Raising the Bar: Evaluating Origin-wide Security Manifests

Steven Van Acker
Chalmers University of Technology
Gothenburg, Sweden

Daniel Hausknecht
Chalmers University of Technology
Gothenburg, Sweden

Andrei Sabelfeld
Chalmers University of Technology
Gothenburg, Sweden

ABSTRACT

Defending a web application from attackers requires the correct configuration of several web security mechanisms for each and every web page in that web application. This configuration process can be difficult and result in gaps in the defense against web attackers because some web pages may be overlooked. In this work we provide a first evaluation of the standard draft for an origin-wide security configuration mechanism called the "origin manifest". The mechanism raises the security level of an entire web origin at once while still allowing the specification of web security policies at the web page level. We create prototype implementations of the origin manifest mechanism for both the client-side and server-side, and provide security officers with an automated origin manifest learner and generator to aid them with the configuration of their web origins. To resolve potential collisions of policies defined by the web origin with policies defined by web pages we formalize the comparison and combination of web security policies and integrate it into our prototype implementation. We evaluate the feasibility of the origin manifest mechanism with a longitudinal study of popular websites to determine whether origin manifest files are stable enough to not require frequent reconfiguration, and perform performance measurements on the Alexa top 10,000 to determine the network traffic overhead. Our results show that the origin manifest mechanism can effectively raise the security level of a web origin while slightly improving network performance.

ACM Reference Format:
Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. 2018. Raising the Bar: Evaluating Origin-wide Security Manifests. In *Proceedings of 2018 Annual Computer Security Applications Conference (ACSAC '18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274694.3274701

1 INTRODUCTION

Today's web connects billions of people across the planet through interactive and increasingly powerful web applications. These web applications are a complicated mix of components on both server- and client-side. Unfortunately, current security mechanisms are spread across the different components, opening up for inconsistencies. Previous work [2, 17, 31, 39, 42] shows that it is hard to securely configure and use these mechanisms.

Web application security policies are typically transmitted through HTTP headers from the server to the client. While most web security mechanisms operate at the level of a single web page, some, like HSTS [18] and HPKP [12], operate at the level of an entire *web origin*. The web origin, or simply *origin*, defined as a combination of the scheme, hostname and port, serves as the *de facto security boundary* in web security. Security mechanisms, if misconfigured at the level for a single web page, may break the operation of an entire origin. For these reasons, it is valuable to define the scope of a security policy at the origin level and meaningfully combine it with application-specific policies for enforcement on the client side. These considerations have prompted the web security community to propose a draft to specify a *security manifest* [16, 41] to allow definition of security policies at the origin level. The goal it to provide a *backward-compatible origin-wide mechanism*, so that security officers can harden web application security without imposing the burden of a new mechanism on developers.

To illustrate the need for the origin manifest, consider a web application for which the developers set a *Content Security Policy (CSP)* [34] for every web page, while missing to configure CSP for their custom 404 error page. If this page has a vulnerability, it puts the entire web application at risk. This scenario is realistic [13, 19, 26], while not limited to error pages or CSP. For web pages where security mechanisms are left unconfigured, this motivates a **fallback policy**: a default setting for a security policy.

Let us extend this example scenario with additional web applications hosted under the same web origin. The same-origin policy (SOP) specifies that access between web origins is not allowed by default. In our extended example the web applications are under the same origin and a vulnerability in one application can potentially put the others at risk since SOP as a security boundary does not protect in this case. To raise the bar for attackers, origin manifest provides a **baseline policy** for an entire origin: a minimum origin-wide security setting which can not be overridden, only reinforced.

Note that the baseline policy can not currently be implemented by simply centralizing web security policies through e.g. the central web server configuration [28]. In such a setup, when the central configuration for a web security header is in conflict with one set by the web application, one header must be prioritized over the other. The baseline policy combines both configurations so that the result is at least as strong as each configuration.

An implementation of the origin manifest mechanism has been initiated for the Chrome browser [40]. At the same time, there are open questions [38] about the potential usefulness of the mechanism. The following research questions are critical to determine whether the origin manifest mechanism is going to make it or break it: How to combine origin-wide and application-specific policies? How to aid developers in configuring origin manifests? What is the

ACSAC '18, December 3-7, 2018, San Juan, PR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274701

expected lifespan of an origin manifest? Does the mechanism degrade performance or, on the contrary, can improve it? This paper seeks to answer these research questions.

Security improvements through origin-wide baseline policies are promising but the draft lacks details on how to resolve situations in which policies defined by the origin collide with policies defined by web pages. Consider a situation in which both origin and web page define different `Strict-Transport-Security` policies. The problem is that `Strict-Transport-Security` does not allow multiple policy definitions for the same page, a situation the origin manifest mechanism should specify how to resolve. To this end we determined the need to compare the security level of security policies, as well as the need to combine security policies into their least upper bound and greatest lower bound. We formalize the comparison and combination of security policies as an extension of the origin manifest mechanism and create an implementation for practical evaluation. During implementation, we also realized that baseline policies do not work well for certain security policies, such as security flags for web cookies, necessitating the introduction of **augmentonly policies**.

In real world deployments the security officers responsible for a web origin are not necessarily the developers of the web applications hosted under that origin. Therefore origin security officers do not always have full control over the configurations of the web applications. Fortunately, origin manifest does not require this level of control to take effect and web applications can stay untouched. Nevertheless, a practical challenge is to define suitable origin-wide security policies with a certain level of desired security but without breaking web applications hosted under the origin. A good starting point is to identify and merge all policies deployed under an origin to create an origin manifest which covers the policies of each web application. To support origin security officers in this non-trivial task we implemented a tool which can learn the deployed security configurations of web applications under an origin. The tool utilizes the policy combinator functions to generate an origin manifest which is in accordance with all observed web application policies. Origin security officers can then refine this generated origin manifest according to their requirements.

A stable origin manifest would reduce the workload on origin security officers, but requires data on how frequently HTTP headers tend to change in real-world web applications. To this end we conducted an longitudinal empirical study over 100 days to analyze the popularity, size and stability of HTTP headers. We used the origin manifest learner and generator to derive origin manifests for each visited origin to get a first insight into the practical composition of origin manifests over a longer period of time. One of our results is an average origin manifest stability of around 18 days.

The origin manifest draft claims that HTTP headers are often repeated and can occupy multiple KiB per request, an overhead which can be reduced by sending the respective headers as part of the origin-wide configuration. Cross Origin Resources Sharing (CORS) preflights, which query the server for permission to use certain resources from different web origins, can be cached per web origin to reduce network traffic. Though intuitively this might seem plausible we feel that both claims can benefit from empirical evidence and practical evaluation. To this end we first implemented a prototype for the origin manifest mechanism using proxies. We then used the prototype in a large-scale empirical study to visit the Alexa top 10,000 and to analyze the network traffic without and retrofitted with origin manifest. Our results show that there is a slight reduction of network traffic when using origin manifests.

Addressing the above-mentioned research questions our main contributions include:

- Extensions to the proposed origin manifest draft:
  - A formal description of security policy comparison and combination functions
  - Introduction of a new `augmentonly` directive
- Automated origin manifest learner and generator[1]
- Evaluation with empirical evidence for:
  - the feasibility of the origin manifest mechanism in the form of a longitudinal study of the popularity, size and stability of observed HTTP headers in the real world
  - the origin manifest mechanism's network traffic overhead, by measuring and studying the network traffic while visiting the Alexa top 10,000 retrofitted with origin manifests

The rest of this paper is structured as follows: Section 2 describes the web security mechanisms which the origin manifest mechanism covers. Section 3 outlines the design of the origin manifest mechanism. Section 4 formalizes comparisons and combinators for security policies. Section 5 provides details of our prototypes that implement the origin manifest mechanism. Section 6 deals with the evaluation of our prototypes. We provide a general discussion in Section 7, list related work in Section 8 and conclude in Section 9.

2 BACKGROUND

Browsers implement certain security-relevant mechanisms which can be configured by servers via HTTP headers. The values of the respective headers therefore represent a security policy enforced by browsers. In this section we briefly explain the security mechanisms that can be configured with an origin manifest.

Set-Cookie. The `Set-Cookie` HTTP header allows the setting of web cookies [3]. Cookies can be configured with additional attributes such as `httpOnly` which makes the cookie inaccessible from JavaScript, and `secure` which disallows the transmission of the cookie over an insecure connection. These attributes form a policy, specifying how cookies should be handled by browsers.

Content-Security-Policy (CSP). A CSP whitelists which content is allowed to be loaded into a web page. To this end CSP defines various directives for different content types such as scripts or images but also for sub-frames or the `base-uri` configuration. The directives whitelist the respectively allowed content. We use CSP level 3 as specified in [34].

Cross-Origin Resource Sharing (CORS). By default the same-origin policy does not permit accessing cross-origin resources. CORS [36] allows web developers to explicitly allow a different origin from accessing resources in their own origin. Under certain conditions, e.g. when a request would have a side-effect on the remote side, browsers will perform an upfront *preflight request* to query whether the actual request will be permitted. In contrast to other security mechanisms, CORS access decisions are communicated through

[1]Our implementations available online on http://www.cse.chalmers.se/research/group/security/originmanifest

sets of HTTP headers. The composition of the different CORS headers forms a CORS policy. All CORS response header names follow the pattern 'Access-Control-*'.

X-Content-Type-Options. Some browsers implement content-type sniffing as a mechanism to verify if the expected content-type of a loaded resource matches the content-type of the actually loaded content. The HTTP response header `X-Content-Type-Options: nosniff` disables this behavior.

X-XSS-Protection. Most browsers implement some form of cross-site scripting (XSS) protection, although no standard exists. The `X-XSS-Protection` header can configure this feature. For instance, `X-XSS-Protection: 1; mode=block` will enable XSS protection and will block the loading of the web page if an XSS attack is detected.

Timing-Allow-Origin. Web browsers provide an API for accessing detailed timing information about resource loading. Cross-origin access to this information can be controlled through the `Timing-Allow-Origin` HTTP header [35]. By default cross-origin access is denied. This header allows to define a whitelist of permitted origins.

Strict-Transport-Security. The HTTP header Strict Transport Security (HSTS) [18] is a mechanism to configure user agents to only attempt to connect to a web site over secure HTTPS connections. This policy can be refined through parameters to limit the policy lifetime (`max-age`) or to extend the effects of the policy to subdomains (`includeSubDomains`).

Public-Key-Pins. The HTTP header `Public-Key-Pins` (HPKP) [12] allows to define a whitelist of public key fingerprints of certificates used for secure connections. If an origin's certificate does not match any of the whitelisted fingerprints for that origin, the connection fails. HPKP policies have a lifetime as specified via the `max-age` directive and can be extended to sub-domains through the `includeSubDomains` directive. Note that this header is deprecated for the Chrome browser [8].

X-Frame-Options. The HTTP header `X-Frame-Options` [29] determines whether the response can be embedded in a sub-frame on a web page. It accepts three values: `DENY` disallows all embedding, `SAMEORIGIN` allows embedding in a web page from the same origin, and `ALLOW-FROM <origin>` allows embedding in a web page from the specified origin. Because this mechanism is not standardized, some directives such as e.g. `ALLOW-FROM` are not supported by all browsers. This is why we do not consider `ALLOW-FROM` in our work. In practice, CSP's `frame-ancestors` directive is meant to obsolete the use of this header [33].

3 MECHANISM DESIGN

The standard draft [41] and its explainer document [16] define the basic origin policy mechanism. We take it as the basis for our work but differ in some parts, for example, by adding the `augmentonly` section. In this section we describe the extended origin manifest mechanism.

3.1 Overview

The origin manifest mechanism allows configuring an entire origin. The origin provides this configuration as a manifest file under a well-known location under the origin, according to the concept of Well-Known URIs defined in RFC 5785 [25]. Browsers fetch this manifest file to apply the configurations to every HTTP response from that origin. To this end, all resource requests to the same origin are put on hold until the respective file is downloaded in order to take effect from the first request on. The manifest file is cached to avoid re-fetching on every resource load. Browsers store at most a single origin manifest per origin. A version identifier, communicated via the `Sec-Origin-Manifest` HTTP header, is used to distinguish manifest versions.

3.2 Configuration structure

An origin manifest is a file in JSON format which contains up to five different sections: `baseline`, `fallback`, `augmentonly`, `cors-preflight` and `unsafe-cors-preflight-with-credentials`. An example manifest file is shown in Listing 1.

```
{
 "baseline": {
  "Strict-Transport-Security":"max-age=42",
 },
 "fallback": {
  "Content-Security-Policy": "default-src 'none'",
  "X-Frame-Options": "SAMEORIGIN"
 },
 "augmentonly": {
  "Set-Cookie": "secure"
 },
 "cors-preflight": [ ],
 "unsafe-cors-preflight-with-credentials": [
  {"Access-Control-Allow-Methods": "OPTIONS, GET, POST",
   "Access-Control-Allow-Origin": "b.com",
   "Access-Control-Allow-Headers":"X-ABC",
   "Access-Control-Max-Age": "1728000"}
 ]
}
```

Listing 1: Origin manifest file example

baseline. This section defines the minimum security level for the supported security mechanisms. A web application can not override these settings, only reinforce them. For example an origin might want to exclusively require secure connections by adding the `Strict-Transport-Security` header with an appropriate value to this section.
The following headers can be used: `X-Content-Type-Options`, `X-Frame-Options`, `X-XSS-Protection`, `Timing-Allow-Origin`, `Strict-Transport-Security`, `Content-Security-Policy`, `Public-Key-Pins`, and CORS headers.

fallback. This section defines default values for any HTTP header. They are only applied in case a web application does not provide the respective HTTP header. The `fallback` section ensures the presence of a policy for a mechanism but can also be used to reduce header redundancy by relying on the definition in the manifest. For example an origin may want to set the custom `X-Powered-By` header on each HTTP response, to indicate which software is being used on the server side. It can do this by placing the header in the origin manifest.
There are no restrictions on which headers can be used in the `fallback` list.

augmentonly. Some HTTP headers can be a mixture of data and security policy. An example is the `Set-Cookie` header which can define the flags `secure` and/or `httpOnly` with the actual data. The `augmentonly` section defines policies which are used to augment a response header's policy.
Currently we only consider one header for this section: `Set-Cookie`.

cors-preflight. This section defines a list of CORS preflight decisions. Each CORS preflight response is represented as a JSON object with the CORS headers as its key-value pairs. In contrast to the previously described sections, `cors-preflight` is only used when CORS preflights are to be sent. Before sending a CORS preflight, the browser consults this list for a cached decision. In case no decision matches the CORS preflight, is the actual web server consulted.

unsafe-cors-preflight-with-credentials. This section is in essence the same as the `cors-preflight` section except that it defines CORS pre-flight responses which also transmit credentials.

3.3 Client-side application

For any HTTP response, the `fallback` policy is applied first, by filling in missing headers with the values from the `fallback` policy. Next, both `baseline` and `augmentonly` policies are applied by strengthening their respective headers with the values from the manifest file.

Both the `unsafe-cors-preflight-with-credentials` and `cors-preflight` policies only act on CORS preflight requests. When any of the rules in these sections match the CORS preflight request, the request is not forwarded to the original destination, but handled inside the browser instead. Besides this shortcut, the CORS mechanism itself remains untouched. Once a response for the CORS preflight request is generated, the `fallback` and `baseline` policies are also applied to it.

3.4 Misconfiguration

Origin manifests can be misconfigured. The mechanism itself only provides a way to define certain configuration options. The respective policies are however not validated or otherwise analyzed for, for example, conflicting policies. For example it is possible to define `X-Frame-Options` policies "`a.com`" in the `baseline` section and "`SAMEORIGIN`" in the `fallback` section of an origin manifest. It is the responsibility of the origin administrator to ensure a meaningful manifest file.

4 POLICY COMPARISON AND COMBINATION

The origin manifest mechanism's baseline policy relies on combining security policies to make them stricter. The ability to determine whether a security policy is stricter than another, implies the ability to compare security policies.

In this section, we formalize the notion of comparing the strictness of security policies, using the *"at least as restrictive as"* $\sqsubseteq$ operator. We then use the $\sqsubseteq$ operator to define the *join* $\sqcup$ and *meet* $\sqcap$ combinators, which can be used to combine security policies into a weaker and stricter policy respectively.

4.1 $\sqsubseteq$ for policy comparison

We formalize the comparison of the security policies specified by HTTP headers relevant in the context of origin manifest. Our formal notation draws on the formalism by Calzavara et al. to describe CSP [6, 7].

Some mechanisms come with a reporting feature. We deliberately do not take reporting into account because they do not affect the enforcement of a policy.

4.1.1 Definitions. Let $\sqsubseteq$ stand for the binary relation between two policies such that $p_1 \sqsubseteq p_2$ if and only if everything allowed by p_1 is also allowed by p_2. That is p_1 is as strict or stricter than p_2.

Not all security policies can readily be compared by strictness. For example the policies `Timing-Allow-Origin: https://a.com` and `Timing-Allow-Origin: https://b.com` both allow a single but different origin. These polices are incomparable, making $\sqsubseteq$ a partial (and not total) order.

We represent each HTTP header as a tuple $\langle a_1, \cdots, a_n \rangle$ of values, so that $\langle a_1, \cdots, a_n \rangle \sqsubseteq \langle b_1, \cdots, b_n \rangle \iff \forall i.\, a_i \sqsubseteq b_i$.

We define ϕ as the empty value and $\phi \sqsubseteq a$ for any a in the same domain, unless otherwise specified. We assume H is the set of header names, O is the set of web origins, M is the set of HTTP methods, KP the set of key pins and $\mathcal{P}$(KP) the superset of key pins.

Table 1 summarizes the comparison rules for all security headers covered by the origin manifest mechanism, with the exception of `Content-Security-Policy`. Let us consider the headers `Access-Control-Max-Age` and `Set-Cookie` as examples.

The `Access-Control-Max-Age:` a header has one argument: a natural number a indicating the maximum allowed time that a CORS preflight may be cached. In this example, a lower number represents a stricter policy, so that $a \sqsubseteq b \iff a \leq b$.

The `Set-Cookie: key=value. . .` a, b header has 2 arguments: a indicates whether a cookie is marked `Secure`, and b whether it is marked `httpOnly`. In this case, specifying either `Secure` or `httpOnly` is stricter than not specifying either, while specifying both `Secure` and `httpOnly` is stricter than any other combination.

4.1.2 Content Security Policy. Calzavara et al. [6, 7] formalize the comparison of CSP policies, but omit CSP2.0 and CSP3.0 features such as nonces, hashes and strict-dynamic. We reuse their formalization, but make special arrangements to be compatible with more modern web pages.

CSP nonces are by nature page specific which conflicts with the fundamental idea of origin manifest. We therefore need to transform every CSP into a policy without nonces. The goal is to have a policy that allows at least what the original policy allows to not break web pages. Nonces can be used to mark inline scripts as being included by the developer. Thus a replacement of nonces must include the 'unsafe-inline' flag. Nonces can also be used to permit loading of scripts from a source file. Therefore a replacement of nonces must include a whitelist with any possible URL. That is the wildcard `*` but also the schemes `http:`, `https:`, `ws:`, `wss:` and `data:`.

Hashes enable inline scripts which hash matches with it but can also enable any loaded script in combination with SRI checks. Though hashes are not a problem in the context of origin manifest directly they make the keyword 'unsafe-inline' being ignored. Therefore removing nonces from CSPs implies removing hashes using the same rules.

Table 1: Compositional comparison rules for security headers. ϕ is the empty value and $\phi \sqsubseteq a$ for any a in the same domain, unless otherwise specified. H is the set of header names, O is the set of web origins, M is the set of HTTP methods, KP the set of key pins and $\mathcal{P}(\mathrm{KP})$ the superset of key pins. Tuples can be compared by comparing their components, since $\langle a_1, \cdots, a_n \rangle \sqsubseteq \langle b_1, \cdots, b_n \rangle \iff \forall i.\, a_i \sqsubseteq b_i$

Header	Notation	With	
`Access-Control-Max-Age:` a	$\langle a \rangle$	$a \sqsubseteq b \iff a \leq b$	$a, b \in \mathbb{N}$
`Access-Control-Expose-Headers:` a		$a \sqsubseteq b \iff a \subseteq b$	$a, b \subseteq \mathrm{H}$
`Access-Control-Allow-Headers:` a			$a, b \subseteq \mathrm{H}$
`Access-Control-Allow-Methods:` a			$a, b \subseteq \mathrm{M}$
`Timing-Allow-Origin:` a			$a, b \subseteq \mathrm{O}, "*" = \mathrm{O}$
`Access-Control-Allow-Origin:` a		$a, b \in \mathrm{O}, a \sqsubseteq "*"$	
`Access-Control-Allow-Credentials:` a		$a \in \{"true", "false"\}, "false" \sqsubseteq "true"$	
`X-Content-Type-Options:` a		$a \in \{"nosniff", \phi\}, "nosniff" \sqsubseteq \phi$	
`X-Frame-Options:` a		$a \in \{"DENY", "SAMEORIGIN", \phi\}$ $"DENY" \sqsubseteq "SAMEORIGIN" \sqsubseteq \phi$	
`Set-Cookie: key=value. . .` a, b	$\langle a, b \rangle$	$a, c \in \{"secure", \phi\}, "secure" \sqsubseteq \phi,$ $b, d \in \{"httpOnly", \phi\}, "httpOnly" \sqsubseteq \phi$	
`X-XSS-Protection:` a, b		$a, c \in \{"1", "0", \phi\}, "1" \sqsubseteq \phi \sqsubseteq "0"$ $b, d \in \{"mode = block", \phi\}, "mode = block" \sqsubseteq \phi$	
`Strict-Transport-Security: max-age=`a, b, c	$\langle a, b, c \rangle$	$a, d \in \mathbb{N}, a \sqsubseteq d \iff a \geq d$ $b, e \in \{"includeSubDomains", \phi\}, "includeSubDomains" \sqsubseteq \phi$ $c, f \in \{"preload", \phi\}, "preload" \sqsubseteq \phi$	
`Public-Key-Pins: max-age=`a, b, c		$a, d \in \mathbb{N}, a \sqsubseteq d \iff a \geq d$ $b, e \in \{"includeSubDomains", \phi\}, "includeSubDomains" \sqsubseteq \phi$ $c, f \in \mathcal{P}(\mathrm{KP}) \setminus \{\}$	

The use of 'strict-dynamic' disables a CSP's whitelist, 'unsafe-inline' and does not block script execution except for HTML parser-inserted scripts. Parser-inserted scripts are only allowed in combination with a valid nonce or hash. Therefore we also need to remove any occurrence of 'strict-dynamic' from CSPs. We apply the same rules as for nonces but also add the 'unsafe-eval' flag because to ensure scripts using eval and eval-like functions can execute normally as in the presence of 'strict-dynamic'.

With these transformations, we can reuse the formalism by Calzavara et al. without any modifications.

4.2 $\sqcup$ and $\sqcap$ for policy combination

When given two policies for a security mechanism, e.g. p_1 = "`a.com b.com`" and p_2 = "`a.com c.com`" for the `Timing-Allow-Origin` security mechanism, we have several options to combine them.

We can combine two security policies, so that the result allows the union of what both policies allow. This combination would weaken both policies and is called the $\sqcup$ operation. In the example, the result of $p_1 \sqcup p_2$ is "`a.com b.com c.com`".

We can also combine two security policies, so that the result disallows the union of what each policy disallows. In other words, the resulting policy would allow the intersection of what both policies allow. This combination would restrict or strengthen both policies and is called the $\sqcap$ operation. In the example, the result of $p_1 \sqcap p_2$ is "`a.com`".

The $\sqcup$ operation can be used to calculate what minimum security policy is currently enforced by the combination of the security policies of all web pages in a web origin. Enforcing this minimum security policy as the `baseline policy` would then not interfere with the security policies already in place for each individual web page.

The $\sqcap$ operation can be used to explicitly calculate the security policy that results from enforcing several security policies sequentially. For instance, when a server sends several CSP policies to the browser, the browser will consult each security policy sequentially and only allow certain behavior if all CSP policies allow it. In effect, the browser implicitly combined the policies with the $\sqcap$ operation.

For the enforcement of the origin manifest mechanism, we must explicitly calculate the result of the $\sqcap$ operation because not all security mechanisms perform this operation implicitly. For instance, when encountering two `Strict-Transport-Security` headers, the browser will enforce the first and ignore the second. For correct enforcement of the origin manifest mechanism, the second header must also be enforced. Therefore, we need to apply the $\sqcap$ operation explicitly.

When we extract a `baseline policy` from the same scenario with two security policies p_2 and p_3 in an HTTP response, we must then apply the $\sqcup$ operation with the current baseline p_1 after first explicitly applying the $\sqcap$ operation on both security policies, in essence computing: $p_1 \sqcup (p_2 \sqcap p_3)$.

Both the $\sqcup$ and $\sqcap$ operations are induced by the partial order $\sqsubseteq$, described in Section 4.1, as is standard in literature [10].

$p = p_1 \sqcap p_2$ if

$$\begin{cases} p \sqsubseteq p_1 \quad \text{and} \quad p \sqsubseteq p_2 \\ \forall x.\, x \sqsubseteq p_1 \quad \text{and} \quad x \sqsubseteq p_2 \implies x \sqsubseteq p \end{cases}$$

$p = p_1 \sqcup p_2$ if

$$\begin{cases} p_1 \sqsubseteq p \quad \text{and} \quad p_2 \sqsubseteq p \\ \forall x.\, p_1 \sqsubseteq x \quad \text{and} \quad p_2 \sqsubseteq x \implies p \sqsubseteq x \end{cases}$$

Note that $\sqcup$ and $\sqcap$ are intentionally undefined when HTTP headers cannot be combined into a single header. Formally, the reason is that the partial order $\sqsubseteq$ does not form a lattice [10], which we demonstrate on the respective examples for $\sqcup$ and $\sqcap$.

For $\sqcup$, consider CSP policies csp_1 ="`script-src a.com`" and csp_2 ="`script-src 'strict-dynamic' 'nonce-FOO='`". Policy csp_1 only allows scripts from `a.com` whereas csp_2 allows any script with a valid nonce and any script loaded from a script with a valid nonce. Policies csp_1 and csp_2 cannot be merged into a single header using the $\sqcup$ operation: CSP ignores whitelists in the presence of `strict-dynamic` for csp_2, but would at the same time have to guarantee that scripts are only loaded from `a.com` for csp_1.

For $\sqcap$, consider `Public-Key-Pins` policies "`pin-sha256="pin1"; max-age=42`" and "`pin-sha256="pin2"; max-age=42`". By definition there should be no public key for which both fingerprints are valid. Similarly to the $\sqcup$ above, note that the result of the $\sqcap$ in this case is intentionally undefined. Introducing a bottom element in the partial order as a result of the $\sqcap$ would be inappropriate, as the goal for this case is to flag an anomaly for developers rather than returning an overly prohibitive result that no communication is allowed.

5 PROTOTYPE IMPLEMENTATIONS

To determine the feasibility of the origin manifest mechanism, we created prototype implementations of the $\sqcup$ and $\sqcap$ combinators, the client-side enforcement mechanism, the server-side manifest handling as well as and automated manifest learning tool on the server-side. These implementations are described in this section.

5.1 Client-side enforcement

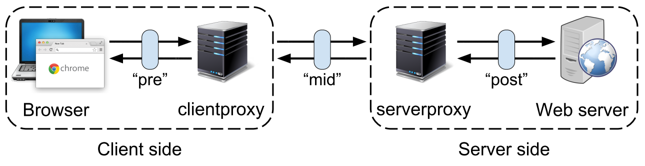

Figure 1: Architectural overview of our prototype implementations. The three measurement points "pre", "mid" and "post" are used during the evaluation only (Section 6.3).

As described in Section 3, the origin manifest describes origin-wide security settings for a web origin, and is stored in a file on the server side. The application of these security settings happens on the client-side, ideally in the user's browser.

The source code for a browser, such as e.g. Chromium, contains millions of lines of C++ code [4]. Modifying this source code to implement a new security mechanism is a difficult task. Because we are only interested in studying the feasibility of the origin manifest mechanism, and in order to avoid the difficulties associated with modifying browser source code, we opted to implement the origin manifest mechanism as a client-side proxy instead. Besides reducing the complexity of the prototype implementation, another advantage of this setup is that it is independent of the browser used.

Our *clientproxy* is located on the client-side and intercepts all traffic from and to the browser, as seen in Figure 1. The clientproxy handles the origin manifest retrieval and application as described in Section 3:

- For requests from the browser towards a web server, the clientproxy adds a `Sec-Origin-Manifest` header to indicate the presence of the origin manifest mechanism and to communicate its version of the manifest file.
- For responses from the web server to the browser, the clientproxy interprets the origin manifest and applies it to the response headers, using the combinator functions. When the web server indicates the presence of a new origin manifest, the clientproxy retrieves the new version automatically and applies it to the current as well as future HTTP responses.
- Any CORS preflight requests sent by the browser that match the rules of the origin manifest, are also handled by the clientproxy without forwarding the request to the web server.

We implemented the clientproxy using mitmproxy v2.0.2 [9] as a mitmproxy addon script, using python v3.5.

5.2 Server-side manifest handling

As a complement to the clientproxy, we also implemented the origin manifest mechanism on the server-side. Instead of modifying the source code of any particular web server software, we chose to implement the server-side prototype as a proxy. This *serverproxy* is located on the server-side, intercepting and modifying any traffic to the web server, as seen in Figure 1.

The serverproxy has three functions:

- serve the origin manifest file to any web client requesting it,
- inform the web clients about the version of the latest origin manifest, through the `Sec-Origin-Manifest` header, and
- strip the HTTP response headers received from the web server according to the `fallback` section in the origin manifest to reduce bandwidth towards the web client.

Just like the clientproxy, the serverproxy was implemented as a mitmproxy v2.0.2 [9] addon script, using python v3.5.

5.3 Automated manifest generation from observed traffic

We implemented a prototype for an automated origin manifest generator which can be used to assist security officers with the creation of an origin manifest. Our implementation does not use any advanced AI or machine learning techniques to "learn". Instead, we apply a pragmatic approach to provide the security officer with a reasonable starting point. The manifest generator hooks into the serverproxy, observing and storing the HTTP headers for all HTTP requests and responses for the back-end web servers for which it is proxying traffic.

After a data collection phase, the manifest generator analyzes the observed HTTP headers and generates origin manifest files for all observed origins. Then the origin manifest mechanism is activated in the serverproxy, so that it will respond to requests related to the origin manifest mechanism such as manifest retrieval and sending the manifest version via the `Sec-Origin-Manifest` header.

The automated generation of the manifest consists of three parts:

- Firstly, the `fallback` section is generated by enumerating all HTTP headers and their values that occur in a certain ratio (`cutoff`) observed responses have in common. By default we use the `cutoff` value 51%. Multiple responses for the same requested URL are counted only once and only common headers and values are considered. To prevent origin manifest creation based on a single HTTP response, we disregard origins with less than `minsize` observed HTTP responses. By default we use the `minsize` of 2.
- Secondly, the `baseline` and `augmentonly` sections are generated by combining observed security headers and values from HTTP responses, using the $\sqcup$ operator described in Section 4.
- Lastly, the `unsafe-cors-preflight-with-credentials` and `cors-preflight` sections are generated from the observed HTTP requests and their responses.

Note that for the manifest generation, we only consider those headers that are applicable to the given origin and content-type. For instance, a CSP header set on an HTTP response which does not have a `Content-Type` of `text/html`, is ignored. Similarly, HSTS headers in HTTP responses on a non-HTTPS origin are ignored.

Some HTTP headers have a large impact on the functioning of HTTP itself and how resources are handled and displayed in the browser. Because it makes no sense to place these headers in the origin manifest, they were blacklisted for automated manifest generation. These headers are: `Content-Encoding`, `Content-Type`, `Content-Length` and `Content-Disposition`.

The automated origin manifest generator is implemented as part of a mitmproxy v2.0.2 addon script using python v3.5.

5.4 Limitations and considerations

The implementations of the clientproxy and serverproxy are fully functional, suffering only minor limitations:

First, we are unable to differentiate between authenticated and unauthenticated CORS preflight requests/responses for the specific case when the browser is using client-side SSL certificates for the given origin. This limitation is intrinsic to our setup using proxies breaking the SSL tunnel. Luckily, the use of client-side SSL certificates is not widespread on the Web [31]. Furthermore, implementing the origin manifest mechanism as a browser modification will not suffer from the same limitation.

Secondly, we must disable strict certificate checking (such as HPKP), simply because of our need to alter both HTTP and HTTPS traffic "in flight". This limitation is again intrinsic to our setup and is no longer an issue when the origin manifest is implemented as a browser modification.

Thirdly, we disable HTTP/2 support in mitmproxy, which it supports by default. Our implementations work with HTTP/2 just as well as with HTTP/1. However, HTTP/2 offers some improvements over HTTP/1 which we do not take advantage of in our prototypes as currently implemented.

Fourthly, our implementation does not limit itself to only HTTPS connections as required in Section 3. For this feasibility study, we do not wish to limit ourselves to only HTTPS, but are also interested to see how the origin manifest mechanism would behave for non-HTTPS origins.

Lastly, note that the origin manifest generator is a proof of concept tool to assist origin security officers in finding a good starting point for composing a meaningful origin manifest based on the currently hosted web applications. We recommend that security officers review generated origin manifests before deployment, and we do not advocate deploying this tool in production environments to generate origin manifests in "real time".

6 EVALUATION

We evaluate the origin manifest mechanism as well as our prototypes with several experiments.

Firstly, we evaluate that our prototypes are working properly and do not break web pages in unexpected ways.

Secondly, we perform a longitudinal experiment to determine if the application of the origin manifest mechanism is practical.

Thirdly, we evaluate the performance of the origin manifest mechanism by measuring its effect on network traffic during a large-scale experiment in which we apply the origin manifest mechanism to the Alexa top 10,000 domains.

All data sizes refer to the uncompressed data in bytes because of our experiment setup.

6.1 Functional evaluation

We evaluated the correctness of our implementation by manually inspecting a randomly chosen subset of the Alexa top 1 million domains and their respective websites, with and without origin manifest.

Fully automated testing to verify the correctness of the implementation was deemed impractical, because typical web pages are often dynamically generated with e.g. advertising, which makes it difficult for an algorithm to determine whether a web application is still operating and rendered correctly before and after application of the origin manifest mechanism. The use of an ad-blocker such as AdBlock [1], would alleviate some of these impracticalities. However, advertising is omni-present on the Web and removing it from the web traffic would interfere with the normal operations of web pages, and thus also with our testing of the origin manifest mechanism.

6.1.1 Setup. The evaluation progressed in two phases: an **interactive phase** and a **visual inspection phase**. Both these experiments used the setup as shown in Figure 1, where browser web traffic is forwarded through both the clientproxy and serverproxy. The interactive phase used a regular browser (Chrome version 63.0.3239.132) in incognito mode, operated by a human. The visual inspection phase used the same browser, but operated by Selenium 3.8.1 [30]. The results of this phase were human inspected.

Interactive phase. We randomly selected 100 domains from the Alexa top 1 million for this experiment. For each of these domains, we visited the top-most page, e.g. http://example.tld for the example.tld domain, and interacted with the web page, mimicking the behavior of a typical user without authenticating for that web site.

The clientproxy and serverproxy both respond to internal URLs that allow state inspection. These inspection tools were used to determine when enough data had been collected: we aimed to navigate on a web domain at least five times and gather data for at least ten web origins.

When enough data was collected, the origin manifest mechanism was activated in both proxies. The browser was then restarted to clear caches and the web pages visited again. During the second visit, we inspected the pages both visually, and tested the functionality of the web page by triggering menus, playing videos, and otherwise interacting with the web page as an ordinary visitor.

Visual inspection phase. We randomly selected another 1000 domains from the Alexa top 1 million for this experiment. Like in the interactive phase, we also visited the top-most page before and after the activation of the origin manifest mechanism.

However, in this visual inspection phase, we simply took a screenshot of the web page using Selenium, before and after activation of the origin manifest mechanism. The browser was restarted in between visits to clear any caches. We repeated these steps four times to have reliable results in the face of dynamic content, such as advertising, resulting in eight screenshots. The screenshots were combined into a single image with four rows of two images: the "before" and "after" screenshots side by side.

The resulting images were inspected visually one by one to determine whether web pages exhibited unusual rendering artifacts. Any images in which the screenshots appeared to differ before and after activation of the origin manifest mechanism, were put aside and their domains revisited using the same technique as in the "interactive phase".

6.1.2 Results. Our manual and visual inspections confirm that our implementations work correctly. From the 1100 domains we visited, we only encountered abnormal behavior in three cases. In each of these cases, the problem was due to the automated learner not receiving sufficient learning input, which could have been easily prevented by changing a parameter. As expected, the automated origin manifest learner and generator tool can be used as a good starting point to formulate an initial origin manifest, although we recommend that the generated manifest should still be reviewed by a human to ensure correct configurations.

6.2 Longitudinal study

We define the *stability* of a header as the average amount of time that we observe the header to be present and its value unchanged. For instance, a stability of 5 days indicates that the header was observed with the same value for an average of 5 days in a row. Likewise, the stability of a manifest file indicates the average lifetime of a manifest file.

The stability of HTTP headers has an impact on the `fallback` section in manifest files and their stability. To be usable in practice, manifest files should be as stable as possible to reduce network traffic and workload of the security officer.

By the size of a header, we mean the total amount of bytes it occupies including its header name.

We conducted a longitudinal study over 100 days to examine the frequency, stability and size of HTTP headers and auto-generated manifest files in the real world.

6.2.1 Setup. We used OpenWPM [11], which is based on Firefox, to visit a set of 1000 domains from the Alexa top 1 million.

The domain list consisted of the top 200 domains, 200 domains randomly picked from the top 201 − 1,000, 200 domains randomly picked from the top 1,001 − 10,000, 200 domains randomly picked from the top 10,001 − 100,000, and finally another 200 domains randomly picked from 100,001 − 1,000,000.

For each domain we visited its top-most page, e.g. for the example domain `example.tld` we visited `http://example.tld`. We set OpenWPM to collect all request and response headers and ran it daily between October 5th 2017 and January 12th 2018, for a total of 100 days. We did not use our origin manifest prototype implementation during data collection.

6.2.2 Results.

HTTP headers. In total we collected 12,322,019 responses over 100 days. We visited a total of 3,575,043 unique URLs (25,533 origins) of which 20,201 URLs (3,682 origins) where visited every day. We counted 2,423 different header names (case-insensitive).

We only consider the headers in responses for those URLs which were observed for every day in our experiment. The frequency of HTTP headers indicates how often they were observed in the combined set of all responses. The stability of headers is computed over all observed responses.

Table 2: Selection of popular headers and security headers with their popularity rank, frequency, average size (bytes) and stability (days).

rank	header	freq.	avg. size	stability
3	server	87.39%	16.13B	32.14d
8	accept-ranges	47.57%	18.03B	68.06d
9	connection	44.61%	19.68B	43.01d
10	x-firefox-spdy	43.55%	16.01B	62.07d
33	x-powered-by	5.96%	21.70B	34.77d
14	access-control-allow-origin	29.95%	32.03B	67.20d
15	x-content-type-options	25.33%	29.02B	77.10d
16	x-xss-protection	23.48%	28.06B	67.78d
19	timing-allow-origin	19.31%	22.31B	26.41d
24	set-cookie	11.63%	395.09B	1.32d
26	strict-transport-security	8.03%	52.52B	22.97d
32	x-frame-options	5.98%	24.15B	76.51d
42	content-security-policy	2.69%	566.50B	5.84d
380	public-key-pins	0.04%	191.25B	23.21d

Table 2 shows a selection of five popular HTTP headers, as well as all security headers relevant to origin manifest. The selected popular HTTP headers are potential candidates for use in origin manifest. We omitted headers such as `Date` and `Content-Type` which are highly response dependent. For each header, we list their observed frequency, stability and size. A longer list of the top 50 most popular headers can be found in Appendix A.

From these results, we can make two observations:

Firstly, some of the average header sizes are quite large. For instance, the `Set-Cookie`, `Content-Security-Policy` and `Public-Key-Pins` headers take up hundreds of bytes on average. This gives credence to the claim from the origin policy draft, that HTTP headers can occupy multiple KiB per request.
Secondly, some headers occur frequently and have a large stability. For instance, the `Server` header occurs in 87.39% of all observed HTTP responses and has a stability or average lifetime of 32.14 days. This evidence also helps support the claim from the origin policy draft that HTTP headers are often repeated.

Origin Manifests. We used the automated manifest generator (See Section 5.3) to create origin manifests for each day. As was the case before, we only used headers from responses for URLs which recurred every day.
The `minsize` parameter was kept to its default of 2 so that no origin manifests are generated based on less than two observed responses. We evaluated the effect of the `cutoff` parameter for values of 50%, 70% and 90%, indicating the minimum size of the majority of responses that must agree on a header value before it is adopted into the `fallback` section of the manifest.

Table 3: The average size in bytes, average stability and the amount of fully stable vs. total number of non-empty generated manifests, for automatically learned origin manifests for different `cutoff` parameter values.

cutoff	average size	average stability	stable vs. all manifests
50%	408.13B	17.87d	883 / 1500
70%	304.17B	18.40d	850 / 1494
90%	282.89B	17.21d	819 / 1493

Table 3 shows the average size and stability, as well as the number of fully (100 days) stable versus all generated non-empty manifests. To measure the individual influence of headers on the stability of manifests, we analyzed the stability of headers in the `fallback`, `baseline` and `augmentonly` sections of the generated manifests. For this analysis, we used `minsize` 2 and `cutoff` 50%.
Table 4 shows the results for the same selection of HTTP headers and the security headers as before. A longer list of the top 50 most popular headers for the `fallback` section can be found in Appendix A.
Both sections `unsafe-cors-preflight-with-credentials` and `cors-preflight` are not listed because of their low inclusion frequency in manifests: 0.07% and 0.66%, respectively.
Based on the results from this experiment, we again make some observations:
Firstly, the `cutoff` parameter affects the size and stability of auto-generated origin manifests, which indicates that the generated manifests should not be used as-is. We recommend a quality inspection by a security officer before putting an auto-generated origin manifest into production.
Secondly, the average stability of the generated origin manifests is around 18 days, which indicates that modifications to the origin manifest are only needed once in a while, reducing the workload of a security officer.

Table 4: Selection of popular headers and security headers with their popularity rank, occurrence frequency (%), average size (bytes) and average stability (days) for the `fallback`, `baseline` and `augmentonly` sections.

rank	header	freq.	avg. size	stability
`fallback (non-security headers)`				
1	server	86.11%	16.84B	31.70d
5	accept-ranges	58.46%	18.05B	50.33d
6	connection	55.87%	19.68B	38.71d
10	x-firefox-spdy	31.87%	16.02B	47.60d
24	x-powered-by	7.99%	22.73B	28.32d
`fallback (security headers)`				
13	*CORS headers*	23.54%	30.42B	60.31d
16	x-content-type-options	15.97%	29.08B	75.01d
22	strict-transport-security	9.81%	51.39B	39.11d
28	timing-allow-origin	5.55%	26.34B	33.62d
32	x-frame-options	4.92%	25.12B	67.42d
33	x-xss-protection	4.12%	32.01B	29.70d
49	content-security-policy	0.95%	693.30B	5.54d
201	public-key-pins	0.07%	210.49B	4.30d
`baseline`				
1	*CORS headers*	28.13%	76.40B	51.24d
2	x-content-type-options	18.18%	29.00B	66.24d
3	x-frame-options	13.45%	24.40B	83.86d
4	strict-transport-security	12.79%	48.25B	45.73d
5	x-xss-protection	10.19%	28.32B	78.81d
6	timing-allow-origin	6.25%	26.29B	48.78d
7	content-security-policy	2.58%	591.33B	13.71d
8	public-key-pins	0.09%	194.84B	49.50d
`augmentonly`				
1	set-cookie	15.21%	19.01B	44.33d

Thirdly, the average origin manifest is only a few hundred bytes in size, which is quite small in comparison to the content served by the typical web origin. This indicates that the incurred network traffic overhead may be manageable.

6.3 Performance measurement

The main goal of the origin manifest mechanism is to improve security. However, the volume of network traffic is increased by transmissions of the origin manifest file and the `Sec-Origin-Manifest` header, and decreased because of the removal of redundant headers and cached CORS preflight requests. This net change in network traffic may have an unintended overhead with a negative impact.
In this section we are interested in measuring the impact of the origin manifest mechanism on the volume of network traffic observed between client and server. Note that we are not concerned with runtime overhead because our proof-of-concept implementations are not implemented as a browser modification as discussed in Section 5.1.

6.3.1 Setup. For this experiment, we augment the setup as described in Section 5 with extra proxies between browser and clientproxy ("pre"), clientproxy and serverproxy ("mid"), and serverproxy and the Web ("post"). This setup is depicted in Figure 1. The extra proxies ("pre", "mid" and "post") only perform logging and allow us to make measurements about the web traffic before and after it is modified by the origin manifest mechanism.

Instead of visiting single web pages, we simulate web browsing sessions where a user visits multiple related web pages. We create the URLs in these web browsing session by querying Bing for the top 20 pages in each of the Alexa top 10,000 domains. A web browsing session is then the set of pages returned by Bing for a single top Alexa domain.

Using Selenium, we automate a Chrome browser to visit each URL in the web browsing session in turn. This process is repeated four times: first, we visit the URLs just after clearing the browser cache ("before-uncached"), followed by a second visit where we do not clear the browser cache ("before-cached"). These first two phases serve to train the automated learner. Then, we instruct the server-proxy to generate origin manifests as described in Section 5.3 and the origin manifest mechanism is activated. We clear the browser cache and visit the URLs again ("after-uncached") and then a final time without clearing the cache ("after-cached"). These four different phases are designed to measure traffic before and after the application of the origin manifest, as well as the impact of the browser cache on the volume of web traffic.

The measurement proxies ("pre", "mid" and "post") record the HTTP headers of all requests and responses in each of the four phases of the experiment. Because of remote network failures, it is possible that some URLs in a web browsing session can not load. We limit ourselves to only those web browsing sessions that were able to successfully visit all the URLs. Furthermore, because of dynamic content such as advertising, the web resources loaded during a web browsing session can differ. For our statistics, we only consider those resources that were loaded in all four phases of a web browsing session.

6.3.2 Results. Bing returned 180,831 URLs of which 180,443 were unique, resulting in an average of 18.04 URLs per Alexa domain and web browsing session.

From the 10,000 top Alexa domains we intended to use as a basis for creating web browsing sessions, only 8,983 were usable. The remaining 1,017 domains did not yield any URLs from Bing, or their respective web browsing session did not deliver reliable results over all four phases of the experiment.

The results of this measurement study are shown in Table 5.

On the first visit, without using previously cached web traffic, we measured a total traffic of 34.3MiB on average per web browsing session, of which 2.1MiB is occupied by HTTP headers and 2.5KiB by CORS preflight traffic. After application of the origin manifest mechanism, we see an average of 128.5KiB of web traffic related to the retrieval of origin manifests files, which includes the `Sec-Origin-Manifest` header in all requests and responses.

As expected, the volume of network traffic for the HTTP headers decreases both because of the use of the origin manifest, and also because of the browser cache. Without the browser cache, the header-only traffic decreases from 2.1MiB to 1.8MiB after application of the origin manifest mechanism, which is a reduction of 13.84%. When using the browser cache, the header-only traffic is first reduced by 10.95% to 1.9MiB, and by 24.00% to 1.6MiB after application of the origin manifest mechanism.

Table 5: Average volume of web traffic measured for the 8,983 web browsing sessions, before and after application of the origin manifest mechanism, without ("uncached") and with ("cached") using the browser cache. Percentages are calculated per row, in relation to the uncached traffic before application of the origin manifest mechanism.

Traffic type	uncached		cached	
	Without origin manifest			
Headers only	2.1MiB	(100.00%)	1.9MiB	(89.05%)
Origin manifests	–	(–)	–	(–)
CORS preflights	2.5KiB	(100.00%)	2.2KiB	(85.88%)
Total	34.3MiB	(100.00%)	27.6MiB	(80.57%)
	With origin manifest			
Headers only	1.8MiB	(86.16%)	1.6MiB	(76.00%)
Origin manifests	128.5KiB	(–)	78.5KiB	(–)
CORS preflights	470.1B	(18.13%)	421.0B	(16.23%)
Total	34.0MiB	(99.28%)	27.3MiB	(79.81%)

The traffic overhead generated by the origin manifest mechanism is due to the transmission of origin manifest files as well as the `Sec-Origin-Manifest` header in requests and responses. We measured an average of 128.5KiB during the uncached phase, which is reduced to 78.5KiB after the browser cache is activated and the browser already has the latest version of each origin manifest file cached.

Requests and responses for CORS preflights before application of the origin manifest mechanism amount to 2.5KiB and 2.2KiB (85.88%) for uncached and cached respectively. This volume of traffic is reduced by 81.87% to 470.1B and by 83.77% to 421.0B for uncached and cached respectively, when the origin manifest mechanism is in use.

All in all, the total size of all web traffic observed throughout a web browsing session, drops from 34.3MiB by 0.72% to 34.0MiB due to application of the origin manifest mechanism, and from 27.6MiB to 27.3MiB (80.57% to 79.81%) when the browser cache is used.

7 DISCUSSION

The introductory example use case in Section 1 highlighted the need for a mechanism such as origin manifest, which the web security community is currently drafting. With our evaluation of this draft, we answer some research questions in order to justify and improve the origin policy's standard draft proposal.

Through our prototype implementation, we evaluated the mechanism in practice and conclude that it is possible to deploy without breaking any websites in unexpected ways. The prototype in form of web proxies indicates that adoption by actual browsers is indeed feasible.

Our large-scale studies confirm suspicions in the standard draft that using origin manifests in a real world setting does reduce the amount of network traffic. But the reduction is rather insignificant in practise. These large-scale studies also showed that origin manifests can be generated in an automated way by observing and learning from web traffic for a particular web origin. The auto-generated manifests serve as a good starting point for an origin security officer to formulate and fine-tune an origin manifest. We

remind however, that our automated origin manifest generator is only a proof of concept tool and we recommend human inspection of its output before deployment. Furthermore, the results from our experiments show that the auto-generated origin manifests do not change too often over time. The average stability of around 18 days thus makes the origin manifest mechanism usable in practice.
Our practical evaluation of the standard draft revealed two oversights in the draft proposal which should be addressed to make the origin manifest mechanism more robust and practical. First, the standard draft does not explicitly specify how to resolve conflicts between security policies set in the origin manifest by the origin security officer, and security policies set by the web developer on individual web pages. To this end we formalized the rules governing the comparison and combination of security policies. Second, we realized that the baseline policies in the origin manifest do not work well for e.g. cookies, which motivated us to introduce `augmentonly` policies. With both these extensions we actively contribute to improving the design and practicality of origin manifest.

8 RELATED WORK

Our work is based on the origin policy proposal which currently exists as a standard draft [41] accompanied by an explainer document [16]. The formalism for CSP is taken from the work by Calzavara et al. [6]. In this section we discuss other works and technologies, and their relation to the origin manifest mechanism.

Site-Wide HTTP Headers. Mark Nottingham's proposal of Site-Wide HTTP Headers [24] has many similarities with the origin policy. In fact, his draft and input have influenced the origin policy draft as mentioned in the draft's acknowledgments. Due to the many similarities of both proposals we believe that our results are also equally insightful to both the work on Site-Wide HTTP Headers as well as origin policy.

Web App Manifest. The Web App Manifest [37] is an upcoming standard to configure web applications and to define for example name, icons and other layout options. It stands to reason to consider integrating the features of origin manifest into Web App Manifest. However there are fundamental differences between both technologies. For example, Web App Manifest allows developers to configure a web application, origin manifest sets a configuration for the entire web origin. Another example is that Web App Manifests can be downloaded out-of-band. Origin manifests must always be fetched before actual content is loaded because the security configurations might affect current and subsequent resource fetches.

Server-side configuration. Web application configuration files like ASP.NET's `Web.config` are written by web application developers for a specific web application, not an entire web origin as origin manifests. Note that the origin manifest mechanism does not try to replace any web application specific configuration mechanisms but adds a way for the origin to express its own requirements.
Server configurations, like for an Apache server, are not necessarily per origin. Nevertheless, one could achieve the same effects as with an origin manifest through server configurations or server-side proxies which enforce, for example, the presence of certain HTTP headers or specific header values. Servers and proxies can set response specific values, for example CSP nonces, which is not meaningful in the context of an origin manifest. The advantage of the origin manifest mechanism is that it provides a mechanism independent of the concrete server-side architecture and requires only minimal changes for deployment. With our combinator functions the origin manifest mechanism does not conflict with server- and response-specific configurations.

Security evaluation. There are several empirical studies which analyze the deployment of security mechanisms on the web [2, 17, 21, 31, 39, 42]. Our work distinguishes from theirs in that we do not analyze the usage of particular security mechanisms, but extract security related headers solely for the automated generation of origin manifests. We do not evaluate the quality of the particular security policies themselves.

HTTP performance. In order to improve network performance, different HTTP compression methods have been proposed both in academia [5, 22, 32, 41] and industry with HTTP/2 [23]. HTTP/2's header compression removes the redundancy of sending the same header again and again. The origin manifest mechanism can also be used to reduce the sending of headers in every response to the client through the `fallback` section. However the origin manifest mechanism's primary goal is not to improve performance but to raise the security level of an entire web origin.
There are also other HTTP performance improvements like the `ETag` cache control mechanism [15], which are addressed in the origin manifest draft [16].

Automated policy generation. Automated generation of policies from existing setups is not a novel idea. E.g. there exist several solutions to find a suitable CSP [14, 20, 27]. The purpose of these tools is to generate a policy when none exists yet. The purpose of the automated origin manifest generator is to generate an origin manifest from already existing policies.

9 CONCLUSION

We provide a first evaluation of the origin manifest mechanism from a current standard draft to enforce origin-wide configurations in browsers. Our evaluation has helped us identify inconsistencies in the draft, leading us to propose a systematic approach to comparing and combining security policies, including general join and meet combinators, as well as augmentonly policies addressing corner cases.

We formally define rules to compare and merge HTTP security policies, which serves as the basis for a client-side enforcement mechanism, a server-side implementation, and an automated origin manifest generation tool.

We use our prototype implementations to evaluate the origin manifest mechanism in a 100-day longitudinal study of popular websites, and a large-scale performance evaluation study on the Alexa top 10,000.

We find that the origin manifest mechanism is an effective way of raising the security level of a web origin and that the origin manifest for a typical origin is stable enough to be of practical use. As a bonus benefit, the origin manifest mechanism slightly reduces the amount of network traffic.

Acknowledgments. We thank Mike West for the discussion about the current state and the future of the origin manifest standard draft. Special thanks from Daniel to Mike are for the collaboration on initializing the origin manifest implementation for the Chrome browser as an internship project at Google. We also thank the anonymous reviewers for their useful comments and suggestions. This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

REFERENCES

[1] AdBlock. https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom. Last accessed: June 2018.

[2] Amann, J., Gasser, O., Scheitle, Q., Brent, L., Carle, G., and Holz, R. Mission accomplished?: HTTPS security after diginotar. In *IMC* (2017), ACM, pp. 325–340.

[3] Barth, A. HTTP State Management Mechanism. RFC 6265, 2011.

[4] Black Duck Software. Chromium (google chrome) project summary. https://www.openhub.net/p/chrome. Last accessed: June 2018.

[5] Butler, J., Lee, W.-H., McQuade, B., and Mixter, K. A Proposal for Shared Dictionary Compression over HTTP. https://lists.w3.org/Archives/Public/ietf-http-wg/2008JulSep/att-0441/Shared_Dictionary_Compression_over_HTTP.pdf. Last accessed: June 2018.

[6] Calzavara, S., Rabitti, A., and Bugliesi, M. CCSP: controlled relaxation of content security policies by runtime policy composition. In *USENIX Security Symposium* (2017).

[7] Calzavara, S., Rabitti, A., and Bugliesi, M. Semantics-Based Analysis of Content Security Policy Deployment. *ACM Transactions on the Web (TWEB)* (2018).

[8] Chris Palmer. Intent To Deprecate And Remove: Public Key Pinning. https://groups.google.com/a/chromium.org/d/msg/blink-dev/he9tr7p3rZ8/eNMwKPmUBAAJ. Last accessed: June 2018.

[9] Cortesi, A., Hils, M., Kriechbaumer, T., and contributors. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/, 2010–. Version 2.0.2, Last accessed: June 2018.

[10] Donnellan, T. *Lattice Theory*. Pergamon, 1968.

[11] Englehardt, S., and Narayanan, A. Online tracking: A 1-million-site measurement and analysis. In *ACM Conference on Computer and Communications Security* (2016), ACM, pp. 1388–1401.

[12] Evans, C., Palmer, C., and Sleevi, R. Public Key Pinning Extension for HTTP. RFC 7469, 2015.

[13] Exploit Database. Apache Tomcat 3.2.1 - 404 Error Page Cross-Site Scripting. https://www.exploit-db.com/exploits/10292/. Last accessed: June 2018.

[14] Fazzini, M., Saxena, P., and Orso, A. Autocsp: Automatically retrofitting CSP to web applications. In *ICSE (1)* (2015), IEEE Computer Society, pp. 336–346.

[15] Fielding, R., and Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. RFC 7232, 2014.

[16] Hausknecht, D., and West, M. Explainer: Origin-wide configuration using Origin Manifests. https://github.com/WICG/origin-policy, 2017. Last accessed: June 2018.

[17] Helme, S. Alexa Top 1 Million Analysis - August 2017. https://scotthelme.co.uk/alexa-top-1-million-analysis-aug-2017/, 2017. Last accessed: June 2018.

[18] Hodges, J., Jackson, C., and Barth, A. HTTP Strict Transport Security (HSTS). RFC 6797, 2012.

[19] Java EE Grizzly NIO. Standard error pages of grizzly-http-server allow cross site scripting. https://github.com/javaee/grizzly/issues/1718. Last accessed: June 2018.

[20] King, A. Laboratory (Content Security Policy / CSP Toolkit). https://addons.mozilla.org/en-US/firefox/addon/laboratory-by-mozilla/. Last accessed: June 2018.

[21] Kranch, M., and Bonneau, J. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *NDSS* (2015).

[22] Liu, Z., Saifullah, Y., Greis, M., and Sreemanthula, S. HTTP compression techniques. In *WCNC* (2005).

[23] M. Belshe, R. Peon, M. T. Hypertext transfer protocol version 2 (http/2). RFC 7540, 2015.

[24] Nottingham, M. Site-wide http headers. https://mnot.github.io/I-D/site-wide-headers/, 2017. Last accessed: June 2018.

[25] Nottingham, M., and Hammer-Lahav, E. Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785, 2010.

[26] OwnCloud. XSS in Error Page. https://owncloud.org/security/advisories/xss-in-error-page/, 2017. Last accessed: June 2018.

[27] Pan, X., Cao, Y., Liu, S., Zhou, Y., Chen, Y., and Zhou, T. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *ACM Conference on Computer and Communications Security* (2016), ACM, pp. 653–665.

[28] plentz. Best nginx configuration for improved security (and performance). https://gist.github.com/plentz/6737338. Last accessed: June 2018.

[29] Ross, D., Gondrom, T., and Stanley, T. HTTP Header Field X-Frame-Options. RFC 7034, 2013.

[30] SeleniumHQ – Browser Automation. http://www.seleniumhq.org. Last accessed: June 2018.

[31] Van Acker, S., Hausknecht, D., and Sabelfeld, A. Measuring login webpage security. In *SAC* (2017).

[32] van Hoff, A., Douglis, F., Krishnamurthy, B., Goland, Y. Y., Hellerstein, D. M., Feldmann, A., and Mogul, J. Delta encoding in HTTP. RFC 3229, 2002.

[33] W3C Web Application Security Working Group. Content security policy level 2, 2016.

[34] W3C Web Application Security Working Group. Content security policy level 3, 2016.

[35] W3C Web Browser Performance Working Group. Resource Timing. https://w3c.github.io/resource-timing/, 2017. Last accessed: June 2018.

[36] W3C Web Hypertext Application Technology Working Group. CORS protocol. https://fetch.spec.whatwg.org/, 2017. Last accessed: June 2018.

[37] W3C Web Platform Working Group. Web app manifest. https://w3c.github.io/manifest/, 2017. Last accessed: June 2018.

[38] Web Incubator CG. Origin Policy Issues. https://github.com/WICG/origin-policy/issues. Last accessed: June 2018.

[39] Weissbacher, M., Lauinger, T., and Robertson, W. K. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID* (2014).

[40] West, M. Chromium bug 751996 - Origin Policy. https://bugs.chromium.org/p/chromium/issues/detail?id=751996, 2017. Last accessed: June 2018.

[41] West, M. Origin Manifest. https://wicg.github.io/origin-policy/, 2017. Last accessed: June 2018.

[42] Zhou, Y., and Evans, D. Why aren't HTTP-only cookies more widely deployed. In *W2SP* (2010).

A STATISTICAL DATA

Table 6: The top 50 most popular HTTP headers with rank, occurrence frequency (%), average size (bytes) and stability (days).

rank	header	freq.	avg. size	stability
1	date	98.90%	33.00B	1.04d
2	content-type	95.94%	27.93B	81.80d
3	server	87.39%	16.13B	32.14d
4	content-length	85.57%	17.74B	18.11d
5	cache-control	80.77%	36.54B	11.63d
6	expires	66.11%	35.44B	1.33d
7	last-modified	64.09%	42.04B	10.99d
8	accept-ranges	47.57%	18.03B	68.06d
9	connection	44.61%	19.68B	43.01d
10	x-firefox-spdy	43.55%	16.01B	62.07d
11	etag	43.07%	26.59B	10.84d
12	content-encoding	35.39%	20.00B	55.19d
13	vary	34.10%	19.80B	51.40d
14	access-control-allow-origin	29.95%	32.03B	67.20d
15	x-content-type-options	25.33%	29.02B	77.10d
16	x-xss-protection	23.48%	28.06B	67.78d
17	age	22.90%	8.10B	1.16d
18	p3p	19.54%	98.52B	59.74d
19	timing-allow-origin	19.31%	22.31B	26.41d
20	alt-svc	18.14%	140.63B	22.20d
21	pragma	17.03%	13.79B	68.83d
22	x-cache	15.54%	19.72B	11.80d
23	via	12.84%	50.53B	2.48d
24	set-cookie	11.63%	395.09B	1.32d
25	cf-ray	8.50%	26.00B	1.05d
26	strict-transport-security	8.03%	52.52B	22.97d
27	cf-cache-status	7.51%	19.01B	8.43d
28	transfer-encoding	6.98%	24.00B	27.23d
29	keep-alive	6.81%	24.70B	2.79d
30	location	6.44%	124.62B	5.50d
31	access-control-allow-credentials	6.17%	36.08B	71.68d
32	x-frame-options	5.98%	24.15B	76.51d
33	x-powered-by	5.96%	21.70B	34.77d
34	x-amz-cf-id	5.03%	67.00B	1.06d
35	access-control-allow-methods	5.01%	42.23B	68.29d
36	content-disposition	4.76%	50.42B	51.31d
37	x-served-by	4.01%	39.00B	1.19d
38	access-control-allow-headers	3.64%	81.51B	88.67d
39	x-cache-hits	3.35%	16.05B	1.49d
40	access-control-expose-headers	3.30%	64.24B	58.63d
41	x-timer	3.19%	33.23B	1.06d
42	content-security-policy	2.69%	566.50B	5.84d
43	x-amz-request-id	2.45%	32.30B	2.62d
44	x-varnish	2.44%	25.79B	1.27d
45	x-amz-id-2	2.43%	85.90B	2.62d
46	x-fb-debug	2.36%	98.00B	1.00d
47	content-md5	2.24%	35.08B	3.53d
48	cf-bgj	1.71%	13.49B	38.81d
49	cf-polished	1.71%	29.87B	38.76d
50	fastly-debug-digest	1.60%	83.00B	21.33d

Table 7: The top 50 most popular headers for origin manifest `fallback` section with rank, occurrence frequency (%), average size (bytes) and stability (days).

rank	header	freq.	avg. size	stability
1	server	86.11%	16.84B	31.70d
2	date	74.18%	33.00B	1.13d
3	content-type	70.74%	29.46B	50.60d
4	cache-control	70.07%	35.13B	24.33d
5	accept-ranges	58.46%	18.05B	50.33d
6	connection	55.87%	19.68B	38.71d
7	content-encoding	51.78%	20.00B	57.17d
8	vary	48.47%	20.36B	42.87d
9	expires	38.80%	35.17B	1.65d
10	x-firefox-spdy	31.87%	16.02B	47.60d
11	last-modified	30.84%	41.97B	5.74d
12	content-length	29.36%	17.25B	8.02d
13	access-control-allow-origin	23.54%	30.42B	60.31d
14	x-cache	18.74%	17.69B	11.03d
15	etag	17.32%	28.01B	4.81d
16	x-content-type-options	15.97%	29.08B	75.01d
17	p3p	14.45%	83.32B	69.90d
18	transfer-encoding	13.14%	24.00B	22.62d
19	via	12.91%	39.40B	3.00d
20	set-cookie	11.26%	232.79B	1.34d
21	pragma	11.19%	13.66B	60.42d
22	strict-transport-security	9.81%	51.39B	39.11d
23	cf-cache-status	8.73%	19.18B	9.08d
24	x-powered-by	7.99%	22.73B	28.32d
25	age	7.32%	6.60B	2.02d
26	alt-svc	6.02%	117.99B	18.36d
27	access-control-allow-methods	5.89%	41.27B	56.22d
28	timing-allow-origin	5.55%	26.34B	33.62d
29	access-control-allow-credentials	5.15%	36.22B	57.81d
30	keep-alive	5.08%	23.30B	10.90d
31	access-control-allow-headers	4.97%	74.45B	78.38d
32	x-frame-options	4.92%	25.12B	67.42d
33	x-xss-protection	4.12%	32.01B	29.70d
34	location	4.07%	71.97B	3.49d
35	access-control-expose-headers	2.61%	74.06B	49.50d
36	x-cache-hits	2.44%	14.85B	3.46d
37	access-control-max-age	2.31%	26.46B	81.00d
38	x-served-by	2.27%	35.32B	1.87d
39	x-amz-cf-id	1.82%	67.00B	1.20d
40	content-language	1.66%	19.61B	21.05d
41	cf-ray	1.65%	26.00B	1.05d
42	x-amz-id-2	1.24%	85.98B	3.04d
43	x-amz-request-id	1.24%	31.99B	3.04d
44	x-aspnet-version	1.23%	24.99B	70.71d
45	cf-bgj	1.19%	13.61B	49.50d
46	x-timer	1.08%	33.83B	1.30d
47	x-ua-compatible	1.07%	28.04B	68.54d
48	x-varnish	0.96%	25.11B	1.19d
49	content-security-policy	0.95%	693.30B	5.54d
50	link	0.93%	141.77B	8.16d