Reining in the Web's Inconsistencies with Site Policy

Stefano Calzavara*, Tobias Urban†‡, Dennis Tatang‡, Marius Steffens§, and Ben Stock§
*Università Ca' Foscari Venezia: calzavara@dais.unive.it
†Institute for Internet Security: urban@internet-sicherheit.de
‡Ruhr University Bochum: dennis.tatang@rub.de
§CISPA Helmholtz Center for Information Security: {marius.steffens,stock}@cispa.saarland

***Abstract*—Over the years, browsers have adopted an ever-increasing number of client-enforced security policies deployed through HTTP headers. Such mechanisms are fundamental for web application security, and usually deployed on a per-page basis. This, however, enables inconsistencies, as different pages within the same security boundaries (in form of origins or sites) can express conflicting security requirements. In this paper, we formalize inconsistencies for cookie security attributes, CSP, and HSTS, and then quantify the magnitude and impact of inconsistencies at scale by crawling 15,000 popular sites. We show that numerous sites endanger their own security by omission or misconfiguration of the aforementioned mechanisms, which lead to unnecessary exposure to XSS, cookie theft, and HSTS deactivation. We then use our data to analyse to which extent the recent *Origin Policy* proposal can fix the problem of inconsistencies. Unfortunately, we conclude that the current Origin Policy design suffers from major shortcomings which limit its practical applicability to address security inconsistencies while catering to the need of real-world sites. Based on these insights, we propose *Site Policy*, designed to overcome Origin Policy's shortcomings and make any insecurity explicit. We make a prototype implementation of Site Policy publicly available, along with a supporting toolchain for initial policy generation, security analysis, and test deployment.**

I. Introduction

Web applications are the key access point to a plethora of online services which we use daily. However, they are also notoriously hard to secure, given the increasing amount and complexity of involved technologies [34]. Browsers implement an ever-increasing amount of server-specified, yet *client-enforced*, security policies to support secure Web application development. These policies are typically deployed through HTTP headers. Prominent examples of such security policies include cookie security attributes [21], *Content Security Policy* (CSP) [40], and *HTTP Strict Transport Security* (HSTS) [14]. Modern Web applications cannot be deemed secure unless such mechanisms are set up and correctly configured.

A key problem of existing client-side security policies is that they build on top of an extremely *fine-grained* enforcement model. Header-based security policies like CSP work at the granularity of individual HTTP responses, i.e., different pages within the security boundary of the same origin can deploy different security policies. While such expressiveness is sometimes useful in practice — since site operators might want to fine-tune security policies on different pages for generic reasons — this threatens their sites' security by allowing for inconsistencies.

To assess the dangers of such fine-grained configuration, our paper starts from an intuitive definition of *inconsistent policy*, a general notion formalizing the dangers coming from the different adoption of the same security mechanism on different pages. Based on our definition, we analyze real-world security policies collected by crawling 15,000 popular sites and quantify inconsistencies at scale. Our analysis highlights several dangerous or potentially insecure practices, providing the first experimental evidence of the need for site-wide security policies in the wild. In particular, we show that inconsistencies might harm the expected guarantees of cookies activating specific security attributes, introduce CSP loopholes enabling script injection on apparently secure sites, and entirely disable protection on HSTS-enabled sites.

Naturally, inconsistencies leading to security issues can be rectified by deploying an *origin-wide* (or even *site-wide*) policy on all pages. The *Origin Policy* (OP) mechanism has been recently proposed specifically for this task, towards saving bandwidth in header communication and mitigating the risk of deploying incorrect security policies on some pages, e.g., error pages [10]. However, OP is not yet implemented in commercial browsers and only received limited attention by the security community so far [36]. Based on the insights of our real-world measurement, we show that the identified inconsistencies can be mitigated by OP only to a very limited extent: the "single policy per origin" model advocated by OP does not match the expectations of real sites, which sometimes deploy multiple policies on the same origin. For example, we observe that 10% of the origins that we crawled deploy more than one CSP. Even worse, we show that the origin boundary of OP is insufficient to fix inconsistencies, e.g., 81% of the *sites* deploy HSTS inconsistently, yet cannot take advantage of OP to rectify this issue. Finally, we identify thousands of cases where inconsistencies are introduced by the omission of security headers. While we can only make educated guesses on whether such omissions are intended, their amount and security impact is concerning enough to question the header-based, opt-in security model of OP.

Based on our analysis, we propose *Site Policy* (SP), which is designed to implement robust countermeasures to the issues we identified and overcome the limitations of Origin Policy. SP provides support for multiple policies within the same origin and also allows for fixing inconsistencies across the whole site, while proposing an opt-out model for security exceptions, thus making any insecurity *explicit*. At the same time, SP centralizes

Network and Distributed Systems Security (NDSS) Symposium 2021
22-24 February 2021, San Diego, CA, USA
www.ndss-symposium.org

a site's entire security concept within a single manifest, which is amenable for automated security analysis. We make a prototype implementation of SP publicly available [8] to experiment with, along with a security analyzer and a policy aggregator designed to help site operators move to SP.

Contributions: To sum up, we contribute as follows:

1) We propose a general notion of inconsistent policy and present its practical application to a set of popular client-side security mechanisms (Section III);
2) We crawl the Tranco top 15,000 sites to quantify inconsistencies at scale. Our analysis shows that different types of inconsistencies arise in a significant fraction of the sites, potentially leading to major security threats (Section IV);
3) We use the collected data to question the effectiveness of the current OP proposal in different directions. This insight is important since OP is not yet implemented in commercial Web browsers; hence this is the right time to analyze and improve its design (Section V);
4) We propose Site Policy (SP) to address the shortcomings of OP for providing consistent levels of security on Web applications. We make a prototype implementation of SP publicly available, along with a supporting toolchain, to experiment with (Section VI).

II. Background and Problem Space

Our work aims to analyze the inconsistent usage of security mechanisms in the wild, particularly concerning cases where inconsistencies cause a security issue, and propose solutions. We use the term *inconsistency* to denote that a security mechanism is configured with different security levels within the same security boundary: Section III provides a precise formal definition, yet here we provide intuitive examples of inconsistencies to motivate our research.

A. Preliminaries

We start by introducing key terminology. First, we denote as a *site* a registerable domain (also called eTLD+1), like `example.com` or `bbc.co.uk`. Each site may have different *subdomains*, e.g., `www.example.com`, which can be hosted via HTTP or HTTPS, and on differing ports. The combination of protocol, subdomain (or hostname), and port constitutes an *origin* as defined by RFC 6454 [3]. One of the key security mechanisms on the Web is the well-known *Same-Origin Policy* (SOP), which ensures two pages with different origins are not allowed to access each other (e.g., through JavaScript). We use the term *page* when referring to an individual HTML document hosted at some URL. We denote as security boundaries an origin (for JavaScript and host-only cookies) as well as a site (for domain-bound cookies and HSTS).

In this paper, we are concerned about inconsistencies in the use of three core security mechanisms: cookie security attributes, CSP, and HSTS. These are supposed to be uniformly used across pages, which makes it easy to identify inconsistencies on a large scale: this is obvious by design for cookie security attributes and HSTS, while for CSP a recent study estimates that 85 % of the sites deploy the same CSP of their landing page on all pages [28]. More importantly, these mechanisms have been designed with clear threat models in mind, which means that one can rigorously reason about the security implications of inconsistencies.

B. Cookie Security Attributes

The HTTP protocol is stateless and the solution to allow for state over HTTP are cookies [2]. Cookies are small pieces of text communicated via an HTTP header or set via JavaScript, which map a key to a value and have optional attributes. Once set, they are stored by the browser and sent to the server in each request that matches the cookie's scope. This scope is defined by the hostname a cookie is associated with and the request path. It is important to note that any cookie set with an explicit `Domain` attribute is valid for all subdomains of that domain. As an example, if a cookie is set as `key=value; Path=/mail; Domain=example.com`, it is sent with all requests to any subdomain of (and including) `example.com`, but only if the requested path is prefixed with `/mail`. We refer to cookies with the `Domain` attribute as *domain cookies*, while the others are referred to as *host-only cookies*.

As cookies are the primary means of managing sessions and other sensitive, client-side persisted information, they come with three important security attributes. `HttpOnly` ensures that cookies are not accessible from JavaScript (to disallow cookie theft through malicious JavaScript). Cookies can be flagged as `Secure`, meaning they must not be sent in unencrypted HTTP requests. The latest addition to the set of attributes is called `SameSite`, which is meant to protect against *Cross-Site Request Forgery* (CSRF) attacks. If this is set to `lax`, cookies are only sent on cross-site requests when the main frame is navigated in a safe way (e.g., through a GET request, not through POST). In case of `strict`, cookies are never sent across sites.

Each of the security attributes protects cookies from a particular type of threat (JavaScript-based cookie stealing, network sniffing, and CSRF attacks). Hence, whenever the same cookie is set with different security attributes on different pages, there is an inconsistency in protection. While we cannot ascertain if the cookie needed protection in the first place, the developer's intent indicates a certain level of protection, i.e., the strictest set of security attributes found for the cookie. Note that browsers overwrite the attributes when the same cookie is set multiple times. As an example, if a page sets a session cookie as both `HttpOnly` and `Secure`, but another page sets the same cookie just as `Secure`, the cookie will be exposed to JavaScript after visiting the second page, allowing an attacker to steal it from any page which can access it. Hence, to ensure the security of cookies against the outlined attacks, all the cookie setting operations need to consistently use the same security attributes. Note that, although the omission of a security attribute might be intentional, it is nevertheless dangerous. For example, an XSS in a page setting the `HttpOnly` attribute on a cookie can be used to steal the cookie from an XSS-free same-origin page that does not set the attribute (using frames or popups).

C. Content Security Policy

Content Security Policy (CSP) [33] is a security mechanism originally aimed at mitigating the severe dangers of Cross-Site Scripting (XSS) attacks. While CSP nowadays has numerous

ingredients to control content inclusion, restrict framing, or enforce TLS [40], we focus here on its original goal, namely XSS mitigation. In essence, a CSP is meant to ensure that only resources explicitly allowed by the developer of a page can be included therein. This is achieved by binding directives of the form `type-src` to a set of allowed sources, which can be as specific as a full URL or as unspecific as `https://*`. Script execution can also be controlled by allowing only script tags with a valid `nonce` attribute or matching a given hash.

Deploying a CSP with a `script-src` (or alternatively a `default-src`) directive implicitly disables a page's ability to run inline scripts, inline event handlers, and string-to-code transformation functions like `eval`. These can be allowed by specifying the `'unsafe-inline'` or `'unsafe-eval'` keywords, respectively. When `'unsafe-inline'` is used in combination with nonces / hashes, it is disregarded by modern browsers. Moreover, to support dynamic inclusion of scripts, scripts with a valid nonce can propagate trust to included scripts via the `'strict-dynamic'` keyword. This keyword voids any allowed hosts for script inclusion, forcing the exclusive use of nonces / hashes to control scripts.

As CSP is a client-side defense mechanism, it needs to be communicated to the browser. This is either done through the `Content-Security-Policy` HTTP header, or in a meta tag with the corresponding `http-equiv`. Since prior work has shown that such meta tags are rarely used in practice [28], we omit these from our inconsistency analysis.

Securely configuring CSP is not a trivial task, as evidenced by multiple studies highlighting the insecurity of deployed CSPs [28, 5, 38, 39]. For example, configuring an allowlist which contains the entire HTTPS scheme or allows arbitrary inline scripts through `'unsafe-inline'` (without nonces or hashes) is obviously unsafe, as the attacker can inject any script content. A notable issue of CSP is that, although it is enforced on a per-page basis, most browser assets (e.g., the Local Storage) are protected on a per-origin basis according to SOP: this means that even a single page with an unsafe CSP (or without CSP) might fully undermine a Web application's security, since the attacker could get scripting capabilities in the application's origin through that single page. Hence, any injection vulnerability on the page would grant access to same-origin content, e.g., cookies, Web storage, and the DOM; also, the attacker would be in the position to exfiltrate server-side secrets using XHR. Based on this, we stipulate that inconsistent adoption of CSP occurs when some pages deploy safe CSPs, while other pages do not.

D. HTTP Strict Transport Security

Even though the Web further progresses towards full usage of transport layer encryption through TLS, browsers do not yet forcefully upgrade all connections to HTTPS to avoid breakage. This introduces the danger of attackers who force a victim's browser to make a request towards the HTTP version of a site, allowing the attacker to freely manipulate the shown content, e.g., to attempt phishing, or to sniff cookies which are not marked `Secure`.

To allow operators to force HTTPS on their sites, *HTTP Strict Transport Security* (HSTS) [14] can be used. In particular, once set for a specific HTTPS host and until the `max-age` value contained in the header is reached, any connection towards that host is automatically upgraded to HTTPS by the browser. Optionally, HSTS can specify the `includeSubDomains` option, which extends protection to all the subdomains of the host setting the security header. This is important to defend against attackers who could otherwise forge cookies from HTTP subdomains and to prevent the exfiltration of domain cookies lacking the `Secure` attribute.

There are two potential inconsistencies in HSTS deployment with dangerous security implications: omitting the header or setting it with a non-positive value of `max-age` on some pages. Omitting the header is dangerous because it might mean that HSTS is sometimes not activated, particularly when no HSTS-protected page on the same host or a parent domain (in case of `includeSubDomains`) was visited. Even worse, setting the value of `max-age` to a non-positive value effectively disables HSTS for that host. Importantly, browsers implement this such that a potential entry for the host is dropped, rather than storing the opt-out. Hence, if the parent domain sets `includeSubDomains`, while the host itself sets `max-age` to 0, HTTP connections to the host are allowed until the parent is visited again, enforcing HTTPS for all its children.

III. Formalizing Inconsistencies

For all of the aforementioned security mechanisms, inconsistencies can cause severe security problems. In this section, we make this intuition more precise by formalizing a general notion of *inconsistent policy* to capture the dangers arising from using the same security mechanism at different security levels. We then show how this definition can be applied to the security mechanisms considered in our paper, which is useful to automate security analyses at scale. In particular, the Web measurement in Section IV utilizes the proposed inconsistency checks on policies collected from real-world sites.

A. Inconsistent Policies

In our formal model, we represent sites via a set of *objects* which require protection, e.g., cookies, and a set of *authorities* which declare security restrictions, e.g., Web pages. Security restrictions are modeled using *security labels*, which are a standard tool in access control and information flow control [20]. Therefore, we define a *policy* as follows:

Definition 1 (Policy)**.** A *policy* $P = (O, A, L, \{\rho_a\}_{a \in A})$ is a tuple comprising:

- a set of *objects* O requiring protection;
- a set of *authorities* A predicating over O;
- a set of *security labels* L;
- a partial function $\rho_a : O \to L$ for each $a \in A$, whose domain models the set of objects controlled by a.

To exemplify the definition at work, consider a simple access control scenario where a file f is owned by two parties a and b. In this scenario, we have $O = \{f\}$ and $A = \{a, b\}$. Assuming that files can be either public ($\bot$) or private ($\top$), we can model their confidentiality by defining $L = \{\bot, \top\}$. If both a and b agree that f should be private, we let $dom(\rho_a) = dom(\rho_b) = \{f\}$ and $\rho_a(f) = \rho_b(f) = \top$.

Given our notion of policy, the definition of *inconsistency* is very intuitive. Specifically, inconsistencies arise when two authorities specify different security restrictions on the same object. In our example, inconsistency would happen for instance if we had $\rho_a(f) = \bot$ and $\rho_b(f) = \top$, meaning that a requires f to be public and b requires f to be private. Accordingly, our definition of inconsistent policy is as follows:

Definition 2 (Inconsistency)**.** $P = (O, A, L, \{\rho_a\}_{a \in A})$ is *inconsistent* iff there exist an object $o \in O$ and two authorities $a, b \in A$ such that $o \in dom(\rho_a) \cap dom(\rho_b)$ and $\rho_a(o) \neq \rho_b(o)$.

This model is useful to reason on the secure deployment of client-side security policies on the Web since we explained that such policies are normally expressed through HTTP headers, meaning that different pages might express conflicting security requirements on the same object.

B. Applications

We now apply our formal framework to different security mechanisms where inconsistencies actually lead to significant security dangers. For each mechanism, we explain such dangers and present their formalization in our framework.

1) Cookies: Cookies can be protected against various attacks through appropriate security attributes. Unfortunately, since multiple pages can set the same cookie on the same site, different pages may assign different security attributes to it. This is a problem because it means that the security guarantees of a cookie depend on the page which set it, which might be exploited by a clever attacker to downgrade security. For example, a `Secure` cookie might be incorrectly set without the `Secure` attribute in some pages, which might downgrade confidentiality against network attackers.

We can capture these security problems by means of the following definition:

Definition 3 (Inconsistent Cookie Security)**.** The site s uses cookie security attributes inconsistently if and only if the following policy $(O, A, L, \{\rho_a\}_{a \in A})$ is inconsistent:

- O includes the set of cookies c_i collected at s. Each c_i is uniquely identified by a triple including its name, domain and path in accordance with RFC 6265 [2];
- A includes all the pages at s, since any page of the site can set a cookie for it, possibly by setting the `Domain` attribute to a parent domain;
- L includes all the sets which can be built from these elements: `HttpOnly`, `Secure`, `SameSite=lax`, `SameSite=strict`;
- For any $a \in A$, we let $dom(\rho_a)$ be the set of the cookies set by a and, for all cookies $c_i \in dom(\rho_a)$, we let $\rho_a(c_i)$ be the set of the security attributes of c_i given by a.

The definition captures that inconsistencies might arise when some security attribute is occasionally missed on a cookie or the `SameSite` attribute is configured with different options (`lax` vs `strict`) on the same cookie.

2) Content Security Policy: CSP can be configured to greatly mitigate the dangers of XSS by forbidding the injection of malicious JavaScript. For example, previous work proposed a definition of *safe* CSP as a minimal baseline to rule out trivial XSS attacks [5]. We update it here to account for the addition of the `'strict-dynamic'` keyword to CSP:

Definition 4 (Safe CSP)**.** A CSP is *safe* if and only if it contains a `script-src` directive (or a `default-src` directive in its absence) bound to a value v satisfying both the following conditions:

1) v does not contain the `'unsafe-inline'` keyword, unless nonces or hashes are also present in v;
2) v does not contain the wildcard $\star$ or any full scheme from the following: `http:`, `https:`, `data:`, unless `'strict-dynamic'` is also present in v.

Clause 1 rules out the injection of inline scripts and event handlers, while clause 2 ensures that script tags are subject to meaningful restrictions on what they can load. To properly protect an origin against XSS, all pages therein should be protected with safe CSPs. Otherwise, the protection offered by the SOP could be voided by attacking an unprotected page.

We thus capture the threats occurring from the use of safe CSPs on some pages, but not others, as follows:

Definition 5 (Inconsistent CSP)**.** The origin o uses CSP inconsistently if and only if the following policy $(O, A, L, \{\rho_a\}_{a \in A})$ is inconsistent:

- O includes a single object, i.e., the origin o itself. This uniformly models all the browser assets protected by the SOP, like the local storage and the DOM;
- A includes all the pages at o, since any such page can provide a different CSP;
- $L = \{\bot, \top\}$ discriminates unsafe and safe CSPs;
- For any $a \in A$, we let $dom(\rho_a) = \{o\}$. We then have $\rho_a(o) = \top$ when a enforces a safe CSP, while $\rho_a(o) = \bot$ otherwise. Note that the latter also covers the case where a does not enforce any CSP.

For example, let o be an origin with two pages $A = \{a, b\}$, where a deploys a safe CSP like `script-src 'self'`, while b does not enforce any CSP. This CSP adoption would be inconsistent because $\rho_a(o) = \top$, while $\rho_b(o) = \bot$. Indeed, an XSS at b would allow the attacker to access the local storage of o, no matter what a does.

3) HTTP Strict Transport Security: Hosts can activate HSTS protection on a per-domain basis, possibly extending it to subdomains through the `includeSubDomains` option. Defining the consistency of HSTS adoption is tricky since inconsistencies might arise within the same origin or across origins with the same parent domain. We thus split the analysis into two cases.

We start by discussing the consistency of HSTS for one origin. The key problem here is that, since HSTS is configured at the page level, two pages hosted on the same origin can specify conflicting policies, which can severely harm security,

e.g., by having `max-age` set to 0 on one page and `max-age` greater than 0 on the other one. Similarly, site operators might just forget HSTS headers on some pages, leaving such pages unprotected when they are accessed before HSTS is activated. We can capture these potential security issues through the following definition:[1]

Definition 6 (Origin-Inconsistent HSTS)**.** The origin o uses HSTS inconsistently if and only if the following policy $(O, A, L, \{\rho_a\}_{a \in A})$ is inconsistent:

- O includes a single object, i.e., the origin o itself. This formalizes that HSTS provides protection for the entire origin;
- A includes all the pages of o, since any such page can provide a different HSTS policy;
- $L = \{\bot, \top\}$ discriminates between disabled and enabled HSTS protection. By "enabled" we mean that HSTS is activated with a `max-age` greater than 0, i.e., we abstract from the duration of protection;
- For any $a \in A$, we let $dom(\rho_a) = \{o\}$. We then have $\rho_a(o) = \top$ when a provides an HSTS policy with the `max-age` attribute set to a positive value and $\rho_a(d_a) = \bot$ otherwise.

A subtler form of inconsistency which crosses the origin boundary occurs when the `includeSubDomains` option is not appropriately used: we recommend here the best practice of activating this option on the root domain to ensure the confidentiality and the integrity of domain cookies, as well as to protect new subdomains which might be created in the future. This suffices to enforce consistent protection across the whole site, unless a subdomain explicitly disables HSTS by setting it with `max-age` set to a non-positive value. This leads to the following more complicated definition:

Definition 7 (Site-Inconsistent HSTS)**.** The site s uses HSTS inconsistently if and only if the following policy $(O, A, L, \{\rho_a\}_{a \in A})$ is inconsistent:

- O is the set of all the subdomains of s plus its root domain, since HSTS protection is enforced on a per-domain basis;
- A includes all the pages of s, since any such page can provide a different HSTS policy;
- $L = \{\bot, \top\}$ discriminates between disabled and enabled HSTS protection. By "enabled" we mean that HSTS is activated with a `max-age` greater than 0, i.e., we abstract from the duration of protection;
- For any $a \in A$, let d_a stand for the domain where a is hosted: we let $dom(\rho_a) = O$ when d_a is the root domain of s, otherwise $dom(\rho_a) = \{d_a\}$. If d_a is the root domain of s, we let $\rho_a(d_i) = \top$ for all $d_i \in O$ when a provides an HSTS policy with the `max-age` attribute set to a positive value and the `includeSubDomains` option, while $\rho_a(d_i) = \bot$ otherwise. If d_a is not the root domain of s, we let $\rho_a(d_a) = \bot$ when a provides an HSTS policy with the `max-age` attribute set to non-positive value, while $\rho_a(d_a) = \top$ otherwise.

[1]For HSTS, the terms "origin" and "host" can be used interchangeably (assuming HTTPS is served on port 443), since HSTS is only valid on HTTPS.

Observe that the two definitions are independent. HSTS can be consistently deployed within a site which contains an origin where HSTS is inconsistently deployed, for instance when `includeSubDomains` is set on the root domain and `max-age` is never set to 0, yet some pages do not use HSTS. Conversely, HSTS can be consistently deployed within all origins of a site, yet be inconsistently deployed at the site, e.g., when `includeSubDomains` is not set on the root domain. However, both definitions are useful from a security perspective. Site-inconsistent HSTS leaves some subdomains unprotected, which might also harm the confidentiality and the integrity of domain cookies. However, site-consistent HSTS is only secure in practice under the assumption that the user actually visits the root domain to activate the `includeSubDomains` option. Site operators can enforce this by including explicit requests to the root domain in all pages, which is, however, error-prone. Using origin-consistent HSTS everywhere is thus recommended to mitigate the risks connected to failures in contacting the root domain during navigation.

IV. Real-World Inconsistencies

After establishing a formal framework to precisely define inconsistencies in Section III, we now want to understand how many inconsistencies occur in the wild. For this reason, we perform a large-scale Web measurement where we apply the inconsistency checks defined in our formal framework to policies collected from existing sites. We first discuss the data collection process and then present our findings, investigating their main security implications. We responsibly disclosed our findings to the operators of all sites mentioned in this section.

A. Dataset Construction

We performed our data collection with the *OpenWPM* [11] framework in Q1 2020 (03/24–04/15) from a single IP address belonging to CISPA in Germany. In particular, we crawled the top 15,000 sites that do not belong to the same entity, i.e., different second-level domain, based on the *Tranco* top 1M ranking [16] generated on 02/24/2020 (ID: PWYJ).[2] The number of analyzed sites is limited compared to other Web measurements, e.g., [11], since, due to the nature of our experiments, we have to scale our analysis vertically, i.e., crawl many pages of the same site. Naturally, our insights are limited by the same problems affecting the majority of large-scale measurements. The used browser instances are not logged into any site, nor do they interact with any site other than by following links. Therefore, we might miss pages of a site that are only accessible when logged in or after the user performs a specific state-sensitive task. Furthermore, our automated crawler might be detected by a page, which might then show different content than for real user visits.

We used *OpenWPM* to browse each site on our list, based on the following strategy. First, we access the homepage, collect all same-site links and group them by origin. Then we navigate all links and recursively repeat the process until we collect up to 300 first-party links to each identified origin or exhaust the links to visit. Finally, we access all yet unvisited links to collect their policies. We limit our analysis to 300 pages, as previous work showed that this amount of pages provides a

[2]Available at https://tranco-list.eu/list/PWYJ.

TABLE I: *Overview of cookie security results*

	overall usage		inconsistent usage	
	# cookies	# sites	# cookies	# sites
Host-only cookies	65,601	7,329	—	—
At least one attribute	27,698	5,726	880 (3%)	429 (7%)
- `HttpOnly`	20,915	5,116	502 (2%)	223 (4%)
- `Secure`	17,220	4,175	590 (3%)	225 (5%)
- `SameSite`	1,678	867	250 (15%)	160 (18%)
Domain cookies	27,991	5,851	—	—
At least one attribute	14,888	4,740	666 (4%)	258 (5%)
- `HttpOnly`	10,572	4,327	154 (1%)	103 (2%)
- `Secure`	9,397	3,140	298 (3%)	168 (5%)
- `SameSite`	2,651	2,209	409 (15%)	101 (5%)
All cookies	93,592	8,883	—	—
At least one attribute	42,586	7,541	1,546 (4%)	642 (9%)
- `HttpOnly`	31,487	7,014	656 (2%)	319 (5%)
- `Secure`	26,617	5,594	888 (3%)	369 (7%)
- `SameSite`	4,329	2,867	659 (15%)	248 (9%)

sufficient overview of a site's behavior [35]. We drop origins for which we identified less than five pages, as the high chance of a lack of inconsistencies would bias our analysis.

Our data collection relies on a *stateless* crawler, i.e., we load a fresh browser instance on each page visit. This allows us to force the server to set fresh cookies on each visit, which maximizes the ability to detect inconsistent uses of security attributes on the same cookie. To apply our formal framework to the collected dataset, we extract and parse the three client-side security policies of interest: cookie security attributes, CSP, and HSTS. We also normalize the collected policies to ignore the order of policy elements and capitalization, as well as to replace dynamically generated CSP nonces with fixed placeholders. Note that while cookies can also be set via JavaScript, we focus exclusively on mechanisms communicated via HTTP headers.

Overall, our crawler successfully loaded URLs from 12,352 (82%) sites. We attribute this difference to the fact that some of the sites on the top list are not meant to be visited in a browser (e.g., APIs endpoints), some service providers exclude users from the EU due to legal reasons (i.e., the GDPR), some sites provided empty pages (e.g., CDNs), or could not be reached within a timeout of 120 seconds. In total, our crawler loaded 14,094,544 URLs (both as the main page as well as frames) spread across 149,671 origins. Removing from our dataset all URLs that belong to an origin with less than 5 visited URLs, we are left with 13,369,750 URLs across 39,967 origins. These origins belong to a total of 10,878 different sites. We now apply the formal framework from Section III to the collected data and identify inconsistencies at scale.

B. Cookie Security Inconsistencies

Overall, we collected 93,592 first-party cookies, including 62,601 host-only cookies and 27,991 domain cookies, set from 8,883 sites. This means that 1,995 (18%) sites did not set any first-party cookie, which might sound surprising, but can be attributed to the presence of GDPR banners which require users to click through. Table I presents an overview of the results of our cookie analysis, which we discuss in the following.

We observe that 42,586 (46%) cookies set at least one security attribute, leading to 7,541 sites (85% of sites using cookies) which make use of security attributes. The most widely used security attribute is `HttpOnly` on 31,487 (34%) cookies, followed by `Secure` on 26,617 (28%) cookies, while the relatively recent `SameSite` is only deployed on 4,329 (5%) cookies. `SameSite` was used in 353 (8%) cases with the `strict` option, while the other 3,982 (92%) cases used the `lax` option. Notably, we observed a number of cases where cookie security attributes were used inconsistently (see Section III-B1). In particular, focusing on the set of cookies and sites where security attributes have sometimes been used, we identified inconsistencies in 1,546 (4%) cookies from 642 (9%) sites. Hence, although most cookies use security attributes consistently, the amount of sites where cookie security attributes are set inconsistently at least once is not negligible. Inconsistencies occurred in 880 (3%) host-only cookies from 429 (7%) sites and 666 (4%) domain cookies from 258 (5%) sites. All three security attributes seem to be inconsistently used in the wild: we identified 656 (2%) inconsistent uses of `HttpOnly`, 888 (3%) inconsistent uses of `Secure` and 659 (15%) inconsistent uses of `SameSite`. As to the latter attribute, 653 inconsistencies came from the omission of the attribute, while only 6 inconsistencies were due to conflicting options (`strict` vs `lax`). We believe that the higher number of inconsistencies in the use of the `SameSite` attribute comes from the relatively recent introduction of this defense mechanism, whose thorough deployment on existing sites has likely been somewhat complex. Moreover, we observe that domain cookies are inconsistently protected slightly more often than host-only cookies in terms of relative numbers. Notably, recall that domain cookies might be set on different subdomains, hence they present larger room for inconsistencies: in particular, we observe that 379 (49%) inconsistencies on domain cookies were introduced by different origins on 78 sites. In the following, we present case studies for each of the attributes.

Inconsistent HttpOnly: On `verizonwireless.com`, we identified an inconsistent protection of cookie `AMAuthCookie`, which is used for authentication purposes. The cookie was set with the `HttpOnly` attribute on 19 pages and without the attribute in 11 pages. An attacker who identifies an XSS vulnerability on the site might inject JavaScript code to steal the cookie unless CSP is configured to mitigate XSS. While the pages we crawled included a CSP, none of them mitigates XSS attacks. We also discovered inconsistent usage of `HttpOnly` for several `ASP.NET_SessionId` cookies, e.g., on `law.com` and `starbucks.com`. Notably, neither of them deploys a CSP, making cookie theft trivial to an attacker who finds an XSS anywhere within the respective origin.

Inconsistent Secure: On `asana.com`, we found an inconsistent protection of cookie `xsrf_token`, which is used to protect against CSRF attacks. In particular, the site used the `Secure` flag inconsistently, meaning that the cookie might be leaked over HTTP. This means that a network attacker can sniff the value of the anti-CSRF token and then perform CSRF attacks. Another interesting example is `imdb.com`. Their session cookie is set to non-`Secure` on the start page, but set to `Secure` on all pages under `/videoembed`. Manually investigating this, we found that visiting `/videoembed` with an existing session cookie leads to the server *not* setting any cookie. However, having a `Secure` session cookie and visiting the start page, we received a fresh copy of the cookie without the `Secure` attribute, which would expose the cookie in clear.

TABLE II: *Number of different CSPs per origin and site*

	1	2	3	4	> 4
CSPs per origin	4,424 (90%)	330 (7%)	70 (1%)	29 (1%)	43 (1%)
XSS mitigation	1,566 (85%)	174 (9%)	43 (2%)	28 (2%)	35 (2%)
CSPs per site	2,022 (73%)	476 (17%)	134 (5%)	49 (2%)	77 (3%)
XSS mitigation	818 (73%)	183 (16%)	46 (4%)	27 (2%)	55 (5%)

TABLE III: *Overview of CSP results*

	# origins	across sites
Deploy CSP at least once	4,896	2,758
May use CSP for XSS mitigation	1,846	1,129
Yet only have unsafe CSPs	1,413	895
Have at least one safe CSP	433	293
Have safe CSPs on all pages	233	171
Inconsistent CSP adoption	200	141
- missing CSP header at least once	162	119
- have both safe and unsafe CSPs	60	44
- usage of `'unsafe-inline'`	59	43
- allowing entire schemes	40	25

We believe this to be attributed to different applications running within the same origin.

Inconsistent SameSite: On `webteb.com`, we found an inconsistent protection of cookie `ASP.NET_SessionId`, which is used for session management. The cookie was sometimes set with the `SameSite=lax` attribute and sometimes without such attribute. This is concerning because the site might be vulnerable against CSRF on some browsers. In particular, while Google Chrome is experimenting with the automated activation of `SameSite=lax` on all cookies, other browsers like Mozilla Firefox are not doing that, meaning that the omission must be considered a security issue. Other instances of such inconsistencies include `kitapyurdu.com`, a Turkish online book store, as well as `eastdane.com`, a clothing online store, that provide inconsistent `SameSite` attributes. These are of particular interest, given that they both feature a login and features to buy items, making them prime targets for CSRF or clickjacking attacks. Both these attacks are in the specific threat model mitigated through `SameSite` cookies.

C. CSP Inconsistencies

Our analysis of CSP is exclusively based on HTML documents, as the use of CSP for script restriction is only meaningful for documents that can run scripts. Therefore, in the following, we report on those URLs that returned a content type of `text/html`, which amount to 13,058,387 pages on 39,255 origins from 10,852 sites. In our experiment, 4,896 (12%) origins across 2,758 (25%) sites made use of CSP on at least one page. Table II shows the number of different CSPs found on individual origins and sites, including those which attempt to mitigate XSS, i.e., those which use the `script-src` or the `default-src` directives. It turns out that 90% of all analyzed origins that deploy CSP just make use of a single policy, while the same holds true for 73% of sites (a site might use different CSPs on different origins). However, that also indicates that around 10% of origins use multiple policies, exacerbating room for inconsistencies.

Table III summarizes the key numbers computed for our CSP analysis. Overall, we observe that CSP could potentially be used for XSS mitigation on 1,846 (5%) origins from 1,129 (10%) sites. Remarkably, only 433 (1%) origins on 293 (3%) sites deploy at least one safe CSP, which is in line with previous work which observed that most CSPs in the wild do not properly mitigate XSS [5, 38, 39, 28]. Out of these security-savvy cases, only 233 (54%) origins across 171 sites make consistent use of safe CSPs across all of their pages. In contrast, 200 (46%) origins across 141 sites having at least one safe CSP deploy CSP inconsistently, i.e., deploy safe CSPs only on a subset of their pages (see Section III-B2). This is interesting, because configuring CSP correctly is far from trivial [28], hence site operators spend significant effort to mitigate the impact of XSS on some pages, yet introduce potential loopholes on other pages. Analyzing why inconsistencies arise, we observe that 162 (81%) origins deploy a safe CSP at least once, but do not ship any CSP header in some other page. As to the other cases, we note that 60 (30%) origins deploy safe CSPs on some pages, yet ship unsafe CSPs on others. In all but one case, this insecurity is caused by the `'unsafe-inline'` keyword, which was included in 59 origins. Additionally, in 40 cases operators allowed an entire scheme (possibly via the wildcard $*$). Note, given the overlapping conditions, the numbers do not sum up to the expected totals.

Google Docs: In analyzing the dataset for origins that have inconsistent protection, we discovered that *Google Docs* was among those. Further investigation highlighted an interesting insight: the inconsistency was caused by a missing header for HTML-rendered spreadsheets. However, all scripts in that spreadsheet had a randomized nonce attached. Hence, while the page itself apparently generated the necessary nonce, no policy was sent to the client. Overall, we found 315 pages under `https://docs.google.com/spreadsheet` to suffer from this problem. It is interesting to note, though, that our crawler loaded an additional 12 pages from `/spreadsheet` as well; these, however, were properly protected. Furthermore, 1,246 pages under the same origin, yet different paths (e.g., forms, viewer, presentations), also were properly protected by CSP. However, given that we could find pages without a CSP in that origin, any protection of CSP may be completely undermined in presence of XSS vulnerabilities.

HackerOne: A second example is *HackerOne*, a widely-used platform to disclose vulnerabilities. Most of the pages we crawled under `https://hackerone.com` delivered either a CSP that restricts scripts to their own origin as well as Google Analytics, or one that allowed certain scripts through hashes. While the latter contains `'unsafe-inline'`, the option is invalid for modern browsers in the presence of hashes. The only exception to these rules were pages under `/resources`, which carried no CSP at all. Judging from the type of content, namely searchable eBooks and articles, these pages seem to be running a different application underneath. Disclosing this, however, we got feedback that the lack of CSP was out of scope for them, and rather an issue of best practices. It must be noted, though, that this decision puts into question having any CSP, as the lack of a policy on *any* page in an origin may undermine the security guarantees provided through CSP.

TABLE IV: *HSTS inconsistencies per origin*

	# origins	across sites
Deploy HSTS on at least one page	14,112	5,352
Origin-inconsistent usage	2,323	1,731
non-positive `max-age` on at least one page	637	415
missing HSTS header on at least one page	1,810	1,465

D. HSTS Inconsistencies

Overall, we identified 14,112 (35%) origins on 5,352 (49%) sites which activated HSTS on at least one page. As discussed in Section III-B3, we consider two types of HSTS inconsistencies, in particular origin-inconsistent (Definition 6) and site-inconsistent (Definition 7) HSTS. Table IV shows the result for origin-inconsistent cases. We find that out of the 14,112 origins with HSTS on at least one page, 2,323 (16%) have inconsistencies. On 1,810 (78%) origins, this is caused by a missing header. While this is problematic when a URL with missing header is visited before any HSTS-protected URL in the origin, a more dangerous practice is exhibited by 637 (27%) origins where at least one page sets `max-age` to a non-positive value, thus deactivating HSTS for the entire origin. This means in particular that HSTS deactivation can be forced by a network attacker.

To assess cases of site-inconsistency, we need to conduct two additional steps. As per Definition 7, to be consistent, a site must set HSTS with `includeSubDomains` for its root, and no subdomain of the site must have a page that disables HSTS. Since most of the sites we visited during crawling redirected us to a subdomain (mostly to `www.site.com`), our crawl data does not necessarily contain HSTS headers for the root domains. To alleviate that, for all domains for which we had not visited the start page in the root domain, we instead visited `https://site.com` and collected the HSTS header for that request. If the request failed, we marked that site as not having HSTS on the root. Also, since HSTS is known to be a trust-on-first-use mechanism, browsers come equipped with the so-called HSTS preload list [13]. We downloaded the list from the Chromium Github repository [9], parsed it, and flagged those sites which were listed with `includeSubDomains`. By combining these two sources of information, we get reliable evidence about the adoption of HSTS on the root domains.

Table V shows an overview of our results. We find that of the 5,352 sites which deploy HSTS somewhere, 4,351 (81%) do so inconsistently. For all but 31 cases this is due to the absence of the `includeSubDomains` option on the root domain (as discussed above). In addition, 415 (10%) sites are susceptible to HSTS deactivation on at least one subdomain, due to the use of a non-positive value of `max-age` somewhere. We further wanted to understand if this could be an explicit choice, i.e., site operators only wanted to use HTTPS on certain subdomains, but not all. To that end, we filtered the results by those sites which have more than one subdomain, and where all subdomains set HSTS on at least one page, giving a set of sites run by operators committed to full HTTPS deployment. Overall, these amount to 1,028 sites. Unfortunately, even out of those more vigilant sites, 718 (70%) did not deploy HSTS consistently, again primarily caused by the lack of `includeSubDomains` in their root.

TABLE V: *HSTS inconsistencies per site*

	# sites	# sites with HSTS on all subdomains
Deploy HSTS on at least one page	5,352	1,028
Site-inconsistent usage	4,351	718
non-positive `max-age` on at least one page	415	84
no `includeSubDomains` on the root	4,320	706

Wired: One of the vulnerable cases we discovered was `wired.com`, which had a `max-age` set to 0 on certain pages. We disclosed this issue and the feedback from the developers was initially of surprise. Working with them, we discovered that only URLs under `/coupons` had set this `max-age` value. Based on both the layout and content of the page, we again believe this to be a separate application under the same origin as the rest of `wired.com`. As we show in the following, this not only had implications on the connection security, but could also lead to cookie leakage through lacking `Secure` attributes.

Cookie Leakage Through HSTS Inconsistencies: To further understand the security implications of inconsistent HSTS adoption, we performed an analysis to estimate the number of cookies which can be exposed in the clear against network attackers. We thus focus on non-`Secure` cookies, which however are set on pages that have deployed HSTS, i.e., that rely on HSTS to protect their cookies rather than the `Secure` attribute. Specifically, we identify three categories of cookies at danger on HSTS-enabled sites:

1) Host-only cookies set by a page that correctly deploys HSTS, yet another page on the same origin has configured `max-age` to a non-positive value. The attacker can exfiltrate these cookies after forcing HSTS deactivation.
2) Domain cookies bound to domains such that HSTS is not activated with `includeSubDomains` for the domain itself, nor at least one of its parents. The attacker can steal these cookies by forcing HTTP requests to a non-existing subdomain (of the set `Domain` value). Since during our analysis we might not have visited all existing parents, we resort to live checking of the unseen parents.
3) Domain cookies bound to domains such that any of their subdomains deactivates HSTS through a non-positive value of `max-age`. After HSTS deactivation, these cookies can be leaked by forcing the victim to visit the now downgraded HTTP subdomain.

For the first case, we find 17 affected origins on 17 sites. Among these cases, two prime examples are the cookies `_-session_id` on `wired.com` and `cookie_session1` on `bankmellat.ir`. Both cookies seem to be related to session management and are thus security-sensitive.

As to the second category, we find that a number of pages across 1,712 sites set domain cookies without `Secure`, while deploying HSTS with a positive `max-age`. There, we identify 1,254 sites affected by lacking `includeSubDomains` on the cookie's domain or any of its parents. Examples include `audible.com` (for which the `session-token` cookie can be leaked), `rentalcars.com` (`JSESSIONID`), and `alipay.com` (`ALIPAYJSESSIONID`). In addition, we found 19 sites which leak a CSRF token in this fashion.

Finally, for the third category, we discover that 54 sites are affected. One example is `taobao.com`, which sets the cookie `_tb_token_` as a domain cookie for `.taobao.com`. However, several pages under `world.taobao.com` set `max-age=0`, thereby enabling leakage of the cookie.

Overall, we conclude that 1,783 sites risk exposing cookies in the clear due to inconsistencies in their HSTS configuration. This is 33% of all the sites that activate HSTS on at least one page, highlighting the significance of the presented threats in the wild.

E. Summary

Our Web measurement showed that inconsistencies in the configuration of client-side security mechanisms frequently occur in the wild. This is caused by the fact that client-side security mechanisms are normally set by individual pages, but affect entire origins or even sites. Inconsistencies in the use of cookie security attributes are limited in terms of absolute numbers (4% of all cookies), yet the amount of sites where we can find at least one such inconsistency is not negligible (9%). In particular, the amount of inconsistent uses of the relatively recent `SameSite` attribute is quite frequent (15%), which suggests that keeping cookie security attributes consistent across the whole site is not straightforward in general. The picture is even worse when looking at more complex security mechanisms like CSP and HSTS. We observed that 46% of the origins which use CSP for XSS mitigation do it inconsistently, thus voiding or severely harming the benefits of protection. HSTS is inconsistently deployed on 16% of origins (see Definition 6) and on 81% of sites (see Definition 7). This might lead to security issues like lack of HSTS activation, forceful HSTS deactivation by network attackers, and cookie leakage, as we could highlight through our empirical analysis. In particular, we showed that 33% of all the sites that activate HSTS might leak cookies in clear due to its inconsistent use.

V. Origin Policy to the Rescue?

As our measurement has shown, many sites on the Web deploy security mechanisms inconsistently. Considering cookies, we find that 642 sites have conflicting security attributes for the same cookie. Moreover, while CSP is known to be hard to securely deploy in practice, 200 out of 433 origins deploying at least one safe CSP have inconsistencies, with the majority (162) being caused by omitted headers. We find omissions to be the major contributor to HSTS inconsistencies as well (on 1,810 out of 2,323 inconsistent origins). All of these issues are caused by the fact that security mechanisms are deployed on a per-page basis. One approach to enforce security policies on a per-origin basis rather than on a per-page basis is the W3C proposal of an *Origin Policy* (OP) [10], formerly known as Origin Manifest. This proposal attempts to address two main issues caused by the need to repeat the same policy on all pages within an origin: waste of bandwidth and potential security flaws introduced by missing headers, typos, and other issues coming from the need to explicitly enforce security on every page. While we do not discuss the impact of conserving bandwidth here, we nevertheless aim to determine if OP can address the inconsistency issues we discovered in the wild.

```
{
 "ids": ["policy-1"],
 "content_security": {
   "policies": ["script-src 'self'
   ↪  https://cdn.example.com"]
  }
}
```

Fig. 1: *Example of an* Origin Policy *defining an origin-wide CSP.*

A. Specification of an Origin Policy

OP enables the definition of an origin-wide security policy stored at a well-known location.[3] This policy is roughly a dictionary binding a set of client-side security policies to an identifier. Clients download the origin policy manifest, cache it, and enforce the (cached) security policies bound to the identifier set in the `Origin-Policy` header of individual pages. Figure 1 shows an example of such a manifest. It defines the policy called `policy-1`, which includes an origin-wide CSP; such policy is uniformly applied to all pages which set the `Origin-Policy` header to the value `allowed=("policy-1")`. Since real-world CSPs can be much longer and complex than the one in our example, this is useful to save bandwidth and reduce room for errors. If the policy ever needs to be updated, the site operator can change its content, update its identifier (e.g., to `policy-2`) and set the header to the new identifier. It is important to note that while the identifier seemingly indicates it might be used as a *selector* for some specific policy, it is just a *serial number*. That is, the browser can check whether, for a given header, the policy with that key is cached. If some policy with a different identifier is cached, this is dropped and replaced with the new origin policy. In the current specification, Origin Policy supports CSP and Feature Policy, although support for HSTS is discussed in its Github issue tracker [25].

B. Applicability of Origin Policy in the Wild

OP supports the definition of a single origin-wide security policy. Exceptions to such policy can be implemented in two ways: (i) omitting the `Origin-Policy` header in specific pages, so as to (possibly) replace the origin policy with new custom policies, or (ii) keeping the `Origin-Policy` header and combining it with additional security headers, so as to refine its baseline guarantees. For example, when a page uses multiple CSPs, all of them are simultaneously enforced according to the OP specification [10].

Next, we evaluate the applicability and potential security benefits of OP in the wild based on real-world data collected in Section IV. Since OP does not support HSTS and cookie security attributes at the time being, we generalize its design to these mechanisms in the expected way. In particular, we assume that OP is extended to enforce a single origin-wide HSTS policy and a single origin-wide set of security attributes for individual cookies.

1) Cookie Security Attributes: Our measurement identified 1,546 cookies on 642 sites with inconsistent security attributes. We cannot ascertain for sure whether these inconsistencies are intended or not, but we expect the large majority of

[3] `/.well-known/origin-policy`

them to be oversights since site operators plausibly put a given attribute on a cookie for good reasons. These security issues would be fixed if site operators used OP to enforce the union of their attributes on these cookies, privileging `SameSite=strict` over `SameSite=lax` in the presence of conflicting configurations. While this approach would likely work for host-only cookies, it might not generalize to domain cookies, since the same domain cookie can be set by different subdomains, hence crossing the origin boundary advocated by OP. In particular, we identified 379 domain cookies on 78 sites where two distinct origins introduced inconsistencies. These inconsistencies would not be fixed by OP unless all origins that set the same cookie agreed on its security attributes within the respective origin policies, which is error-prone and requires careful vetting of the Web application.

2) CSP: Our measurement identified 472 origins that deploy more than one single CSP and, hence, cannot take immediate advantage of OP. In particular, site operators might deploy the most frequently used policy as a single origin-wide CSP, but would be forced to explicitly code exceptions on several pages, e.g., by dropping the `Origin-Policy` header and by supplying appropriate CSPs where needed. This would force site operators to repeat the same CSPs unerringly on a subset of pages, which does not solve the key security issue which OP tries to address. On a related note, we also identified 162 origins which deploy safe CSPs on some pages, but do not use CSP on other pages. These constitute a significant fraction of the 433 origins that try to deploy a safe CSP at least once (37%). While OP supports this practice by omitting the `Origin-Policy` header, this is dangerous and it is worth noting that header omissions might be intended or unintended. We cannot ascertain whether missing CSP headers in our dataset are intended or not. Nevertheless, it is fair to assume that site operators might accidentally forget security headers somewhere, including the `Origin-Policy` header. Given how widespread this practice is, we argue that OP does not provide sufficient countermeasures against this threat.

3) HSTS: Recall our analysis builds on two notions of inconsistency for HSTS, i.e., origin-inconsistency and site-inconsistency. Our measurement identified inconsistent HSTS adoption on 2,323 origins. Inconsistencies have been introduced by missing headers in 1,810 cases, which are very likely oversights, considered that even a single page with HSTS could already activate the security mechanism on the entire origin. These security issues would be fixed if site operators used OP to enforce a single HSTS with the maximum value of `max-age` collected across all pages on the origin, possibly coding explicit exceptions whenever needed. However, we also observed that 4,351 sites deploy HSTS inconsistently, which amounts to 81% of the sites which use HSTS. Most notably, these issues are due to the lack of adoption of the `includeSubDomains` option on the root domain (4,320 cases). These inconsistencies would not be fixed by OP, since they cross the origin boundary: in particular, we have already discussed that using HSTS consistently within all origins does not suffice to deploy HSTS consistently within a site.

C. Limitations of Origin Policy

Based on the results of our analysis, we argue that the current OP design has three main limitations:

1) *Single Policy Support*: OP only supports a single policy per origin, excluding outdated cached policies. Though exceptions can be implemented as explained, extensive adoption of exceptions essentially voids the benefits of OP. Specifically, our data show that a non-negligible amount of origins do not enforce a single policy with few well-defined exceptions, but rather deploy a small set of policies reused on different sets of pages. OP does not support this design.
2) *Cross-Origin Inconsistencies*: OP does not include tools to fix cross-origin inconsistencies. Though this is not yet a problem for the current OP proposal, this issue would become important if OP evolved to be a collector of client-side policies beyond CSP and Feature Policy, as it seems to be [25, 36]. In particular, our data indicate that many inconsistencies in the use of HSTS and cookie security attributes are indeed cross-origin, which suggests that the origin boundary is not appropriate to fix them.
3) *Implicit Security Exceptions*: simply omitting the `Origin-Policy` header in specific pages is useful to implement exceptions to the origin policy, yet this practice is a dangerous way to relax security since it comes with the implicit assumption that the header omission is deliberate. Clearly, we have no rigorous way to assess whether the many header omissions we found in the wild were intended or not. However, mechanisms such as HSTS, which are meant to be enforced origin/site-wide, require explicit opt-out; hence not having the header has differing effects, as it depends on what URLs on the same origin/site the browser visited before. Hence, we believe it to be likely that the lack of the header is an unintended omissio and we think it is fair to assume that unintended header omissions would also occur for the `Origin-Policy` header. On a related note, new subdomains created on a site are insecure by default until their OP is specified and enforced.

VI. Proposal: Site Policy

As our measurement has shown, Web sites often suffer from inconsistencies in their security mechanisms. Based on our observations, we believe that in several cases, different applications operate under the same origin or site. Hence, the Web server merely acts as a reverse proxy, and each application sends its own (in)secure headers. As we have shown before, Origin Policy by itself cannot address this, especially the omission of headers. To solve the limitations of OP described in Section V, we propose a novel solution to address OP's shortcomings. We call this proposal *Site Policy* (SP), and we detail it in the following. We also discuss the security benefits of SP and report on a prototype implementation, which we make publicly available to experiment with [8].

A. Overview

Similarly to OP, SP relies on a centralized manifest for policy specification (stored at `https://site.com/.well-known/site-policy`). SP is designed to help site operators ensure that security policies are consistently used site-wide and enjoy appropriate security guarantees. In particular, SP has three key differences compared to OP:

1) *Multiple Policy Support*: SP acts as a centralized collector of multiple policies. Each page can individually

pick one of these policies by naming its corresponding identifier in the `Site-Policy` header, similar to what a page would do to pick a specific OP version using the `Origin-Policy` header. Note the important difference here: while OP includes multiple versions of the same policy, applied origin-wide, SP offers a choice between several available policies for different pages. This way, one can enforce multiple policies from a fixed set on the same origin without falling back to the traditional approach of configuring security headers on individual pages, which is too fine-grained and error-prone.

2) *Site-Wide Policy Support*: the SP manifest is located on the root domain and can contain site-wide security policies, which also cover subdomains. This means that SP goes beyond the origin boundary: by defining site-wide security policies, one can easily fix cross-origin inconsistencies like those occurring for HSTS and cookie security attributes.
3) *Explicit Security Exceptions*: requests lacking a valid `Site-Policy` header fall back to a default policy applied on a per-domain basis, defining the intended security guarantees. Hence, security exceptions must be explicitly coded by choosing a policy that sacrifices security. The key benefit is that site operators have tools to opt-out from security where necessary, rather than having to opt-in for security everywhere. Since the SP manifest operates site-wide and mandates secure defaults, new subdomains are automatically protected.

The combination of these features makes SP a framework where all security inconsistencies on a site can be fixed by design, while making insecurity explicit and easy to detect. We now present SP in more detail and further elaborate on this.

B. Specification Details

Figure 2 shows the structure of an SP manifest. It contains top-level entries for policy specification, which allow site operators to bind policies to identifiers. In particular, the `policies` section (lines 53–69) defines three policies available on the site, all of which include entries for CSP, HSTS and host-only cookies. These entries can be filled with identifiers bound to policies in sections `csp-policies` (lines 3–6), `hsts-policies` (lines 7–17) and `hostcookie-policies` (lines 18–38), respectively. Policies for domain cookies are defined in the `domaincookie-policies` section (lines 39–52), while default policies for different subdomains are given in the `default-policies` section (lines 70–75).

The enforcement model of SP works as follows. Before making any request to a URL on a given site, the browser determines if an SP manifest is available and, if so, downloads and caches it. If a page sets the `Site-Policy` header to a valid policy identifier from the `policies` section, say `policy_default`, that policy is applied on the page. If instead the header is omitted or contains an invalid policy identifier, the matching algorithm iterates over all the elements of `default_policies`, finds the policy bound to the longest common suffix of the domain of the page, and finally applies it. For example, if a page on `https://bar.foo.domain.com` does not specify the `Site-Policy` header, `policy_default` is applied to it, since the longest common domain suffix is `domain.com`. Each manifest must have a default policy for the root domain

```
1  {
2    "max-age": 3600,
3    "csp-policies": {
4      "empty": "",
5      "secure_csp": "script-src 'self'"
6    },
7    "hsts-policies": {
8      "empty": "",
9      "hsts1": {
10       "max-age": 63072000,
11       "includeSubDomains": false
12     },
13     "secure_hsts": {
14       "max-age": 31536000,
15       "includeSubDomains": true
16     }
17   },
18   "hostcookie-policies": {
19     "secure_hostcookies": {
20       "<default>": {
21         "secure": true,
22         "httponly": true,
23         "samesite": "lax"
24       }
25     },
26     "optout_session_cookie": {
27       "<default>": {
28         "secure": true,
29         "httponly": true,
30         "samesite": "lax"
31       },
32       "session": {
33         "secure": true,
34         "httponly": true,
35         "samesite": "None"
36       }
37     }
38   },
39   "domaincookie-policies": {
40     "domain.com": {
41       "<default>": {
42         "secure": true,
43         "httponly": true,
44         "samesite": "lax"
45       },
46       "CID": {
47         "secure": true,
48         "httponly": false,
49         "samesite": "lax"
50       }
51     }
52   },
53   "policies": {
54     "policy_default": {
55       "csp": "secure_csp",
56       "hsts": "secure_hsts",
57       "hostcookie": "secure_hostcookies"
58     },
59     "policy_optout": {
60       "csp": "empty",
61       "hsts": "empty",
62       "hostcookie": "secure_hostcookies"
63     },
64     "policy1": {
65       "csp": "secure_csp",
66       "hsts": "hsts1",
67       "hostcookie": "optout_session_cookie"
68     }
69   },
70   "default_policies": {
71     "domain.com": "policy_default",
72     "www.domain.com": "policy1",
73     "optout.domain.com": "policy_optout"
74   }
75 }
```

Fig. 2: *Example of a Site Policy manifest*

where it is deployed, e.g., domain.com in the example, to ensure a baseline security policy everywhere.

We now discuss additional details for the specific security mechanisms which we consider in the paper. If SP was extended to additional mechanisms, one would need to perform similar considerations, yet the nature of SP would stay the same.

1) CSP: For CSP, the enforcement algorithm is aligned with the Origin Policy proposal. In particular, this means that any CSP chosen via the Site-Policy header is always enforced, yet individual pages might also supply their own CSPs using CSP headers or meta tags. When this happens, all CSPs are simultaneously enforced as required by the CSP specification [40]. CSP enforcement comes with a caveat with respect to the usage of nonces, also shared with OP. As the name suggests, a nonce should be used only once. Therefore, we cannot specify a nonce in the SP manifest, as this would be static. To alleviate this, we pick up on a proposal from Google [19] for modified handling of nonces. In particular, instead of having the server generate the nonce, which is delivered to the client and enforced there, we propose that the client generates a random nonce for each request and sends it to the server via a request header (e.g., CSP-Nonce). The server then adds this nonce to all inline scripts, and in its CSP sets the 'nonce' option, without mentioning the value of the nonce. The CSP enforcement on the client then works as usual but, instead of relying on a server-generated nonce, it uses the one associated with the outgoing request. In this way, both SP and OP can support nonces.

2) HSTS: When the HSTS header set on a page conflicts with the HSTS settings chosen via the Site-Policy header, SP enforces the HSTS policy with the longest max-age among them, activating the includeSubDomains option when at least one policy includes it. It is also worth noting that SP may actually subsume the includeSubDomains option, since its manifest file naturally supports the specification of policies which apply to a set of subdomains. In our example, all the subdomains of domain.com without an explicit entry inherit the HSTS policy secure_hsts. This is useful to simplify HSTS deployment, since activating the includeSubDomains option correctly is far from straightforward: developers have to explicitly include a request to the root domain in each page to appropriately ensure that the option always covers all the subdomains of the site. Moreover, embedding HSTS within SP also supports the ability of a site to activate HSTS on all but one tree of subdomains, which could be dangerous from a security perspective, yet still desired by site operators to prevent breakage. For regular HSTS, once includeSubDomains is set, all subdomains will be accessible only via HTTPS. If the option is omitted, instead, all subdomains without the header can be loaded over HTTP. In our design, assuming optout.foo.domain.com (and all of its children) are supposed to not have HSTS, we simply add a default policy entry in the manifest for that domain prefix and set its HSTS entry to the empty string.

3) Cookies: Regarding cookies, we generally distinguish between host-only cookies and domain cookies. The former are governed by entries from hostcookie-policies, as selected by the policy specified in the Site-Policy header or via a fallback to default policies in its absence, and the latter are governed according to the longest suffix match of the Domain attribute within entries of domaincookie-policies. In the respective selected entry, we choose the attributes that match the name of the cookies that should be set and resort to the attributes of <default> if the name is not explicitly mentioned. Note that when setting cookies, the application can set other attributes in the Set-Cookie header as well. In that case, SP enforces the union of the security attributes in the manifest and the header. If the SameSite attribute is set with conflicting options, the most restrictive one is taken. Note that we exclude path matching from the policy selection for simplicity and based on the observation that in the wild, differing security policies for cookies on the same origin with the same name, yet different paths, are rare. In particular, we only found 36 such cases on 93,592 collected cookies.

To exemplify, assume https://www.domain.com/ sets a cookie called session with the Domain attribute set to domain.com. Following our algorithm, we first pick the entry for domain.com in the domaincookie-policies entry. Since session is not explicitly mentioned, we resort to the default specified in line 41 and apply the maximum protection to this cookie. If, instead, this would be a host-only cookie, we would first choose the default policy policy1 according to our policy matching algorithm and use the host cookie policy defined in line 32. This would enforce that the cookie is set with HttpOnly and Secure enabled, while opting-out from the use of SameSite.

In line with our goal of making insecure deployments explicit, we mandate that domain-cookie policies always have an entry for the root domain as a fallback. Additionally, all cookie policies must specify a <default> entry. By construction, domain-cookie policies are not bound to single policies, as their security considerations should be uniform across the site.

4) Caching and Policy Updates: Similar to OP, the SP manifest is not fetched on each HTTP request, but rather cached in the browser. To support policy updates, we use a mandatory max-age attribute in the SP manifest (see line 2 of Figure 2) to express cache duration, rather than using a versioning system like OP. We opt for this since SP enforces default policies on pages lacking the Site-Policy header, meaning we cannot rely on communicating versions as OP does.

We suggest site operators keep the value of max-age relatively short, e.g., one hour. This ensures that policy updates are propagated to all pages within a short time frame while having only a very limited impact on performance, since the SP manifest only needs to be fetched once per hour. If the cache ever needs to be explicitly flushed before its expiration for generic reasons, we propose introducing an option in the Clear-Site-Data header [22] to clear the cached manifest.

C. Security Analysis

We designed SP to overcome the key limitations of OP, based on our analysis of real-world sites. In particular, recall that we identified: (i) 379 cookies on 78 sites which suffer from cross-origin inconsistencies in the use of cookie attributes, (ii) 472 origins which deploy different CSPs in different pages; (iii) 4,351 sites which present cross-origin inconsistencies in their HSTS deployment. All these cases are out of scope for OP, yet the SP design supports them. We also identified thousands

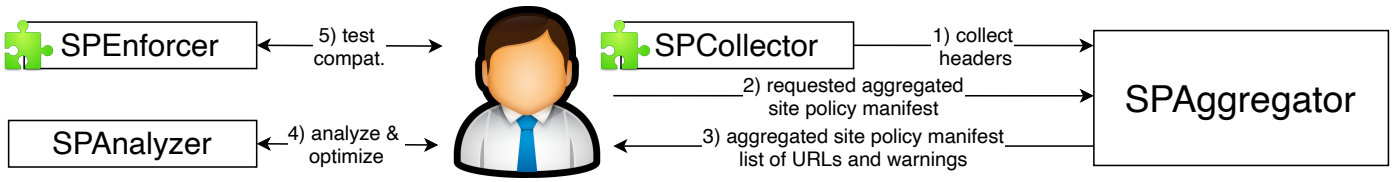

Fig. 3: *Overview of our prototype toolchain*

of inconsistencies due to missing security headers: in general, we cannot tell whether these omissions are intended or not, yet we can state with certainty that they are dangerous. SP forces site operators to make deliberate omissions explicit by choosing an insecure policy in the `Site-Policy` header. Unintended omissions, instead, would fall back to explicit defaults, which can provide useful security guarantees.

There is an important point to note. SP supports more use cases than OP and can be leveraged to fix all the inconsistencies within the same site, thus providing holistic protection over it. However, fixing inconsistencies might lead to breaking functionality, e.g., when a safe CSP is deployed on a page which requires inline scripts. To support functionality, SP still supports dangerous practices like using CSP inconsistently within the same origin, which might be required when site operators need to support unsafe inline scripts on a subset of the pages. Nevertheless, SP has the distinctive advantage of centralizing all security policies within the same manifest, which means that one can draw meaningful security conclusions by statically analyzing the manifest. We discuss examples below.

1) CSP: For CSP, one can conclude that XSS mitigation is correctly put in place simply by checking that all CSPs within the SP manifest are safe (Definition 4), which prevents inconsistencies by definition. If some unsafe CSPs are found, it is still possible to instruct site operators about the dangers of using them on origins that enforce a safe CSP by default. Checking this information in the manifest is straightforward.

2) HSTS: The SP manifest immediately clarifies the protection offered by HSTS across the site. In particular, it is possible to detect whether HSTS is activated for any subdomain of the site just by inspecting the SP manifest. If any policy does not activate HSTS or sets `max-age` to a non-positive value, it is possible to warn operators about the security risks of using it on an origin which otherwise activates HSTS. SP makes it straightforward to activate origin-consistent and site-consistent HSTS. For the former, it suffices to activate HSTS in the default policy of the origin to be protected, while for the latter, it is enough to set HSTS in the default policy of the root domain.

3) Cookies: The SP manifest prevents inconsistencies in the security attributes of domain cookies by design, since each domain cookie matches exactly one entry of the SP manifest (possibly a default one). SP is more liberal for host-only cookies since a single entity owns those cookies, yet detecting an insecure use of security attributes is straightforward. In particular, it is possible to find the baseline security guarantees of cookies by finding the intersection of their attributes for all matches found in the available policies (picking `lax` over `strict` when the `SameSite` attribute is set with conflicting options). Note that, when the intersection is different from the individual entries, an inconsistency might occur.

Furthermore, since the SP manifest includes information about HSTS, we can also use it to reason about the necessity of using the `Secure` attribute on cookies. In particular, the `Secure` attribute is redundant in two cases: (i) for host-only cookies, when HSTS is activated on the host; (ii) for domain cookies, when the domain set in their `Domain` attribute and all its children are protected with HSTS. This information is readily available from the SP manifest, as explained above.

D. Prototype Implementation

We implemented a prototype toolchain to support a future adoption of SP. Figure 3 shows an overview of the main components. A site operator willing to adopt SP installs a Chrome extension (*SPCollector*) and a Flask service (*SPAggregator*) on their machine. When browsing the site, the extension monitors responses for all relevant headers (currently: cookie, CSP, and HSTS headers) and sends the collected information to the Flask backend (1). Once the site has been navigated to the extent the operator wants, they query SPAggregator to request the site policy manifest (2). This manifest serves as the (potentially insecure) starting point for their site policy, containing entries for all observed security decisions, a list of URLs for which the generated policies have to be deployed, and optionally warnings about policy insecurity (3). Subsequently, the developer adapts the policy, verifying it for security with the *SPAnalyzer* (4). In parallel, the improved site policy is tested using the *SPEnforcer* extension, allowing the developer to detect breakage (5). We make our entire toolchain available for download [8].

1) SPCollector & SPAggregator: SPCollector is a simple Chrome extension that records all the observed security headers in HTTP responses and sends them to the SPAggregator. In this component, we store all URLs and the observed headers in a database. Once the user is done browsing the site, they invoke the policy aggregation step, which combines all observed policies into a single manifest. The aggregation acts as follows:

- For CSP, we first normalize all policies, i.e., rewrite nonces to a fixed value (as our proposal operates on a modified CSP implementation). Also, we parse and normalize all HSTS policies, ignoring non-existent options.
- For cookies, we check which cookies violate our default secure policy (`HttpOnly; Secure; SameSite=lax`) and generate an entry in the policy for them. Recall that we identify cookies disregarding the path for simplicity. For conflicting sets of attributes for a cookie, we choose the least restrictive one to avoid breakage and to expose the possible insecurity directly in the manifest.
- Considering the observed headers, we generate combined policies (CSP, HSTS, and cookie) as needed by the page. So, assuming that we only ever observed the combination

of CSP1/HSTS1 and CSP2/HSTS2, we would not generate a policy for CSP1/HSTS2 (and vice-versa).
- If there are just one policy for CSP and one policy for HSTS, we add an entry to the `default-policies`, setting the default policy for the origin to the single combination of the CSP, HSTS, and cookie policies.
- Besides, we generate a default policy for CSP, HSTS, and cookies. Specifically, we set CSP to `script-src 'none'`, we activate HSTS with `max-age=31536000` and `includeSubDomains`, and set all cookies to `HttpOnly; Secure; SameSite=lax`. We set the default policy for the entire domain to this strict combination. Assuming the developer deploys the correct selection of `Site-Policy` headers where needed and navigated enough pages to generate their default policies, this policy will never be selected. Hence, it just serves as a secure default to fall back to in case of forgotten headers.
- Apart from the JSON manifest, the aggregator also reports a mapping from policies to URLs. This allows the site operator to deploy appropriate `Site-Policy` headers in their Web server without any breakage.

2) SPAnalyzer & SPEnforcer: SPAnalyzer is a Python script implementing the security analyses described in Section VI-C. Given an input SP manifest, SPAnalyzer provides a security report about the state of XSS mitigation granted by CSP, the extent of the HSTS deployment, and the security guarantees of individual cookies. SPEnforcer is a Chrome extension using the *webRequest* API to implement the SP specification in Section VI-B. This is straightforward, since *webRequest* allows one to intercept network traffic and modify security headers.

VII. Related Work

All security mechanisms considered in the present paper received attention from the security community. However, as far as we know, no prior work studied the problem of inconsistencies that we investigate.

Inconsistencies in Web Security: The study of inconsistencies in Web security has primarily focused on analyzing bugs leading to different levels of protection across browsers. Notably, previous work focused on incoherent implementations of the SOP [30, 29], broken support for security mechanisms in mobile browsers [17], and differing guarantees in clickjacking protection [7]. Other papers also studied the inconsistent deployment of HTTP headers between the desktop and the mobile version of the same site [18, 37]. These works differ from the present paper since we are not concerned about inconsistent security across different browsers or across different variants of the same site. Instead, we study the inconsistent adoption of the same security mechanism across different pages of the same site. This aspect is an orthogonal issue with major security implications, yet so far overlooked by prior work.

CSP: The insecurity of the current CSP ecosystem has been studied in several research papers, though mostly focusing on CSPs collected from landing pages [5, 38, 6, 28]. A paper by Somé et al. [32], which has interesting connections with our work, discusses CSP violations due to the SOP. The authors discuss the dangers of using iframes on CSP-enabled sites, particularly when the framing document and the framed content share the same origin, but one of them lacks CSP. This is a specific kind of inconsistency in the use of CSP. They proposed to fix this issue by deploying an origin-wide CSP. In comparison, our analysis is more general since it detects inconsistencies when some pages use a safe CSP, but some others in the same origin do not. Under this condition, the presented attack can still be performed when the attacker finds an injection on a page lacking a safe CSP.

In addition to measuring CSP adoption and bypasses, Pan et al. [26] proposed to automatically curate CSPs from observed scripts. Similarly, Roth et al. [27] relied on automated CSP generation through observed scripts to assess the dangers of gadget-enabling libraries that are co-hosted with benign, required JavaScript. Eriksson and Sabelfeld [12] proposed AutoNav, a tool capable of automatically curating `navigate-to` directives for CSP. We rely on similar ideas to collect policies observed in the wild to curate the initial version of a site policy manifest for review by site operators.

HSTS: The first paper dealing with the deployment of HSTS is from 2015 [15]. The authors empirically investigated how widespread the HSTS adoption was back then and analyzed its security implications, but did not discuss inconsistencies. A more recent measurement study from 2017 looked at HTTPS security enhancements, also discussing HSTS [1]. The authors found a significant increase in HSTS adoption and also analyzed inconsistent deployment practices. However, their definition of inconsistency is different from ours: they do not consider different configurations of HSTS within the same site, but rather whether HSTS is uniformly used on all associated IP addresses. They also discuss inconsistencies in the headers identified across accesses from different geographic positions, which again is unrelated to our study.

Cookie Security Attributes: Previous research studied the adoption of the `HttpOnly` attribute [41] and of the `Secure` attribute [31], discussing the dangers from misuse in the wild. Other work also proposed client-side defenses designed to selectively activate security attributes on session cookies [23, 4]. None of these papers, however, studied the inconsistent adoption of security attributes on the same cookie.

Origin Policy: Since OP is still a working draft lacking support in browsers, it did not attract major attention from the security community so far. Van Acker et al. [36] were the only ones who studied OP. In their work, the authors performed an analysis of OP from different angles: (1) a formal investigation of open problems in the OP draft, (2) a prototype implementation of OP, (3) a prototype manifest generator, and (4) a Web measurement. The main goal of their measurement was checking the stability of HTTP headers, i.e., for how long they are left unchanged, and the bandwidth saving enabled by OP. This information is valuable and suggested that OP is promising, yet it is insufficient to provide definite answers on the practicality of OP. Our work highlights that OP is ill-equipped to rectify inconsistencies discovered in the wild. We also use our data to discuss major limitations in the OP design, which we propose to fix through a new solution (SP). Finally, it is also worth mentioning the Site-Wide HTTP Headers proposal [24]. Despite its name, this proposal is very similar to OP, since security headers are still applied origin-wide, and, to the best of our knowledge, it is not currently discussed for implementation in real browsers.

VIII. Conclusion

The overly fine-grained enforcement model of existing client-side security mechanisms makes them prone to the risk of inconsistencies. In this paper, we formalized inconsistencies for three prominent security mechanisms and showed their security dangers on 15,000 popular sites. We also showed that Origin Policy, the only available tool to fix inconsistencies by design, cannot be successfully applied to solve these issues in the wild. We thus proposed Site Policy (SP), a security mechanism explicitly designed to overcome OP's drawbacks in light of our real-world findings. SP provides the tools to fix all the inconsistencies that we defined and encountered in the wild. SP naturally supports secure Web application development, centralizes security, and supports safe defaults, while being backward compatible with existing sites which (temporarily) need to support specific inconsistencies. A key point to note is that any potential room for inconsistency must be made explicit in the SP manifest, which simplifies future researchers' measurements interested in the effectiveness of SP and, more importantly, allows operators to understand their worst-case security guarantees. We hope that our work will be useful to browser vendors, especially at this stage, where support for OP in commercial browsers is still in the making.

Future work should carry out a user study with a pool of site operators to collect their feedback about the prototype toolchain put forward in the present work and further refine the Site Policy proposal through interaction with major browser vendors and standards bodies like the W3C.

Acknowledgments

We would like to thank the reviewers for their advice on how to improve the presentation of our paper. In particular, we thank Adam Doupé for shepherding our paper. This work was partially supported by the Ministry of Culture and Science of the State of North Rhine-Westphalia (MKW grant 005-1703-0021 "MEwM") and received funding from the EU's Horizon 2020 Program under grant agreements No 830927 (project CONCORDIA).

References

[1] J. Amann, O. Gasser, Q. Scheitle, L. Brent, G. Carle, and R. Holz, "Mission accomplished? HTTPS security after DigiNotar," in *ACM IMC*, 2017.

[2] A. Barth, "HTTP State Management Mechanism," Internet Requests for Comments, Internet Engineering Task Force, RFC 6265, April 2011. [Online]. Available: https://tools.ietf.org/html/rfc6265

[3] ——, "The Web Origin Concept," Internet Requests for Comments, Internet Engineering Task Force, RFC 6465, December 2011. [Online]. Available: https://tools.ietf.org/html/rfc6454

[4] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, "CookiExt: Patching the Browser Against Session Hijacking Attacks," *Journal of Computer Security*, vol. 23, no. 4, pp. 509–537, 2015.

[5] S. Calzavara, A. Rabitti, and M. Bugliesi, "Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild," in *ACM CCS*, 2016.

[6] ——, "Semantics-based analysis of content security policy deployment," *ACM Trans. Web*, vol. 12, no. 2, pp. 10:1–10:36, 2018.

[7] S. Calzavara, S. Roth, A. Rabitti, M. Backes, and B. Stock, "A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web," in *USENIX Security*, 2020.

[8] S. Calzavara, T. Urban, D. Tatang, M. Steffens, and B. Stock, "Site Policy Implementation," 2020, accessed: 2020-05-22. [Online]. Available: https://github.com/site-policy/site-policy

[9] "The Chromium HSTS Preload List," 2020, accessed: 2020-05-16. [Online]. Available: https://raw.githubusercontent.com/chromium/chromium/50face029619bb344b723e9e3add04dfa73a1078/net/http/transport_security_state_static.json

[10] D. Denicola and M. West, "Origin Policy," 2020, accessed: 2020-05-19. [Online]. Available: https://wicg.github.io/origin-policy/

[11] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *ACM CCS*, 2016.

[12] B. Eriksson and A. Sabelfeld, "Autonav: Evaluation and automatization of web navigation policies," in *Proceedings of The Web Conference 2020*, 2020, pp. 1320–1331.

[13] Google Inc., "HSTS Preload List," 2020, accessed: 2020-05-16. [Online]. Available: https://hstspreload.org/

[14] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," Internet Requests for Comments, Internet Engineering Task Force, RFC 6797, November 2012. [Online]. Available: https://tools.ietf.org/html/rfc6797

[15] M. Kranch and J. Bonneau, "Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning," in *NDSS*, 2015.

[16] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation," in *NDSS*, 2019.

[17] M. Luo, P. Laperdrix, N. Honarmand, and N. Nikiforakis, "Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers," in *NDSS*, 2019.

[18] A. Mendoza, P. Chinprutthiwong, and G. Gu, "Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites," in *WWW*, 2018.

[19] Mike West, "Strict CSP for Everyone," 2019, accessed: 2020-05-11. [Online]. Available: https://github.com/mikewest/strict-csp-for-everyone/

[20] B. Montagu, B. C. Pierce, and R. Pollack, "A Theory of Information-Flow Labels," in *IEEE Computer Security Foundations Symposium*, 2013.

[21] Mozilla, "The Set-Cookie HTTP Header," 2019, accessed: 2020-05-19. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie

[22] ——, "Clear-Site-Data," 2020, accessed: 2020-05-22. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Clear-Site-Data

[23] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen, "SessionShield: Lightweight Protection against Session Hijacking," in *International Symposium on Engineering Secure Software and Systems*, 2011.

[24] M. Nottingham, "Draft: Site-Wide HTTP Headers," Internet Requests for Comments, Internet Engineering Task Force, DRAFT, November 2016. [Online]. Available: https://www.ietf.org/archive/id/draft-nottingham-site-wide-headers-01.txt
[25] "Should HSTS be part of this," 2019, accessed: 2020-05-19. [Online]. Available: https://github.com/WICG/origin-policy/issues/25
[26] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "Cspautogen: Black-box enforcement of content security policy upon real-world websites," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 653–665.
[27] S. Roth, M. Backes, and B. Stock, "Assessing the impact of script gadgets on csp at scale," in *Proceedings of the 2020 ACM Asia Conference on Computer and Communications Security*, 2020.
[28] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies," in *NDSS*, 2020.
[29] J. Schwenk, M. Niemietz, and C. Mainka, "Same-Origin Policy: Evaluation in Modern Browsers," in *USENIX Security*, 2017.
[30] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the Incoherencies in Web Browser Access Control Policies," in *IEEE Symposium on Security and Privacy*, 2010.
[31] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information," in *IEEE Symposium on Security and Privacy*, 2016.
[32] D. Somé, N. Bielova, and T. Rezk, "On the Content Security Policy Violations Due to the Same-Origin Policy," in *WWW*, 2017.
[33] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *WWW*, 2010.
[34] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in) security," in *USENIX Security*, 2017.
[35] T. Urban, M. Degeling, T. Holz, and N. Pohlmann, "Beyond the Front Page: Measuring Third Party Dynamics in the Field," in *TheWebConf*, 2020.
[36] S. Van Acker, D. Hausknecht, and A. Sabelfeld, "Raising the Bar: Evaluating Origin-wide Security Manifests," in *ACSAC*, 2018.
[37] T. Van Goethem, V. Le Pochat, and W. Joosen, "Mobile Friendly or Attacker Friendly? A Large-Scale Security Evaluation of Mobile-First Websites," in *AsiaCCS*, 2019.
[38] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy," in *ACM CCS*, 2016.
[39] M. Weissbacher, T. Lauinger, and W. Robertson, "Why Is CSP Failing? Trends and Challenges in CSP Adoption," in *RAID*, 2014.
[40] World Wide Web Consortium, "Content Security Policy Level 3—Working Draft," 2018, accessed: 2020-01-11. [Online]. Available: https://www.w3.org/TR/CSP3/
[41] Y. Zhou and D. Evans, "Why Aren't HTTP-only Cookies More Widely Deployed?" in *Workshop on Web 2.0 Security and Privacy*, 2010.