. User-centric PKIs are also appropriate such as (Dumas et al., 2017a). The latter presents the advantage of easily getting a certificate for the program editor while only using existing tools. In the same vein, using the DNSSEC infrastructure (Arends et al., 2005), as described in (Jøsang and Dar, 2011) is also an alternative. Currently, some practical aspects have to be developed, in order to allow scripting languages such as JavaScript to make DNSSEC requests.

Others normalized solutions, like *Certificate Transparency* (Ryan, 2014) seem adequate by publicly showing the certificates related actions. In the same vein, plenty adaptations of the previous solutions using blockchains have been proposed: some examples of mixing PKIX and blockchains (Garay et al., 2018; Axon. and Goldsmith., 2017; Yakubov et al., 2018), or PGP and blockchains (Yakubov et al., 2018). We also refer to (Kubilay et al., 2019) for the approach mixing Certificate Transparency and Blockchains. Their main strength lies into their public checking of certificates and decentralized way of working. In the context of publishing the integrity file, public verifiability is also desirable. In our case, these solutions enable to add a versioning management to our schemes without any user-side change.

2.5 Definitions

We start by defining generic schemes called *Program Externalizer*. This notion captures the will of distributing program via an insecure platform and so is applicable to our use case in a web context.

Definition 2.4. *A* ***Program Externalizer Scheme****(PES) is composed of three algorithms:*

- $\mathsf{ParamGen}(1^k) \to (\mathsf{sp}, \mathsf{pp})$*: From the security parameter* 1^k*, it outputs the public* (pp) *and private (secret) parameters* (sp)*.*
- $\mathsf{IntFileGen}(\mathsf{sp}, Policy) \to IF$*: From the private parameters* sp *and a security policy Policy, it outputs the integrity file IF containing the security policy.*
- $\mathsf{Match}(\mathsf{pp}, IF, In) \to \{0,1\}$*: From the public parameters, the integrity file IF and the input In, it outputs* 1 *if the latter matches the security policy of IF,* 0 *otherwise.*

Definition 2.5 (Security of Program Externalizer Scheme)**.** *Let* $\mathsf{PES} = (\mathsf{ParamGen}, \mathsf{IntFileGen}, \mathsf{Match})$ *be a program externalizer scheme. The challenger starts by sending the public parameters* pp*. Then it chooses a Policy, generates a valid IntFile and sends it to the adversary* $\mathcal{A}$*.* $\mathcal{A}$ *has the possibility to requests the challenger about the validity of* (In^*, IF^*)*, so that the challenger computes* $\mathsf{Match}(\mathsf{pp}, In^*, IF^*)$*, and returns the results to* $\mathcal{A}$*.* $\mathcal{A}$ *wins the game whenever it outputs a valid forgery* (In^*, IF^*)*, defined as: either* IF^* *does not correspond to the original policy or* In^* *is not valid, and* $\mathsf{Match}(\mathsf{pp}, In^*, IF^*) = 1$*. The outputs of the game is the value of* Match *on the adversary forgery, denoted* $\mathbf{Game}_{\mathcal{A}}^{\mathsf{PES}} = b \in \{0,1\}$*. The advantage of* $\mathcal{A}$ *is written:*

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{PES}}(1^k) = \Pr\left[\mathbf{Game}_{\mathcal{A}}^{\mathsf{PES}} = 1\right]$$

We say that a PES *scheme is secured if the advantage of the adversary is negligible w.r.t. the security parameter* 1^k*.*

This security models captures the idea that the attacker is successful if it is able to distribute a different input file, but still considered as valid. In this model,

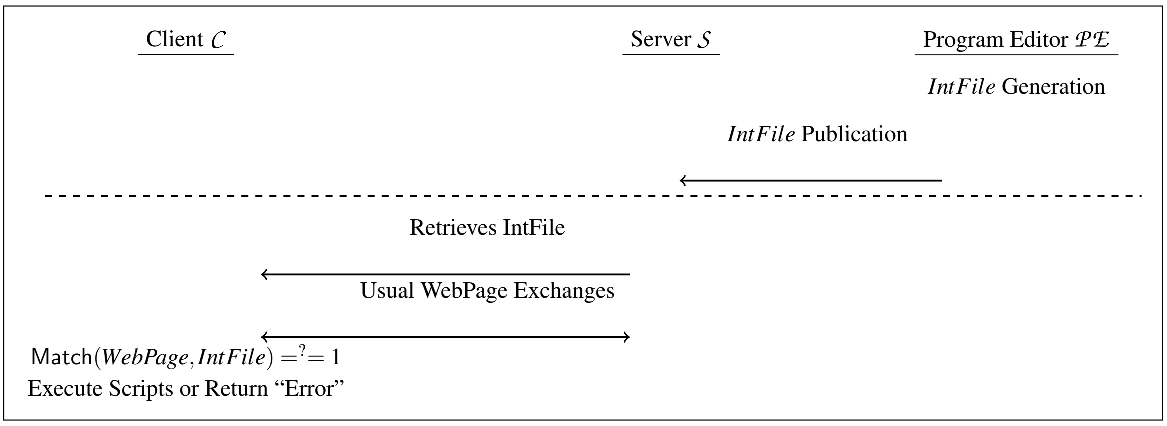

Figure 1: Overview of an execution of the Outsourced Script Integrity protocols.

we assume that only one valid IntFile is available. Versioning of IntFile is discussed in 2.4.

3 THE OUTSOURCED SCRIPT INTEGRITY PROTOCOLS

We propose an efficient and practical PES applied on the integrity of web scripts. Here, we describe three protocols: the first two represent the main functionalities, whereas the third one describes an execution involving each entity.

3.1 Generation of the Integrity File

Firstly, the program editor i.e., $\mathcal{PE}$, should have the list of authorized scripts, along with the security policy required. This means that $\mathcal{PE}$ owns: ($\{\mathsf{hd}\}$, $\{\mathsf{script}\}$). From both of these lists, $\mathcal{PE}$ generates an *Integrity File* which contains $\{\mathsf{hd}\}$, an hashes list of each script $\{\mathsf{H}(\mathsf{script})\}$, and the digital signature of the whole previous information. The complete process is described into the Algorithm 1.

Algorithm 1: Generation of IntFile: IntFileGen.

```
Require: : A private key sk and two sets: $\{\mathsf{hd}\}, \{\mathsf{script}\}$
Ensure: : Generation of $IntFile$
  1: Let $M := (\{\mathsf{hd}\}||\{\mathsf{H}(\mathsf{script})\})$
  2: Computes $\sigma := \mathsf{Sign}(\mathsf{sk}, M)$
  3: return $IntFile := (M||\sigma)$
```

This integrity file is the keystone in these protocols: it will be used as the reference during the comparison with the received web page. It contains a list of hashes, which will be checked by the client to ensure the script integrity. In other words, this file is the deported trust of the editor. Once the $IntFile$ has been generated, the program editor deploys it on the (untrusted) outsourced service of its choice: cloud or web server for instance. From now on, the part of the program editor is over: every future interaction will be done between the (malicious) server and the client.

3.2 Verification of the Integrity File

Once the integrity file has been generated, one needs to check its validity. This obviously relies on the signature verification to ensure that the integrity file has indeed been provided by the program editor $\mathcal{PE}$. Then, a comparison with the content retrieved from the untrusted web server and the one from the $IntFile$ should be made. A total of two checks are made: first on the security policy, and then on the scripts themselves. The security policy provided by the server should match exactly the one decided by the $\mathcal{PE}$: each header included into the web page must also be present in the integrity file. Otherwise, the client stops the check, and the browser does not display the page. In the script verification, the comparison method is relaxed: the client only checks that each script needed by the web page is allowed by the program editor. Therefore, if only a part of the scripts is required i.e., some of them are listed on the integrity file but not on the web page, then the verification is validated. However, a script which has not been added into the integrity file cannot be added by the server. On one hand, we choose to only store the script hashes in order to reduce the size of $IntFile$. This implies that the client has to compute hashes of the retrieved scripts to make the comparison. On the other hand, we directly store the headers, since there are short strings. Moreover, this allows the client to directly checks the security policy deployed by the $\mathcal{PE}$. The complete verification process is detailed in the Algorithm 2.

Algorithm 2: Integrity Verification: Match.

```
Require: A public key pk, the $WebPage = (\{\mathsf{hd}\}_{WP}||\{\mathsf{script}\}_{WP})$ and the $IntFile = \{\mathsf{hd}\}_{IF}||\{\mathsf{H}(\mathsf{script})\}_{IF}||\sigma$
Ensure: Return true if the WebPage integrity is verified, false otherwise.
 1: if $\mathsf{Verif}(\mathsf{pk}, IntFile) ==$ false then
 2:     return false
 3: end if
 4: if $(\#\{\mathsf{hd}\}_{WP} \neq \#\{\mathsf{hd}\}_{IF}) \vee (\#\{\mathsf{script}\}_{WP} > \#\{\mathsf{H}(\mathsf{script})\}_{IF})$ then
 5:     return false
 6: end if
 7: if $\exists \mathsf{hd}_i \in \{\mathsf{hd}\}_{WP}$ s.t. $\mathsf{hd}_i \notin \{\mathsf{hd}\}_{IF}$ then
 8:     return false
 9: end if
10: if $\exists \mathsf{script}_i \in \{\mathsf{script}\}_{WP}$ s.t. $\mathsf{H}(\mathsf{script}_i) \notin \{\mathsf{script}\}_{IF}$ then
11:     return false
12: end if
13: return true
```

3.3 Protocol Execution Instance

We now show a correct way of executing the previous protocols, with a client $\mathcal{C}$, a server $\mathcal{S}$ and a program editor $\mathcal{PE}$. We denote by $\Delta_X, X \in \{WP, IF\}$ the validity period of the cache for the web page (WP) or the integrity file (IF). The current cache period of WP or IF is denoted with $\delta_X, X \in \{WP, IF\}$.

Algorithm 3: Outsourced Script Integrity (OSI) Protocols.

```
 1: $\mathcal{PE}$ : Let $IF = \mathsf{IntFileGen}(\{\mathsf{hd}\}_{IF}||\{\mathsf{script}\}_{IF})$
 2: $\mathcal{PE}$ : Publication of $IF$
 3: if $\mathcal{C} : \delta_{WP} > \Delta_{WP}$ then
 4:     $\mathcal{C} \leftarrow Req = (\{\mathsf{hd}\}_{WP}||\{\mathsf{script}\}_{WP})$ from $\mathcal{S}$
 5: end if
 6: if $\mathcal{C} : \delta_{IF} > \Delta_{IF}$ then
 7:     $\mathcal{C} \leftarrow IF$ from $\mathcal{S}$.
 8: end if
 9: if $\mathcal{C} : \mathsf{Match}(\mathsf{pk}_{\mathcal{PE}}, WP, IF) ==$ false then
10:     return $\perp$
11: end if
12: $\mathcal{C}$: Display Web Page
```

Now, we prove that our protocols are correct.

Theorem 3.1 (Correctness)**.** *Let the Outsourced Script Integrity (*OSI*) described in the Algorithm 3 be a* PES*. Then, if the signature schemes* $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verif})$ *with* $(\mathsf{pk}_{\mathcal{PE}}, \mathsf{sk}_{\mathcal{PE}}) \xleftarrow{\$} \mathsf{KeyGen}_\Sigma(1^k)$ *is correct, and if for all sets* $\{\mathsf{hd}\}_X, \{\mathsf{script}\}_X, X \in \{IF, WP\}$ *such that* $\{\mathsf{hd}\}_{WP} = \{\mathsf{hd}\}_{IF}$ *and* $\{\mathsf{script}\}_{WP} \subset \{\mathsf{script}\}_{IF}$*, then* OSI *is correct.*

Proof. We assume that the client is able to get the public key $\mathsf{pk}_{\mathcal{PE}}$ of the developer $\mathcal{PE}$. Then, it is able to successfully check the signature $\sigma = \mathsf{Sign}(\mathsf{sk}_{\mathcal{PE}}, \{\mathsf{hd}\}_{IF}||\{\mathsf{H}(\mathsf{script})\}_{IF})$ of the integrity file $IF = (\{\mathsf{hd}\}_{IF}||\{\mathsf{script}\}_{IF}||\sigma)$ previously retrieved. If the server $\mathcal{S}$ sends a valid web page $WP = (\{\mathsf{hd}\}_{WP}||\{\mathsf{script}\}_{WP})$, the latter will contain all the headers hd from IF, along with at least a subset of allowed scripts. Thus, the cardinal checks on the sets are successfully passed. Then, $\mathcal{C}$ is able to compute the hash for each script and to find a correspondence such that for all scripts into WP, its hash is equal to a hash located into the set $\{\mathsf{H}(\mathsf{script})\}_{IF}$. Then, for all correct requests $(\mathsf{pk}_{\mathcal{PE}}, WP, IF)$ the Match will return true. Therefore, the protocol described in the Algorithm 3 is correct. □

4 SECURITY ANALYSIS

The following theorem summarized the security provided by the OSI schemes. As usual when the security relies on signature schemes, the mechanisms used to get public parameters are abstracted, so that a PKI is assumed to be deployed. For instances, some adequate architectures are discussed in 2.4.

Theorem 4.1. *Let the* OSI *be an instance of a* PES*. Let* $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verif})$ *be a digital signature scheme, and* H *a hash function. If* Σ *is secure against existential forgery under adaptive chosen-message attacks, and that* H *is* 2^{nd}*-preimage resistant, then* OSI *is a secure* PES*. For any p.p.t. adversary* $\mathcal{A}$ *against* OSI*, there exist adversaries* $\mathcal{B}$ *against* Σ *and* $\mathcal{C}$ *against* H *such that:*

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{OSI}}(1^k) \leq 2 * \mathbf{Adv}_{\mathcal{B}}^{2^{\mathrm{nd}}\mathrm{Pre}}() + \mathbf{Adv}_{\mathcal{C}}^{\text{UF-CMA}}(1^k)$$

Proof. By contradiction, let us suppose that it exists an efficient adversary $\mathcal{A}$ which is able to break the scheme. From the security game $\mathbf{Game}_{\mathcal{A}}^{\mathsf{PES}}$, the adversary $\mathcal{A}$ wins if it sends a modified IF^* (i.e., unlike the one provided at the beginning of the challenge) or a non valid WP^* (w.r.t. the definition of the correctness) such that Match returns 1. Assuming that it exists such an adversary $\mathcal{A}$, we construct efficient adversaries which, according to the behavior of $\mathcal{A}$, break either the digital signature scheme Σ or the hash function H.

From the protocol definition, we have the web page, $WP = (\{\mathsf{hd}\}_{WP}||\{\mathsf{script}\}_{WP})$ and the integrity file $IF = (\{\mathsf{hd}\}_{IF}||\{\mathsf{H}(\mathsf{script})\}_{IF})||\sigma)$, where $\sigma = \mathsf{Sign}(\mathsf{sk}_{\mathcal{PE}}, \{\mathsf{hd}\}_{IF}||\{\mathsf{H}(\mathsf{script})\}_{IF})$. We define two events respectively associated to the previous cases: WP-Forged and IF-Forged. Then, we obtain:

$$\mathbf{Adv}^{\mathsf{OSI}}_{\mathcal{A}}(1^k) \leq \Pr[\mathsf{WP\text{-}Forged}] + \Pr[\mathsf{IF\text{-}Forged}] \quad (1)$$

$\boxed{\text{Case 1: WP-Forged}}$

Firstly, we assume the event WP-Forged i.e., the adversary $\mathcal{A}$ has an attack such that only the web page has been altered, so that $\mathsf{Match}(\mathsf{pk}, IF, WP^*) = 1$, where IF is obtained from the signature oracle at the beginning of the attack. Then, $\mathcal{A}$ has modified at least one header or one script i.e. $\{\mathsf{hd}\}_{WP^*} \neq \{\mathsf{hd}\}_{IF}$ or $\{\mathsf{script}\}_{WP^*} \not\subset \{\mathsf{script}\}_{IF}$. Let WP-Hd-Forged and WP-Script-Forged be the associated events to each of these possibilities. Then, we obtain: $\Pr[\mathsf{WP\text{-}Forged}] \leq \Pr[\mathsf{WP\text{-}Hd\text{-}Forged}] + \Pr[\mathsf{WP\text{-}Script\text{-}Forged}]$. Yet, the event WP-Hd-Forged is impossible, since it relies on a direct comparison between two sets, no attack could be realized. Thus $\Pr[\mathsf{WP\text{-}Hd\text{-}Forged}] = 0$. The second event $\Pr[\mathsf{WP\text{-}Script\text{-}Forged}]$ relies on the hash function H. In this case, let $\mathcal{B}$ be an adversary against H. From $\mathcal{A}$, the adversary $\mathcal{B}$ retrieves the set $\{\mathsf{script}\}_{WP^*}$. In particular, it extracts the value $\mathsf{script}_{WP^*} \in \{\mathsf{script}\}_{WP^*}$ such that $\mathsf{H}(\mathsf{script}_{WP^*}) = \mathsf{H}(\mathsf{script}_{IF})$ (with $\mathsf{script}_{IF} \in \{\mathsf{script}\}_{IF}$) and $\mathsf{script}_{WP^*} \neq \mathsf{script}_{IF}$. Then, the adversary $\mathcal{B}$ has found a collision triplet $(\mathsf{script}_{WP^*}, \mathsf{script}_{IF}, h)$, with h the associated hash value. Hence, we obtain:

$$\Pr[\mathsf{WP\text{-}Script\text{-}Forged}] \leq \mathbf{Adv}^{2^{\text{nd}}\text{Pr}}_{\mathcal{A}}() \quad (2)$$

$\boxed{\text{Case 2: IF-Forged}}$

Secondly, we assume that the adversary $\mathcal{A}$ has an attack such that the integrity file has been altered, so that $\mathsf{Match}(\mathsf{pk}, IF^*, WP) = 1$. By definition, $IF = (\{\mathsf{hd}\}_{IF} || \{\mathsf{H}(\mathsf{script})\}_{IF} || \sigma)$, so $\mathcal{A}$ must have modified the content: namely at least one element from one of the set $\{\mathsf{hd}\}_{IF}$ or $\{\mathsf{H}(\mathsf{script})\}_{IF}$. As previously, we associate each possibility to an event, that being IF-Hd-Forged for the first case, and IF-Script-Forged for the second one. Hence, we can bound the probability that the event IF-Forged occurs: $\Pr[\mathsf{IF\text{-}Forged}] \leq \Pr[\mathsf{IF\text{-}Hd\text{-}Forged}] + \Pr[\mathsf{IF\text{-}Script\text{-}Forged}]$. The event $\Pr[\mathsf{IF\text{-}Hd\text{-}Forged}]$ implies that at least one element from the original set $\{\mathsf{hd}\}_{IF}$ has been modified. Consequently, the signature σ must also have been recomputed, otherwise the $\mathcal{A}$ cannot win the security game. Then, the adversary has sent the following parameters $(\mathsf{pk}, (\{\mathsf{hd}\}^*_{IF}, \{\mathsf{H}(\mathsf{script})\}_{IF}) || \sigma^*)$. Let $\mathcal{B}$ be an adversary against the digital signature scheme Σ . By using $\mathcal{A}$ as a subroutine, $\mathcal{B}$ extracts $M^* = (\{\mathsf{hd}\}^*_{IF} || \{\mathsf{H}(\mathsf{script})\}_{IF})$ and σ^*. Then, it wins the digital signature security game by sending $(M^* || \sigma^*)$ to $\mathcal{O}_{\mathsf{Verif}}$. This shows $\Pr[\mathsf{IF\text{-}Hd\text{-}Forged}] \leq \mathbf{Adv}^{\text{UF}-\text{CMA}}_{\mathcal{A}}(1^k)$. Finally, we have to bound the $\Pr[\mathsf{Script\text{-}IF\text{-}Forged}]$. In this case, we proceed by using game-based sequence, where the transitions rest on aborting on specific events.

<u>Game G_0</u>: This is the event Script-IF-Forged, i.e. the adversary $\mathcal{A}$ succeeds in modifying the set script.

<u>Game G_1</u>: We exclude hash attacks by aborting whenever there are two different script_1 and script_2 which return the same digest from the function H. By defining $\mathcal{B}_{G_0}$ as an adversary for the Game 0, which outputs these two collision values for H, we can bound the probability by the $\mathcal{B}_{G_0}$'s advantage in breaking H. Hence, we obtain:

$$\mathbf{Adv}^{\text{G}_0}_{\mathcal{A}}(1^k) \leq \mathbf{Adv}^{\text{G}_1}_{\mathcal{A}}(1^k) + \mathbf{Adv}^{2^{\text{nd}}\text{Pre}}_{\mathcal{B}_{G_0}}()$$

<u>Game G_2</u>: We abort whenever a successful forgery is made on the digital signature scheme Σ. We define an adversary $\mathcal{B}_{G_1}$ which outputs the forgery such that: (M^*, σ^*), with $M^* = (\{\mathsf{hd}\}^*_{IF}$, $\{\mathsf{H}(\mathsf{script})\}_{IF})$ and σ^* the associated signature. The probability of aborting is:

$$\mathbf{Adv}^{\text{G}_1}_{\mathcal{A}}(1^k) \leq \mathbf{Adv}^{\text{G}_2}_{\mathcal{A}}(1^k) + \mathbf{Adv}^{\text{UF-CMA}}_{\mathcal{B}_{G_1}}(1^k)$$

We have excluded hash collisions and signature forgery: from this point, the attacker has no possibility to win the game since every parameters have been dismissed. Therefore: $\mathbf{Adv}^{\text{G}_2}_{\mathcal{A}}(1^k) = 0$. Hence, we obtain:

$$\Pr[\mathsf{Script\text{-}IF\text{-}Forged}] \leq \mathbf{Adv}^{2^{\text{nd}}\text{Pre}}_{\mathcal{B}_{G_0}}() + \mathbf{Adv}^{\text{UF-CMA}}_{\mathcal{B}_{G_1}}(1^k) \quad (3)$$

Assuming that it exists $\mathcal{B}$ against H and $\mathcal{C}$ against the signature scheme, from the equations 1, 2 3, we obtain:

$$\mathbf{Adv}^{\mathsf{OSI}}_{\mathcal{A}}(1^k) \leq 2 * \mathbf{Adv}^{2^{\text{nd}}\text{Pre}}_{\mathcal{B}}() + \mathbf{Adv}^{\text{UF}-\text{CMA}}_{\mathcal{C}}(1^k)$$

$\square$

5 IMPLEMENTATION - WEBEXTENSION PLUGIN

To implement our solution, we decide to develop a browser plugin, using the WebExtension standard. Using plugins provides us with an effective way to be compatible with most browsers (Firefox, Chrome, Edge) and most platforms (Windows, Linux, Mac, Android...) with a minimum effort. This is a proof of concept [1] of our solution, but our main goal is to push

[1]Source code is available at github.com/wallix/.

it directly into the navigator as a new security feature. The WebExtension framework, used to develop such a plugin, is done with JavaScript. We choose to use only a minimum of non standard Javascript libraries and make our dependencies as small as possible to ensure an easy as possible code audit. Since this plugin needs to intercept any response received by the browser, coding flaws should be detected to not compromise the user security. In the following PoC, we assume that the key has been given out-of-bands. Besides being the easiest solution to implement, this also fulfills the requirements of the security software editor use case: the key could have easily been given *a priori*.

5.1 Configuration Options

Only some specific websites will be monitored by the plugin. We intend to minimize the number of parameters for each site, so that it will be easy for any user to add a website with the Javascript Integrity feature.

At the moment, only two options are mandatory and must be set by the user:

- The website domain name;
- The public key used to sign the *IntFile*.

These parameters must be retrieved from a trusted source as they ensure the authenticity of the *IntFile*. This critical step is out of the scope of this document. We plan to implement one of the previous solutions (discussed in 2.4) in a future version. We only target some specific websites and needs first. The domain name specifies the URI we want to monitor. The drawback of storing the domain name is that we cannot follow cross-origin content such as *iFrames*. These frames may come from another domain, and include remote Javascript as well. However we cannot ensure that they implement our *IntFile* based solution. To enforce security on *iFrames*, we can:

- Ignore the *iFrame*;
- Include cross-origin information in our *IntFile*. If the website changes, we have to change our *IntFile*, so we have to monitor continuously this site for any changes in Javascript or HTTP headers.

Currently, we have only implemented the first solution. We assume that iFrames can easily be avoided on the websites we target. For now, we only use a public key to check the integrity of the *IntFile*, without metadata (date, revocation capabilities, ...). This could be improved by using X.509 certificates for example, but from our knowledge, there is no embedded parser available for WebExtension without importing a *huge* library such as (Strozhevsky, 2018). This configuration is stored in the browser local storage.

5.2 Dynamic *IntFile* Retrieval

As seen previously, the *IntFile* can be stored on a non secure storage. This enable us to store this file into the same website as the one we are monitoring at a well-known address: *https://www.domain.com/.well-known/site-manifest.json*.

The plugin will have to fetch this file on a regular basis, at least on each session startup, and once a day. As we need our plugin to be fast, our strategy is to load each file at browser startup. We assume that the amount of monitored website will be small enough, so that it will not block or slow the browser down nor interfere with the user experience. If the retrieval does not work, for example after the deletion of the *IntFile*, a warning must and will be displayed to the user. Then he can either choose to continue by using his last cache entry (if exists) or to reject the connection. For simplicity, the *IntFile* will be a JSON file, storing the hashes, the HTTP headers we want to monitor and a version number. Some other fields might be added in the future (such as a public key to enable automatic updates).

```
{
  url: 'lab.wallix.com',
  version: 1.0,
  headers: [
  { name: 'X-Content-Type-Options',
    value: 'nosniff, nosniff'},
  { name: 'Cache-Control',
    value: 'max-age=0, private, must-revalidate'}],
  scripts: [
  { src: '/static/script.js',
    integrity: 'sha256-819f04e570...9aae97'},
  { src: '/static/script1.js',
    integrity: 'sha256-43ff03e7a5...30f8e4'}],
}
```

Listing 1: IntFile.

5.3 IntFile Signature Verification

As we get the *IntFile* from an untrusted source, we have to check that it was signed by the public key stored in the plugin configuration. The file is signed following the JOSE standard (IETF, 2018), with the protected headers. The signature is done following these steps:

1. Encoding payload as a base64 string;
2. Defining a header for metadata (signature algorithm, ...) as a JSON object;
3. Encoding metadata as a base64 string;
4. Appending payload and metadata and sign the whole string;
5. Appending the signature to the file.

```
{
  payload: 'MGY1MGFjO...Tc2MzkK',
  header: 'OTRY2YwYWJ...jlNz0cK',
  signature: 'N2FlNTM...jk1NjAK'
}
```

Listing 2: JOSE format.

If the signature verification failed, we prompt an error message to the user. Then, there are two possibilities: either stopping the navigation to this website or continuing with the plugin deactivated only for this specific website. Moreover, the user is able to configure the duration while the plugin is disabled.

5.4 HTTP Header Verification

As seen previously, the policy enforcement is mainly done through HTTP headers, like *Content Security Policy*, *Public Key Pins*, *XSS Protection*, *HTTP Strict Transport Security...* As we don't trust the web server, we must ensure that they are properly set and configured. For our needs the *Content Security Policy* is the most important header, as it allows us to define the source where the browser might download external scripts (and also deny inlined scripts) with the *script-src* parameter. We also thought to use the parameter *require-sri-for* (MDN, 2018) which enforces the presence of *SRI* for each Javascript defined. This header is supported in Chrome and Edge but not in Firefox, which means that we cannot count on this behavior for our extension. Thus, we manually check that every script tag has an integrity attribute.

HTTP headers are available through the *onHeaderReceivedListener()* callback, from the background plugin context. This callback can be synchronous or asynchronous. As we need to inspect headers and page content before the browser parses and constructs the DOM, we have to use the synchronous call. The headers chosen to construct the security policy are defined in the *IntFile*. For each one of them, we need to ensure that it appears only once in the server response. If a header is duplicated, browser comportment is undefined and might take any (or all) of them into account.

5.5 Page Content Verification

The page content verification ensures that for each script defined in the document, an associated hash has previously been configured in the *IntFile*, so that no untrusted Javascript will be loaded. The page content can be retrieved from lots of callbacks, but we need to ensure that the browser will not parse the DOM nor execute Javascript before all the checks have been realized. So we need to find a way to block the response until our analysis is done. The better way we found is to make a data filter with *filterResponseData()* from inside the *onHeaderReceivedListener()*. The goal of parsing the whole document is to search for every Javascript elements. DOM has a specific attribute, called *document.scripts* which will retrieve all the lists for any document. The latter method avoids to reimplement a DOM parser. To get a proper result, we need to feed the DOM parser with the whole document at once. The *ondata* event can be triggered multiple times for a single page, so we cannot use it directly to build our document. Instead, we define a string that will contain chunks and then, before closing the filter, will give this string to the DOM parser, as seen in listing 5.5.

```
let pageContent = ''
let filter = browser.webRequest.
              filterResponseData(details.requestId)
let decoder = new TextDecoder("utf-8")
let encoder = new TextEncoder()

filter.ondata = event => {
  pageContent += decoder.decode(event.data)
}

filter.onstop = event => {
  let isPageValid = validatePage(pageContent, config['scripts'])

  if (config['ignore_errors'] || (areHeadersValid && isPageValid)) {
    filter.write(encoder.encode(pageContent))
  }
  else if (areHeadersValid == false)
    filter.write(encoder.encode('Bad headers'))
  ...
  filter.disconnect()
}
```

Listing 3: Document retrieval.

The *validatePage()* function can then parse the DOM, search for every scripts and:

- Returns an error if at least one script does not have an integrity attribute, as the *require-sri-for* would do;
- Returns an error if the hash provided in the attribute does not match the hash provided in the *IntFile*;
- Returns success if there are not any error.

By using the CSP parameter *script-src*, we ensure that all the imported scripts come from an external source (i.e., there is no inline script). Furthermore, our plugin enforces the SRI checking from a trusted hash pool. Some Javascript methods cannot be monitored, such as *importScript()* in a *WebWorker*. These loading methods do not provide any way to check the integrity, so we recommend to avoid them.

5.6 Error Handling

All checks are done in the background context of the script before rendering the page. We have no way

Table 1: Average timings (in seconds) and relative changes of the plugin overhead.

	10 scripts	100 scripts	1000 scripts
No Integrity Attribute	0,373	2,28	24,18
With SRI	0,355	2,312	24,874
With Our Plugin	0,401	2,554	26,393
Relative Change	13%	11%	6%

to provide any user feedback since the method *runtime.sendMessage()* is not instantiated in the user context. The only way we found is to build an error page providing information to the user on why the check failed (*IntFile* was not up-to-date, signature verification failed, some hashes were incorrect...) and to prompt the error message afterward. The user can then choose to override this error and to force the page reloading.

5.7 Performance Analysis

Since the plugin interferes with the browsing experience of the user, we must limit its impact on the browser efficiency. To check the validity of Javascript inside the document, we use a method running in the background context of our plugin and blocking the request. This method does not interfere with other requests or other tab activity in the browser. To observe the impact of the plugin on the page loading time, we have made a benchmark by measuring it (in seconds) with *Firefox Development Tools* on a couple of pages. The plugin might impact the performances at two steps of the algorithm: during the background DOM building and while checking the page compliance with the rules in the site-manifest. Both of them are only executed on watched pages, meaning the impact on non-supervised sites is just the time to check if the domain belongs to the list of watched websites. This benchmark is made with a local server distributing pages on a Firefox version 65.0 browser executed on a computer with Intel Core i7-8550U processor and 16 GB of RAM for different types of pages. To generate a pool of scripts, we have locally downloaded the library "jquery-3.3.1.min.js" (85 KB), and added one space at the end of the latter file. Consequently, the associated hashes are distinct for each script. The results are summarized into Table 1.

These results show that the average overhead from the plugin has a low impact on the browsing. We run benchmarks on an unrealistic number of scripts to amortize the imprecision of the *Mozilla Firefox Development Tools* timer. In practice, the difference of behavior during the browsing is imperceptible for the user.

6 CONCLUSION

In this article, we propose protocols to ensure the integrity of web scripts with a malicious web server. These protocols allow a developer to deploy their program on outsourced services, such as a cloud, without having to grant any trust. Although focused on the web applications, our protocols provide general solution, easily adaptable to distribution of any applications.

Our solution suggests several improvements. A fine-grain management of versioning might be desirable, along with a dedicated implementation. However, this might partly be answered using a publicly verifiable key management. Generalizing the type of objects taken into account, as *iFrames* or images, is an interesting enhancement regarding the compatibility of the plug-in. Managing multiple integrity files along with chaining signatures seems to be a reasonable approach to solve this issue. As seen in the implementation, some way of loading Javascript (ex: *importScript()*) cannot be checked as they do not provide a specific way to include *SRI* hashes. We must focus on these methods to ensure that an insecure Javascript cannot be loaded by any script.

REFERENCES

Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S. (2005). DNS security introduction and requirements. RFC 4033, RFC Editor. http://www.rfc-editor.org/rfc/rfc4033.txt.

Axon., L. and Goldsmith., M. (2017). Pb-pki: A privacy-aware blockchain-based pki. In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications - Volume 6: SECRYPT, (ICETE 2017)*, pages 311–318. INSTICC, SciTePress.

Bielova, N. (2013). Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *The Journal of Logic and Algebraic Programming*, 82(8).

Chen, J., Jiang, J., Duan, H., Wan, T., Chen, S., Paxson, V., and Yang, M. (2018). We still don't have secure cross-domain requests: an empirical study of cors. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 1079–1093. USENIX Association.

De Volder, K. (2006). Jquery: A generic code browser with a declarative configuration language. In *International Symposium on Practical Aspects of Declarative Languages*, pages 88–102. Springer.

Dumas, J.-G., Lafourcade, P., Melemedjian, F., Orfila, J.-B., and Thoniel, P. (2017a). Localpki: An interoperable and iot friendly pki. In *International Conference on E-Business and Telecommunications*, pages 224–252. Springer.

Dumas, J.-G., Lafourcade, P., Orfila, J.-B., and Puys, M. (2017b). Dual protocols for private multi-party matrix multiplication and trust computations. *Computers & security*, 71:51–70.

Eskandari, S., Leoutsarakos, A., Mursch, T., and Clark, J. (2018). A first look at browser-based cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 58–66. IEEE.

Farber, D. A., Greer, R. E., Swart, A. D., and Balter, J. A. (2003). Internet content delivery network. US Patent 6,654,807.

Garay, J. A., Kiayias, A., Leonardos, N., and Panagiotakos, G. (2018). Bootstrapping the Blockchain, with Applications to Consensus and Fast PKI Setup. In *Public-Key Cryptography – PKC 2018*. Springer.

IETF (2018 (accessed 12/3/2018)). *Javascript Object Signing and Encryption (jose)*. https://datatracker.ietf.org/wg/jose/charter/.

Jim, T., Swamy, N., and Hicks, M. (2007). Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 601–610. ACM.

Jøsang, A. and Dar, K. S. (2011). Server certificates based on dnssec. In *Proceedings of NordSec*.

Kubilay, M. Y., Kiraz, M. S., and Mantar, H. A. (2019). Certledger: A new pki model with certificate transparency based on blockchain. *Computers & Security*, 85:333–352.

Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. (2017). Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society.

MDN (2018 (accessed 12/12/2018)). *CSP:require-sri-for*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-sri-for.

Meyerovich, L. A. and Livshits, B. (2010). Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *2010 IEEE Symposium on Security and Privacy*, pages 481–496. IEEE.

Mitropoulos, D., Stroggylos, K., Spinellis, D., and Keromytis, A. D. (2016). How to Train Your Browser: Preventing XSS Attacks Using Contextual Script Fingerprints. *ACM Transactions on Privacy and Security*.

Mozilla (2008). *Signed Scripts in Mozilla*. https://www-archive.mozilla.org/projects/security/components/signed-scripts.htm.

Mozilla (2019). *Security and the jar protocol*. https://developer.mozilla.org/en-US/docs/Mozilla/Security/Security_and_the_jar_protocol.

Nadji, Y., Saxena, P., and Song, D. (2009). Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, volume 20.

Nakhaei, K., Ansari, E., and Ansari, F. (2018). Jssignature: Eliminating third-party-hosted javascript infection threats using digital signatures. *CoRR*, abs/1812.03939.

Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. (2012). You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 736–747. ACM.

Rogaway, P. and Shrimpton, T. (2004). Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Roy, B. and Meier, W., editors, *Fast Software Encryption*, pages 371–388. Springer Berlin Heidelberg.

Ruderman, J. (2018 (accessed 12/20/2018)). *Same-origin policy*. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

Ruohonen, J., Salovaara, J., and Leppanen, V. (2018). On the integrity of cross-origin javascripts. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 385–398. Springer.

Ryan, M. D. (2014). Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*.

Saiedian, H. and Broyle, D. (2011). Security vulnerabilities in the same-origin policy: Implications and alternatives. *Computer*, 44(9):29–36.

Soni, P., Budianto, E., and Saxena, P. (2015). The sicilian defense: Signature-based whitelisting of web javascript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1542–1557.

Strozhevsky, Y. (2018 (accessed 12/12/2018)). *PKI.js*. https://github.com/PeculiarVentures/PKI.js.

Team, T. P. M. (January 20, 2019 (accessed 01/21/2019)). *Response to analysis of ProtonMail's cryptographic architecture*. https://protonmail.com/blog/cryptographic-architecture-response/.

W3C (2018). *Historical trends in the usage of client-side programming languages for websites*. https://w3techs.com/technologies/history_overview/client_side_language/all.

Weinberger, J., Braun, F., Marier, F., and Akhawe, D. (2016). Subresource integrity. W3C recommendation, W3C. http://www.w3.org/TR/2016/REC-SRI-20160623/.

West, M. (2018). Content security policy level 3. W3C working draft, W3C. https://www.w3.org/TR/2018/WD-CSP3-20181015/.

West, M. (October 20, 2017 (accessed 01/29/2019)). *Signature Based SRI*. https://github.com/mikewest/signature-based-sri.

Yakubov, A., Shbair, W., and State, R. (2018). Blockpgp: A blockchain-based framework for pgp key servers. In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 316–322. IEEE.

Yakubov, A., Shbair, W. M., Wallbom, A., Sanda, D., and State, R. (2018). A blockchain-based pki management framework. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6.