How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security

Ben Stock
CISPA, Saarland University
Saarland Informatics Campus

Martin Johns
SAP SE

Marius Steffens
CISPA, Saarland University
Saarland Informatics Campus

Michael Backes
CISPA, Saarland University
Saarland Informatics Campus

Abstract

While in its early days, the Web was mostly static, it has organically grown into a full-fledged technology stack. This evolution has not followed a security blueprint, resulting in many classes of vulnerabilities specific to the Web. Even though the server-side code of the past has long since vanished, the Internet Archive gives us a unique view on the historical development of the Web's client side and its (in)security. Uncovering the insights which fueled this development bears the potential to not only gain a historical perspective on client-side Web security, but also to outline better practices going forward.

To that end, we examined the code and header information of the most important Web sites for each year between 1997 and 2016, amounting to 659,710 different analyzed Web documents. From the archived data, we first identify key trends in the technology deployed on the client, such as the increasing complexity of client-side Web code and the constant rise of multi-origin application scenarios. Based on these findings, we then assess the advent of corresponding vulnerability classes, investigate their prevalence over time, and analyze the security mechanisms developed and deployed to mitigate them.

Correlating these results allows us to draw a set of overarching conclusions: Along with the dawn of JavaScript-driven applications in the early years of the millennium, the likelihood of client-side injection vulnerabilities has risen. Furthermore, there is a noticeable gap in adoption speed between easy-to-deploy security headers and more involved measures such as CSP. But there is also no evidence that the usage of the easy-to-deploy techniques reflects on other security areas. On the contrary, our data shows for instance that sites that use `HTTPonly` cookies are actually *more* likely to have a Cross-Site Scripting problem. Finally, we observe that the rising security awareness and introduction of dedicated security technologies had no immediate impact on the overall security of the client-side Web.

1 A Historic Perspective on Web Security

The Web platform is arguably one of the biggest technological successes in the area of popular computing. What modestly started in 1991 as a mere transportation mechanism for hypertext documents is now the driving force behind the majority of today's dominating technologies. However, from a security point of view, the Web's track record is less than flattering, to a point in which a common joke under security professionals was to claim that the term *Web security* is actually an oxymoron.

Over the years, Web technologies have given birth to a multitude of novel, Web-specific vulnerability classes, such as Cross-Site Scripting (XSS) or Clickjacking, which simply did not exist before, many of them manifesting themselves on the Web's client side. These ongoing developments are due to the fact that the Web's client side is under constant change and expansion. While early Web pages were mostly styled hypertext documents with limited interaction, modern Web sites push thousands of lines of code to the browser and implement non-trivial application logic. This ongoing development shows no signs of stopping or even slowing down. The trend is also underlined by the increase in client-side APIs in the browser: while in 2006 Firefox featured only 12 APIs, it now has support for 93 different APIs ranging from accurate timing information to an API to interact with Virtual Reality devices[1]. This unrestricted growth led to what Zalewski [41] dubbed *The Tangled Web*.

Now, more than 25 years into the life of the Web, it is worthwhile to take a step back and revisit the development of Web security over the years. This allows us to gain a historical perspective on the security aspects of an emerging and constantly evolving computing platform and also foreshadows future trends.

Unfortunately, the majority of Web code is commercial and, thus, not open to the public. Historic server-

[1] A list of all available features in current browsers is available at `http://caniuse.com/`

side code that has been replaced or taken offline cannot be studied anymore. However, the Web's client side, i.e., all Web code that is pushed in the form of HTML or JavaScript to the browser is public. And thankfully, the Internet Archive has recognized the historical significance of the Web's public face early on and attempts to preserve it since 1996.

Thus, while the server-side portion of old Web applications is probably gone forever, the client-side counterpart is readily available via the Internet Archive's Wayback Machine. This enables a novel approach to historical security studies: A multitude of Web security problems, such as Client-Side XSS or Clickjacking, manifest themselves on the client side exclusively. Hence, evidence of these vulnerabilities is contained in the Internet Archive and thus available for examination. Many of the current state-of-the-art security testing methods can be adapted to work on the archived version of the sites, enabling an automated and scalable security evaluation of the historic code.

Thus, we find that the archived client-side Web code offers the unique opportunity *to study the security evolution of one of the most important technology platforms during (almost) its entire existence*, allowing us to conduct historical analyses of a plethora of properties of the Web. This way, we are not only able to investigate past Web trends, but also draw conclusions on current and future Web development trends and (in)security. In the following, we give a brief overview of our conducted study and outline our research approach.

Technological Evolution of the Web's Client Side We first examine the evolution of client-side technologies, i.e., which technologies prevailed in the history of the Web. We then systematically analyze the archived code on a syntactical level. The focus of this analysis step is on observable indicators that provide evidence on how *diversity*, *complexity*, and *volume* of this code has developed over the years, as all these three factors have a direct impact on the likelihood of vulnerabilities. Section 3 reports on our findings in this area. The overall goal of this activity is to enable the correlation of trends in the security area with ongoing technological shifts.

Resulting Security Problems With the ever-growing complexity of the deployed Web code and the constant addition of new powerful capabilities in the Web browser in the form of novel JavaScript APIs the overall amount of potential vulnerability classes has risen as well. As motivated above, several of the vulnerabilities which exclusively affect the client side have been properly archived and, thus, can be reliably detected in the historical data. We leverage this capability to assess a lower bound of vulnerable Web sites over the years. Section 4 documents our security testing methodology and highlights our key findings in the realm of preserved security vulnerabilites.

Introduction of Dedicated Security Mechanisms To meet the new challenges of the steadily increasing security surface on the Web's client side, several dedicated mechanisms, such as security-centric HTTP headers or JavaScript APIs, have been introduced. We examine if and how these mechanisms have been adopted during their lifespan. This provides valuable evidence with respect how the awareness of security issues has changed over time and if this awareness manifests itself in overall improvements of the site's security characteristics. We discuss the selected mechanisms and the results of our analysis in Section 5.

Overarching Implications of our Analysis Based on the findings of our 20-year-long study, we analyze the implications of our collected data in Section 6. By looking at historical trends and correlating the individual data items, we can draw a number of conclusions regarding the interdependencies of client-side technology and client-side security. Moreover, we investigate correlations between actual vulnerabilities discovered in historical Web applications and the existence of security awareness indicators at the time, and finish with a discussion of important next steps for Client-Side Web security.

2 Methodology

In this section, we present our methodology of using the Internet Archive as a gateway to the past, allowing us to investigate the evolution of the Web's client side (security), and outline our technical infrastructure.

2.1 Mining the Internet Archive for Historical Evidence

To get a view into the client-side Web's past, the Internet Archive (`https://archive.org`) offers a great service: since 1996, it archives HTML pages, including all resources which are included, such as images, stylesheets, and scripts. Moreover, for each HTML page, it also stores the header information initially sent by the remote server allowing us to even investigate the prevalence of certain headers over time.

For a thorough view into how the Web's client side changed over the years, we specifically selected the 500 most relevant pages for each year. Given that these are the most frequented sites of the time, they also had the greatest interest in securing their sites against attacks.

For this purpose, we analyzed the sites identified by Lerner et al. [19] as the 500 most important sites per year. For 1996, the Internet Archive only archived copies of less than half of these sites, though. Therefore, for our work, we selected the years 1997 to 2016. For each year, we used the first working Internet Archive snapshot of each domain as an entry point.

Unlike Lerner et al. [19], who investigated the evolution of tracking, though, we did not restrict our analysis to the start pages of the selected sites. Instead, we followed the first level of links to get a broader coverage of the sites. In doing so, we encountered similar issues as described in the previous work: several sites were unavailable in the archive and links often led to content from a later point in time. To allow for a precise analysis, we excluded all domains that either had no snapshot in the Archive for a given year or did not have any working subpages. Moreover, when crawling the discovered links, we excluded any that resulted in a redirect to either a more recent, cached resource or the live version of the site. Also, when a page redirected to an older version, we only allowed the date to deviate at most three months from the start page. On average, this allowed us to consider 422 domains per year [2]. On these domains, we crawled a grand total of 659,710 unique URLs, yielding 1,376,429 frames, 5,440,958 distinct scripts, and 21,169,634 original HTTP headers for our analysis. Since the number of domains varies for each year, throughout this paper we provide fractions rather than absolute numbers for better comparability.

Threats to Validity Given the nature of the data collected by the Internet Archive, our work faces certain threats to validity. On the one hand, given the redirection issues to later versions discussed above, we cannot ensure a complete coverage of the analyzed Web site, i.e., we might miss a specific page which carries a vulnerability or might not collect an HTTP header only sent when replying to a certain request, e.g., a session cookie sent after login. Moreover, since the Archive's crawler cannot log into an application, we are unable to analyze protected parts of a Web site.

The analyses of Client-Side XSS vulnerabilities are based on the dynamic execution of archived pages, for which we use a current version of Google Chrome. To the best of our knowledge, this should not be cause for over-approximation of our results. On the contrary, Internet Explorer does not automatically encode any part of a URL when accessed via JavaScript, i.e., especially in the case of Client-Side Cross-Site Scripting, our results provide a lower bound of exploitable flaws.

Nevertheless, we believe that the Archive gives us the unique opportunity to get a glimpse into the state of Web security over a 20-year time frame. Moreover, several works from the past that investigate singular issues we highlight as part of our study confirm our historical findings [16, 10, 24, 17, 31, 38].

[2]For a full list of domains see `https://goo.gl/eXjQfs`

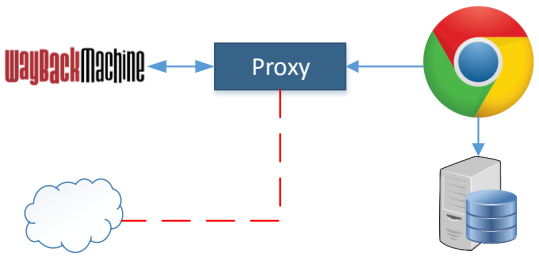

Figure 1: Infrastructure overview

2.2 Technical Infrastructure

In this section, we briefly explain the technical infrastructure used to conduct our study.

Custom Proxy and Crawlers To reduce the load on the Wayback Archive, we set up our own proxy infrastructure. Archive.org adds certain HTML and JavaScript elements to each cached page to gather statistics. In our proxy, before persisting the files to disk, we removed all these artifacts which would taint our analysis results. The proxy infrastructure is depicted in Figure 1: for crawling, we used Google Chrome. The proxy was set up such that it only allowed access to archived pages. With our crawlers, we then collected all scripts and all headers sent from the Archive servers. Note that apart from the regular HTTP headers, the Archive also sends the original headers of the site at the time of archiving, prefixed with `X-Archive-Orig-`, allowing us to collect accurate original header information.

Data Storage and Parsing We stored all information gathered by our crawlers in a central database. For data analysis, we developed several tools, e.g., to parse header information. Moreover, to analyze the collected HTML and JavaScript we employed lightweight static analysis techniques. To discover relevant HTML elements, e.g., object tags, we used Python's BeautifulSoup to parse and analyze the HTML. For JavaScript, we developed a lightweight tool based on esprima and node.js to parse JavaScript and extract features such as called APIs, parameter passed to the APIs, or statements contained in each analyzed file.

Dynamic Dataflow Analysis To automatically verify the existence of Client-Side Cross-Site Scripting issues in the archived pages, we leveraged the techniques we developed for CCS 2013 [17]. To that end, we ran their modified version of Chromium on the cached pages to gather all data flows from attacker-controllable sources to dangerous sinks, such as `document.write` or `eval`. Subsequently, we ran an exploit generator to craft URLs modified in such a way that they would allow to exploit a vulnerability. The crawlers were then sent to visit these *exploit candidates*. If indeed a vulnerability existed, this triggered the payload allowing us to automatically verify the flaw. As this is not a contribution of this work, we refer the reader to [17] for further details.

3 Evolution of Client-Side Code

In this section, we discuss how client-side active content evolved over time, showing that JavaScript remains the only prevailing programming language on the Web's client side. While in the beginning of the Web, all content was merely static and at best linking to other documents, the Web has changed drastically over the course of the years. After server-side programming languages such as PHP enabled designing interactive server-side applications, at the latest starting with the advent of the so-called Web 2.0 around 2003, client-side technology became more and more important. To understand how this client-side technology evolved over time, we analyzed the HTML pages retrieved from the Internet Archive, searching specifically for the most relevant technologies, i.e., JavaScript, Flash, Java, and Silverlight.

Figure 2 shows the technologies we discovered in the top-ranked sites in our study over time. We observe that starting from the beginning our of study in 1997, JavaScript was widely deployed, while initially Java applets could also be discovered in few cases. Generally speaking, though, Java and Silverlight did not play a significant role in active technologies used by the top sites. Over the years, JavaScript usage increased, spiking from about 60% to 85% in 2003, reaching its peak in 2009 with 98.3% of all sites using JavaScript. This number remained stable until 2016. Curiously, not all sites appear to be using JavaScript. This, however, is caused by two factors: on the one hand, the Alexa top 500 list contains a number of Content Distribution Networks, which do not carry any JavaScript on their static HTML front pages. Moreover, we found that in some cases the Archive crawler could not store the included JavaScript. As our analysis only considers *executed* JavaScript, this makes for the second part of non-JavaScript domains.

Starting from 2002, we can also observe an increase in the usage of Flash. Its share increased, also reaching its peak in 2009 with 48%. However, we also observe that the use of Flash decreases noticeably in the following years ending with only approximately 20% of the 2016 site population using it. This is in part related to modern browsers nowadays switching off Flash by default, and moreover the fact that HTML5 can be used to develop interactive advertisements instead of Flash.

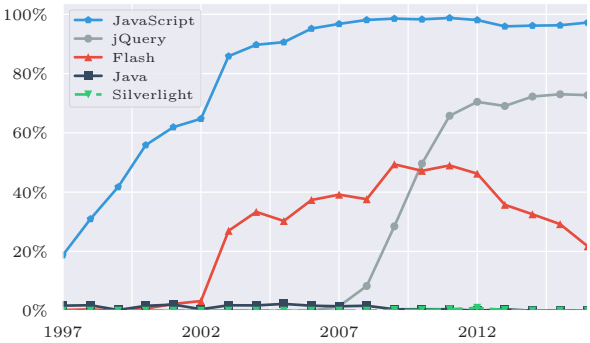

Figure 2: Technologies used by top 500 sites

In addition to the core technologies, we considered jQuery in our analysis, since it is one of the main drivers behind powerful JavaScript applications. We find that after it was first introduced in 2006, the major sites quickly picked up on its use. Until 2011, coverage quickly grew to over 65% of all sites using it, whereas by 2016, almost 75% of the major sites were using jQuery.

JavaScript as the Powerhouse of the Web 2.0 As we observed in the previous section, at least starting in 2003, JavaScript was omnipresent on the Web. To understand the magnitude of its success, we analyzed all JavaScript which was included in external scripts (not considering libraries like jQuery). We selected these instead of inline scripts (i.e., such scripts that do not have a `src` attribute, but contain the code in the script tags) as the major functionality of Web sites is mostly contained in such external scripts instead of being intermixed with the HTML code. Figure 3 shows the average number of statements in each external script by year, i.e., whenever a domain included a single external file in 2016, it contained more than 900 statements. As the figure shows, this number increased steadily over the years, while at the same time, the average number of scripts included in each frame remained stable at about four scripts per frame.

Moreover, we analyzed the Cyclomatic Complexity of all scripts per year. Designed by McCabe [22] in 1976, it measures the number of potential paths through the program, which equals the number of different test cases needed to cover all branches of the program. Figure 4 shows the results of our analysis, averaged per external script (excluding well-known libraries) in each year.

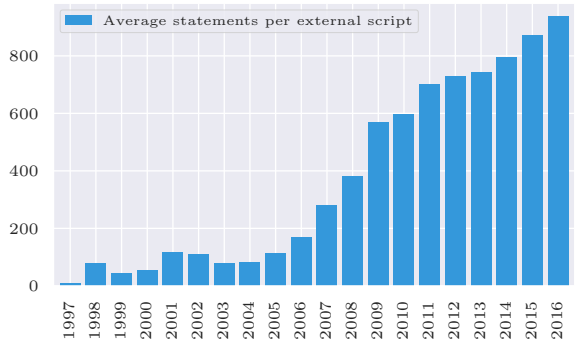

Figure 3: JavaScript Statement Statistics

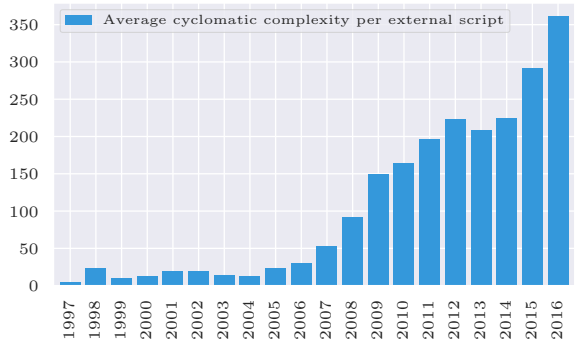

Figure 4: Cyclomatic Complexity Statistics

By 2016, each external script on average had more than 300 paths to cover. Also, the graph depicts the trend of an ever-increasing number of paths, underlining the increased complexity of modern applications.

These figures clearly show that modern JavaScript applications are more powerful than ever, but also incur a higher complexity due to the large code base to maintain.

Script Inclusions from Remote Domains Next to the amount of JavaScript code, we investigated the origin of the code. Browsers allow for Web sites to include script content from remote origins, which is often used to incorporate third-party services (e.g., for site analytics, advertising, or maps) or to reduce traffic on a site (e.g., by including jQuery from Google). However, when such remote scripts are included, they operate in the origin of the including site, i.e., they can modify both the global JavaScript state as well as the DOM. This adds more complexity to the page, since inclusion of third-party content might have side-effects, ranging from modified functionality all the way to vulnerabilities introduced by third-party code. As Nikiforakis et al. [24] have comprehensively demonstrated, the inclusion of third-party scripts has an immediate impact on a Web site's security characteristics, that scales negatively with the number of external code sources. Figure 5 shows the evolution of remote inclusions over time, plotting the number of distinct remote origins used in average domains. Starting from 2000, domains started using third-party inclusions. The trend since then is clearly pointing upwards, reaching almost 12 distinct remote origins per domain by 2016.

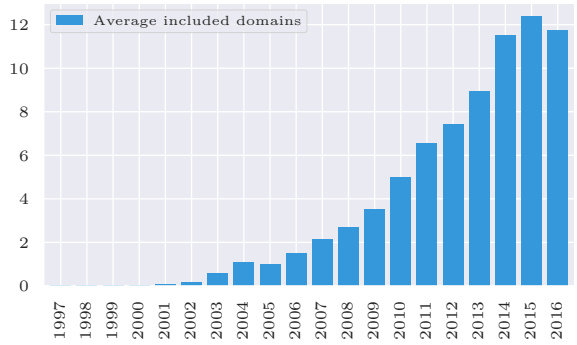

Figure 5: JavaScript Inclusion Statistics

Cross-Domain Data Access Modern Web sites are often interconnected, bringing the need for cross-domain communication and data access. However, such communication is prohibited by the Same-Origin Policy (SOP), which states that resources may only access each other if they share an origin, i.e., protocol, host, and port match [41]. To nevertheless allow applications to communicate across these boundaries, different technologies can be used. One technique to do so is called *JSONP*, short for JSON with padding. The SOP has certain exemptions, such as the fact that including scripts from a remote origin is permitted. JSONP leverages this by providing data in the form of a script, where the data is contained as JSON, wrapped in a call to a function which is typically specified as a URL parameter. This way, a site may include the script from a remote origin, effectively getting the data as the parameter to the specified callback function. There are, however, a number of security issues associated with this, such as cross-domain data leakage [18] or the Rosetta Flash attack [32]. To detect JSONP in the data, we pre-filtered all scripts in which any given URL parameter was contained in the response as a function call. Subsequently, we manually checked the results to filter out false positives. The results of our analysis are depicted in Figure 6, showing that at most about 17% of all sites used JSONP during our study timeframe. Moreover, we observe a slight decrease since 2014.

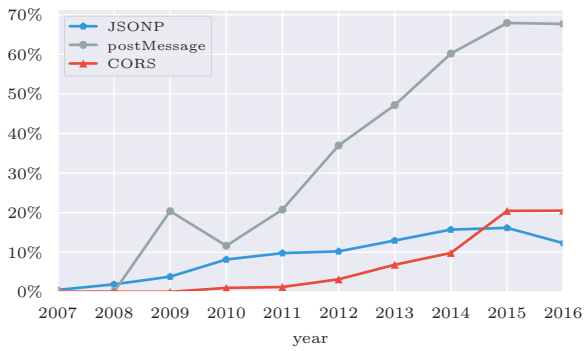

Figure 6: Cross-Origin Data Sharing Techniques

While with JSONP, the developer has to ensure that no unauthorized origin can include the endpoint, Cross-Origin Resource Sharing (*CORS*) is secure by default. CORS is a policy deployed by the server and is meant to control data access when a request is conducted with the XMLHttpRequest API [41]. By default, such a request does not carry authentication credentials in the form of cookies. If a snippet wants to do such an authenticated request across domains, the remote HTTP server has to specifically allow the snippet's origin in the `Access-Control-Allow-Origin` header; a wildcard is not sufficient to grant access. In our study, we found that CORS deployment has overtaken the use of JSONP in 2014 and has increased drastically resulting in 20% of the investigated sites to deploy such a header.

The most recent addition to the technologies which may be used for cross-domain communication, which was introduced with HTML5, is postMessage [7]. This API allows for cross-domain message exchange whenever two sites are rendered in the same browser tab (or popup window). It can be used to convey even complex JavaScript objects allowing for a seamless communication between origins. The API has gained a lot of popularity since its inception and we find that over 65% of the sites in 2016 either received postMessages or sent them.

Summary To sum up, we observe that over time, JavaScript has remained the most important scripting language on the Web. At the same time, with increasingly powerful applications, the complexity of the Web platform has risen, and new APIs are constantly added to browsers. In turn, JavaScript applications have become much more complex, showing a steady increase in the amount of code executed by the client, including code from an increasing amount of different sources, and exchanging data across the trust boundaries of the domain. Moreover, even legacy technology like Flash still remains in use by a notable fraction of sites. Thus, securing a modern Web application with all its different components is challenging. Hence, in the following sections, we analyze how security evolved over time by first discussing a number of issues we discovered, and subsequently showing which countermeasures were deployed.

4 Discovered Security Issues

Based on the technologies we identified as most prevalent in the previous section, in this section, we highlight security issues pertaining to these technologies, as discovered in our study. To that end, we report on the Client-Side XSS vulnerabilities we found, analyze the insecure usage of postMessages over time, outline the (in)security of cross-domain communication in Flash, and show the general pattern of including outdated third-party library versions.

4.1 Client-Side XSS Vulnerabilities

The term Cross-Site Scripting (XSS) was first introduced in 2000 by a group of security engineers at Microsoft [29]. At first, this issue was believed to only be caused by insecure server-side code. In 2005, Amit Klein wrote an article about what he dubbed *DOM-based Cross-Site Scripting* [12], detailing the risk of XSS through insecure client-side code. He called it DOM-based since he argued that it was caused by using attacker-provided data in interactions with the Document Object Model (DOM). Nowadays, this does not hold true anymore considering that the `eval` construct allows for JavaScript execution without the use of any DOM functionality. Hence, this type of issue is also referred to as *Client-Side Cross-Site Scripting* [34].

In contrast to Cross-Site Scripting caused by server-side code, Client-Side XSS can be discovered in the HTML and JavaScript code that was delivered to the client and in this case to the Archive crawler. Therefore, this data source allows us to investigate when the first instances of this attack occurred and how many sites were affected over the course of the last 20 years. To that end, we used an automated system developed by us to crawl the pages, collect potentially dangerous data flows, and generate proof-of-concept exploits for each of the flows [17].

Compared to our previous work, which was conducted on live Web sites, the archived data has one drawback: in case an exploit could only be triggered by modifying a search parameter, this effectively changed the URL and, hence, the corresponding page was not contained in the Archive. Therefore, for each site without a verified exploit, we sampled one potentially vulnerable flow and analyzed the JavaScript code manually. In doing so, we

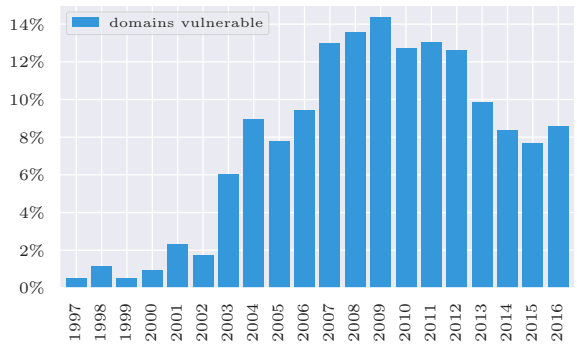

Figure 7: Client-Side XSS Vulnerabilities per year

could manually verify that 33% (142/427) of the sampled domains were in fact vulnerable.

Figure 7 shows the results of our analysis. Even back in 1997, sites were vulnerable to Client-Side XSS. We also notice a distinctive increase in vulnerabilities starting from 2003, which coincides with the advent of the Web 2.0. Moreover, the results are stable at around 12% to 14% from 2007 to 2012. In 2013, when we published our work on discovering Client-Side XSS, we found approximately 10% of the Top 10.000 sites to be vulnerable, which aligns with our findings. After 2013, the numbers slightly drop, leaving about 8% of the 2016 sites exploitable.

4.2 Insecure postMessage Handling

To allow for an easy cross-domain communication channel, sites may use postMessages to communicate between documents of differing origins. The API gives guarantees on the authenticity and confidentiality of a message — a receiver can verify the origin of the sender, and the sender may specifically target an origin ensuring that the message is not accessible by any other origin. In practice, however, these checks are often omitted [31]. We therefore analyzed our data set in two dimensions: handling of received postMessages without performing origin checks and calls to the postMessage API with a wildcard origin.

Given the large amount of data we collected, i.e., 8,992 distinct scripts, we opted to analyze the postMessage receivers in a light-weight fashion. To that end, whenever our static analysis discovered that a postMessage receiver was registered, we checked the file for access to the `origin` property of the message. Although it was shown by Son and Shmatikov [31] that the existence of an origin check does not preclude a vulnerability, we present the results as an estimation over our study period. The results of our analysis are shown in Table 1. When-

	postMessage received	no origin check	postMessage sent	wildcard target
2009	0.5%	0%	20.9%	2.2%
2010	10.8%	2.4%	5.9%	3.9%
2011	18.5%	8.4%	19.0%	14.8%
2012	32.7%	11.4%	32.7%	17.9%
2013	31.9%	21.8%	41.2%	22.8%
2014	40.0%	19.6%	52.2%	33.0%
2015	50.5%	18.1%	62.9%	45.8%
2016	48.0%	26.3%	64.1%	50.3%

Table 1: postMessage Statistics

ever a site used at least one postMessage receiver without an origin check, we marked this domain as not using the origin check. We find that the data does not show a trend towards more secure usage. On the contrary: in 2016, more than half of the pages which received postMessages had at least one receiver without an origin check.

A missing origin check does not necessarily result in a vulnerability, as pointed out by Son and Shmatikov [31]. In their work, only 13 out of 136 distinct receivers led to an actual vulnerability. Their analysis efforts, however, were mostly manual; hence, while an in-depth analysis of the discovered receivers is not feasible for our work, we leave a more automated approach to such analyses to future work.

Apart from the authenticity issue of postMessages, not specifying a target origin might endanger the confidentiality of an application's data. Table 1 shows the results of our analysis. Note that in comparison to received postMessages the numbers vary, since not every site that sends postMessages also receives them. Moreover, the high number of postMessage senders in 2009 is related to Google page ads, which featured postMessages in 2009, but removed its usage in 2010. Even though not every message with a wildcard target is necessarily security-critical, we find that by 2016, more than half of the sites we analyzed sent at least one such message. We leave further investigation of the actual exploitability of such insecure postMessages to future work.

4.3 Flash Cross-Domain Policies

Similar to JavaScript, Flash also offers APIs to conduct HTTP requests, either to the same site or across domain boundaries. To nevertheless ensure the user's protection against cross-domain data leakage, Flash tries to download a policy file (`crossdomain.xml`) from the remote origin before allowing access to that site's content. If it is missing, no data can be exchanged between the sites [35]. If it exists, the policy file can specify which origins may access the site's data, and can contain wildcards, e.g., to allow for all subdomains of a given domain

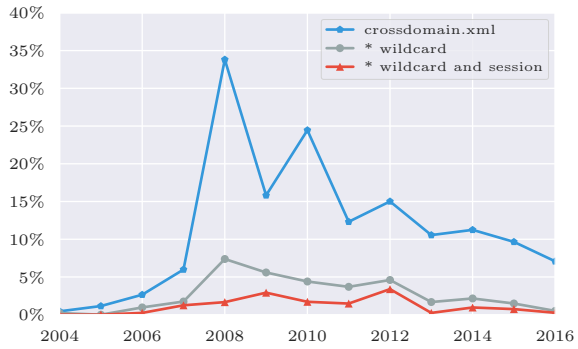

Figure 8: Crossdomain.xml files and wildcards

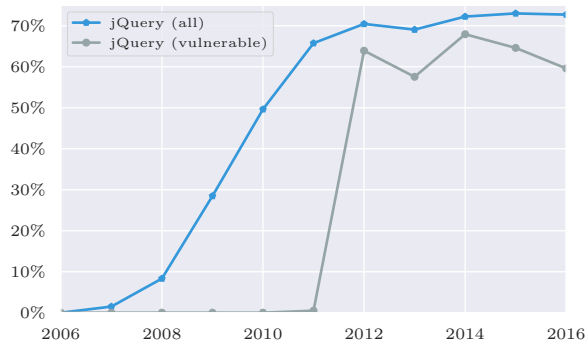

Figure 9: jQuery usage and vulnerability statistics

to gain access (`*.domain.com`). However, this wildcard can also be used to whitelist any site (`*`) or any prefix (`prefix*`), e.g., `prefix.otherdomain.com` for cross-domain access.

In part, these policy files are also stored by the Web Archive. However, we discovered a number of cases where no policy file was available [3]. Therefore, all results we present in the following must be considered lower bounds. This drawback of the archived data is also clearly visible in the number of sites hosting a `crossdomain.xml` file in 2009, as shown in Figure 8.

We analyzed the crossdomain policy files for dangerous wildcards, i.e., wildcards allowing access to any remote origin. The results are shown in Figure 8 as the grey line. We find that for 2008, about 7% of all domains had such wildcards and the number decreased afterwards (along with the general decline in the use of Flash, and hence `crossdomain.xml` files). The existence of a wildcard does not necessarily imply a vulnerability, since access might be granted to any domain by a content distribution network [16]. Hence, we also analyzed which of the domains with wildcard policies had artefacts of a login, e.g., login pages or session cookies. The result of this analysis is also shown in the graph as the red line. Here, we observe that at most 3% of all domains should be considered vulnerable, which is in line with the results presented by Lekies et al. [16] and Jang et al. [10].

4.4 Usage of Outdated Libaries

Much of the success of JavaScript stems from the powerful libraries used by many Web sites. The most widely used library on the Web is undoubtedly jQuery; in our work we found that up to 75% of the Web sites we analyzed used jQuery. The usage pattern is also shown in Figure 9. When code is included from a third party into any Web site, this code is implicitly trusted, i.e., it runs in the origin of the including Web site. Therefore, whenever any third-party component is vulnerable, this implies that all sites which include the flawed code will suffer from the vulnerability.

To understand the risk associated with this, we used retire.js [25], a tool to detect libraries and report known vulnerabilities in them, on the versions of jQuery we collected in our study. Moreover, for each domain that used jQuery we checked if the used version had a known vulnerability at the time of use. The results are also depicted in Figure 9: it becomes clear that the majority of Web sites used outdated versions of jQuery, for which known vulnerabilities existed at the time.

Although this paints a grim picture, a vulnerable library does not necessarily directly imply a site at risk. As an example, one vulnerability which was disclosed in 2012 [11] could only be triggered if user-provided input was used in a CSS selector. Nevertheless, as previous work has shown, such outdated libraries can cause severe security issues, such as Client-Side XSS [34].

Next to jQuery, only the YUI library was discovered on a notable fraction of domains. In 2011, its usage reached its peak with about 10% prevalence, dropping off until 2016 to 3.5% of the domains that included the library. Similar to what we observed for jQuery, the fraction of domains running a known vulnerable version of YUI is high: for 2016, 85% of the sites running YUI ran a vulnerable version. These results are comparable to the results of Lauinger et al. [14].

5 Indicators for Security Awareness

In this section, we highlight a number of features we can measure, that indicate whether a site operator is aware of Web security mechanisms.

Most of the security awareness indicators can be found

[3] A prime example is the facebook.com policy `http://web.archive.org/cdx/search/cdx?url=facebook.com/crossdomain.xml`, which does not have entries between 2011 and 2013

in the HTTP headers of the responses. The Web Archive records all headers it received when originally crawling the target site, which allows us to go back in time and investigate how many sites used any of the relevant headers. For our work, we identified a number of these relevant headers, which we discuss in the following in the order of their introduction on the Web. An overview over the fraction of domains that make use of these security headers is shown in Figure 10. Moreover, Table 2 shows when each of the discussed security measures was implemented by the major browsers.

5.1 HTTP-only Cookies

The Web's principle feature for session management is the use of cookies. These are sent along with every request to the server, allowing a user to establish a session in the first place and for the server to correctly attribute requests to a user. At the same time, by default, cookies are accessible from JavaScript as well, making these session identifiers prime targets for Cross-Site Scripting attacks. To thwart these attacks, starting from 2001 browsers added support for so-called *HTTP-only* cookies. This flag marks a cookie to only be accessible by the browser in an HTTP request, while at the same time disallowing access from JavaScript.

We mark a domain as using HTTP-only cookies when at least one cookie was set using the `httponly` flag. This only represents a lower bound for the sites, though. Naturally, the Archive crawler does not log into any Web application. It is reasonable to assume that some sites do not use session identifiers until the need arises, i.e., a user successfully logs in. Hence, sites might have made use of the header more frequently, but the archived data cannot account for such behavior.

In our study, we found that sites started employing this technique in 2006 before any other security measures were in place. Moreover, we can clearly observe a trend in which by 2016, over 40% of all domains made use of this. This indicates that the admins of the most relevant sites on the Web are well-aware of the dangers of cookie theft and try to mitigate the damage of an XSS attack.

	Chrome	IE	Firefox
HTTP-only Cookies	2008	2001	2006
Content Sniffing	2008	2008	2016
X-Frame-Options	2010	2009	2009
HSTS	2010	2015	2010
CSP	2011	2012	2010

Table 2: Browser support for Web security features

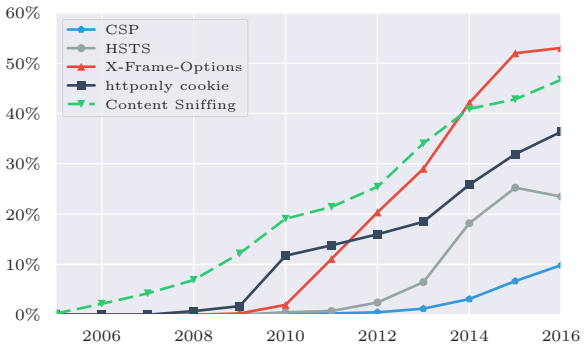

Figure 10: Use of Security Headers per year

5.2 Disallowing Content Sniffing

Although all markup and programming languages used in the Web are well-specified, Web application developers often make more or less subtle mistakes when building their sites. To still allow users of these sites an unhindered view on the pages, modern browsers are very error-tolerant, i.e., they compensate for a number of mistakes which can be introduced by developers. One of the mechanisms used to achieve this tolerance is content sniffing, a technique used by browsers to guess the type of content being shown, to allow for proper rendering. The HTTP/1.1 standard specifically states that such sniffing should only happen when no `Content-Type` header is sent from the server [5]. Depending on the implementation of the browser, this can either can be done by analyzing the URL (e.g., looking for a .html suffix) or by investigating the content of the resource [41].

For the sake of presentation, let us assume a Web site offers users a way to upload text files (marked by a .txt suffix). In this case, a user can upload a text file containing HTML code. If a victim's browser is now lured to the URL hosting the .txt file and no explicit `Content-Type` header is sent, modern browsers will analyze the content, deem it to be HTML and render it accordingly. This effectively results in the attacker's HTML and accompanying script code to be executed under the origin of the vulnerable site, leading to a Cross-Site Scripting vulnerability. In addition, improper content sniffing could also lead to the site being used to host malware.

To prevent such attacks, Internet Explorer first implemented the `X-Content-Type-Options` header in 2008 [15]. When the value of this header was set to `nosniff`, it would prevent IE from trying to guess the content. In the specific case, Internet Explorer's algorithm was also more aggressive than RFC2616 demanded: it tried to sniff content regardless of the presence of any `Content-Type` headers. Google Chrome showed a similar behavior, which can also be controlled

using the X-Content-Type-Options header. Similar to what we observed for HTTP-only cookies, at first only few sites adopt this security mechanism. Again, a notable increase can be observed over time, resulting in almost 47% of the analyzed sites using the protective measure by 2016.

5.3 Clickjacking Protection

Another potential danger to Web applications is so-called *Clickjacking* [9]. This type of attack is a sub class of the more general attack dubbed *Unsolicited Framing* by Zalewski [41]. The main idea relies on the ability of an attacker to mask a frame pointing to a benign-but-buggy site with the opacity CSS attribute on his own site. The attacker now tries to motivate the victim to click in the area in which this hidden frame resides. This can, e.g., be achieved by a pretend browser game. However, instead of interacting with the apparent browser game, the victim actually clicks in the hidden frame. The extend of this attack can range from invoking actions, such as soliciting likes on a social media site, all the way to an attack outlined by Jeremiah Grossman, in which Clickjacking was used to gain access to the video and audio feed from the victim's computer [6].

While the unsolicited framing itself was already discussed before the devastating demonstration by Grossman, the clear attack potential as shown by their attack in 2008 motivated browser vendors to develop and deploy a protective measure, dubbed the X-Frame-Options header (for short also *XFO*). Even though the X in the name denotes the fact that this was not a standardized header, it was introduced within a few months after the presented attack by Internet Explorer and Firefox, while Chrome followed a year later (see Table 2). The notion of this header is simple: framing can either be completely blocked (DENY), only be allowed from the same origin (SAMEORIGIN), or specifically allowed for certain URLs (ALLOW-FROM url) [23]. Depending on the browser, there also exists a variant ALLOWALL, which effectively disables any protection as well as SAMEDOMAIN, which is an alias for SAMEORIGIN. Note, however, that these values are not presented in the accompanying RFC, which was introduced in 2013 [28].

For our measurements, we only counted sites which use the protective measure by either setting it to DENY, SAMEORIGIN, its alias SAMEDOMAIN, or ALLOW-FROM with a specific list of domains. The results are depicted in Figure 10. The results indicate that although the header was introduced in 2010, an increase in its usage can only be observed starting from 2012. The number of sites using XFO increased rapidly since then and reached its peak in 2016 with 53% of all sites deploying it. Note, however, that use of the header has been deprecated by Content Security Policy (CSP) Level 2 [39] starting from around 2015, being replaced by the more powerful frame-ancestors directive of CSP.

5.4 Content Security Policy

One of the biggest client-side threats to any Web application is the injection of markup, either HTML or even JavaScript code, into it. In such a case, the browser cannot distinguish between markup originating from the developer of the application and the attacker's code. Hence, all code is executed, leading to a client-side code injection known as Cross-Site Scripting. To mitigate the exploitability of such an injection vulnerability, the W3C has proposed the so-called *Content Security Policy*. In its foundation, CSP is a technique that aims to specifically whitelist sources of code with the goal of stopping any attacker-provided code from being executed. To that end, a Web application that deploys CSP sends a header containing a number of whitelisted code origins, e.g., self or cdn.domain.com. Even if an attacker manages to inject her own markup into the application, the code is bound to be hosted on either the site itself or cdn.domain.com. The main assumption here is that the attacker is unable to control any code on these origins. Also, by default, CSP disallows the use of inline script elements and the eval construct.

CSP has many more directives, allowing Web developers to control which hosts may be contacted to retrieve images or stylesheets, specifying how the site may be framed (deprecating the X-Frame-Options header), or to report violations of the policy. The setup of a properly working policy, however, is non-trivial, as has been shown by previous work [37, 38]. Nevertheless, we deem the presence of a CSP header to be an indicator for the awareness of a site's operator. Given the results from previous work, investigating the security of the policies of single sites is out of scope for our work.

Initially, CSP was introduced by Firefox and WebKit-based browsers (including Chrome) with different names, i.e., X-Content-Security-Policy and X-WebKit-CSP, respectively. We therefore count the presence of these headers as a regular use of the nowadays standardized Content Security Policy. As we can observe in Figure 10, even though implemented in browsers for a number of years, CSP only was used in the wild starting from 2013 by any of the major sites. As previous work has shown, setting up CSP for legacy applications is very challenging. Our data indicates that even by 2016, less than 10% of the sites we considered deployed any CSP at all. Hence, although CSP mitigates the effect of XSS vulnerabilities in JavaScript-enabled Web applications, its adoption still lags far behind other security measures.

5.5 HTTP Strict Transport Security

Along with the success of the Web as the number one platform for information access also came a number of attacks on the connection between client and server. While in the Web's beginning, transfer of sensitive information was less likely to occur, modern Web applications almost always require a login. Arguably, the transport of such sensitive information should be conducted in a secure manner, i.e., should always be encrypted. On the other hand, network attackers have an interest to gain access to such information. To that end, they might either eavesdrop (in case of a passive network attacker) on a plaintext connection, or try to manipulate a connection to an extend where the encryption is dropped, e.g., by SSL stripping attacks [13]. In addition, Web developers might accidentally transmit sensitive information over insecure channels. An example of this is the use of cookies without specifically setting the `secure` flag. In that case, the cookies are transferred in any connection to the domain for which they were set, regardless of the use of HTTPS.

To ensure that neither an active attacker can strip SSL nor an unknowing developer can accidentally build an insecure application, browsers implement HTTP Strict Transport Security, or HSTS for short [8]. With this HTTP header, browsers can be instructed to only connect to a domain via HTTPS regardless of the URL and to only do so using a validated certificate.

Obviously, HSTS is only a relevant feature for any site that runs via HTTPS. The Archive.org crawler, however, does not store whether a site was retrieved via HTTP or HTTPS, i.e., based on the historical data we gathered, we cannot decide whether a site was running HTTPS in the first place. Also, setting an HSTS header on an unencrypted connection has no effect, i.e., it is ignored by the browser [13].

Support for HSTS was first introduced in Chrome and Firefox in 2010. In addition to the header, browsers also feature a preload list of domains, to which only HTTPS connections are allowed, regardless of the existence of the HSTS header. For our analysis, we therefore considered both the headers as well as entries for the domains in the preload list for January of each year. Our analysis shows that only very few domains made use of HSTS until 2012. Starting from 2013, we observe a steady increase, resulting in almost 30% adoption rate by 2016.

5.6 Additional Indicators for Security Awareness

On top of the headers we discussed so far, we identified additional features which indicate awareness of potential security problems. In 2010, Bates et al. [2] showed that the built-in XSS filter of Internet Explorer could not only be bypassed by encoding data in UTF-7, but even be used to disable Clickjacking protections or conduct phishing attacks. At that time, Internet Explorer allowed Web sites to specifically disable its XSS filter by sending the `X-XSS-Protection: 0` header to the client. In our study we found that in 2009 and 2010, 30 and 55 sites, respectively, disabled the XSS filter in IE by sending this header. For 2009, all these sites were related to Google (e.g., including Youtube), showing that the issues in IE were known to Google before the publication in 2010. One reasonable explanation is that Google engineers were confident that no XSS vulnerabilities were contained in their sites, and wanted to ensure that no vulnerabilities could be introduced into otherwise bug-free sites.

A more recent feature for securing the client side is the sandbox feature of iframes in HTML5. Using this feature, a site may restrict the capabilities of an iframe, e.g., by disabling JavaScript or isolating content in a unique origin, thereby mitigating any exploitable vulnerabilities in the sandboxed content. We found that only three sites made use of it in any of the HTML pages we analyzed, showing that this feature is hardly used.

6 Key Insights

In this section, we discuss the key insights of our study results. We first discuss the takeaways on client-side technology, following with the implications of our analysis for client-side security. Finally, we investigate the correlation between discovered vulnerabilities and the awareness indicators outlined in the previous section.

6.1 Client-Side Technology

The Web's Complexity is *still* on the Rise In our study of the Web's evolution, we found that although several technologies for client-side interaction were developed over the years, the only prevailing one is JavaScript. Moreover, we determined that the general complexity of JavaScript kept rising over the years. On the one hand, the average number of statements per external script has almost reached 1,000 by 2016 — without counting powerful libraries such as jQuery. On the other hand, code does not necessarily only originate from a site's developer, but often resides on remote domains. In our work, we found that on average, a domain in our 2016 dataset included script content from almost twelve distinct origins, which is an increase by almost 100% since 2011. Along with the introduction of powerful new APIs in the browsers, which nowadays, e.g., allow for client-to-client communication that was never envisioned by the Web's server/client paradigm, we find that the general complexity of client-side Web applications is on the rise.

Involvement of Third Parties Including content from third parties allows Web sites to outsource certain parts of their application, e.g., advertisements. However, whenever code is included from a remote domain, it may contain vulnerabilities, which effectively compromise the including site. With the rise in complexity in these third-party libraries, the risk for vulnerabilities also increases. As an example, while jQuery 1.0 only contained 768 statements, the most recent version 1.12 (in the 1.* branch) already consists of 3,541 statements. Moreover, sites rarely update their third-party components. As shown by Lauinger et al. [14], a large number of Web sites use outdated versions of well-known libraries, such as jQuery, which contain exploitable flaws. In our work, we found that especially jQuery was often used in versions with known vulnerabilities at the time of use. Moreover, the fraction of domains that use such vulnerable third-party libraries remained high since 2012.

Another risk of including third-party code stems from the fact that this code may be arbitrarily altered by the remote site or in transit. In the recent past, this was used to conduct large-scale DDoS attacks against Web sites used to bypass censorship in China [21]. However, such attacks can be stopped if sites start implementing Subresource Integrity, which ensures that included JavaScript is only executed when its has the correct checksum [1].

The Rise of the Multi-Origin Web The Web's primary security concept is the Same-Origin Policy, which draws a trust boundary around an application by only allowing resources of the same origin to interact with one another. In the modern Web, though, applications communicate across these boundaries, e.g., between an advertisement company and the actual site. In our work, we observed a clear trend towards interconnected sites, especially using postMessages, which are used by more than 65% of the Web sites we analyzed for 2016. In addition, we note that the usage of CORS is on the rise as well, with 20% of the 2016 domains sending a corresponding header. Given the nature of the Internet Archive crawler, i.e., the fact that it cannot log in to any applications, all these numbers need to be considered lower bounds. Hence, we clearly identify a trend towards an interconnected, multi-origin Web in which ensuring authenticity of the exchanged data is of utmost importance.

6.2 Client-Side Security

Client-Side XSS Remains a Constant Issue One of the biggest problems on the Web is Cross-Site Scripting. In our work, we studied the prevalence of the client-side variant of this attack over the years. We found that with the dawn of more powerful JavaScript applications as a result of so-called Web 2.0, the number of XSS vulnerabilities in the JavaScript code spiked. Between 2007 and 2012, more than 12% of the analyzed sites had a least one such vulnerability. Even though the general complexity of JS applications kept rising after 2012, the number of vulnerable domains declined, ranging around 8% until 2016. Nevertheless, given our sample of the top 500 pages, such attacks still threaten a large fraction of the Web users and developer training should focus more on these issues.

Security vs. Utility Many new technologies introduced in browsers come with security mechanisms, such as the authenticity and integrity properties provided by the postMessage API. However, oftentimes these features are optional — a developer may, e.g., choose not to check the origin of an incoming postMessage. As we observed in our work, technology which enables communication across domain boundaries, such as JSONP, Flash's ability to access remote resources, or postMessages, is often used without proper security considerations. Especially in the context of postMessages, this is a dangerous trend: more than 65% of the sites we analyzed for 2016 either send or receive such messages, with a steady increase in the previous years (see Figure 6). As shown by Son and Shmatikov [31], improper handling of postMessages can result in exploitable flaws. Hence, any technology added to browsers should default to a secure state similar to CORS.

Complexity of Deploying Security Measures During the course of our study, a number of new security mechanisms were introduced in browsers. In our analysis, we found that the rate of adoption varies greatly for the different technologies. As an example, within two years of being fully supported by the three major browsers, the `X-Frame-Options` header was deployed by 20% of the sites we analyzed, within four years its adoption rate even reached more than 40%. In contrast, although CSP has been fully supported since 2012, even after four years, only about 10% of the sites we analyzed deployed such a policy. The main difference between the two types of measures is the applicability to legacy applications: XFO can be selectively deployed to HTML pages which might be at risk of a clickjacking attack. In contrast, CSP needs to be adopted site-wide to mitigate a Cross-Site Scripting. Moreover, previous work [37, 38] has shown that deploying a usable CSP policy is non-trivial, especially considering the multitude of third-party components in modern Web apps. In contrast, even though deprecated by CSP, the X-Frame-Options header still shows increased usage in 2016. Hence, we find that the more effort needs to be put into securing a site with a specific security mechanism, the less likely sites in our study were to adopt the mechanism.

6.3 Correlating Vulnerabilities and Awareness Indicators

To understand whether there is a correlation between actual vulnerabilities and the general understanding of security concepts for the Web, we compared the set of sites vulnerable against Client-Side XSS attacks with their use of security indicators. The intuition here is that the use of a security indicator implies a more secure site. We chose Client-Side XSS specifically, since a vulnerability can be proven, whereas it is, e.g., unclear if usage of an outdated library could actually lead to an exploit.

The results of this analysis are shown in Figure 11: for each indicator we checked how many sites were vulnerable against a Client-Side XSS attack. To reduce noise, we only included an indicator for a given year if it was present on at least 10% of the analyzed sites. Hence, we exclude all years before 2009, since (as shown in Figure 10) no security measure was deployed on at least 10% of the sites. For each year, we plot the fraction of sites which carry the indicator *and* are susceptible to XSS. In addition, the graph shows the baseline as all sites that do not have any indicator.

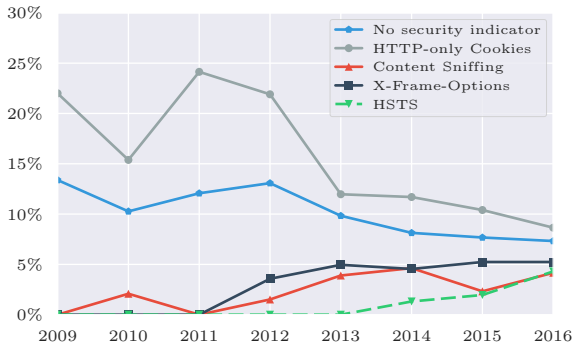

Figure 11: Security Headers vs. Client-Side XSS

HTTP-only Cookies When considering the `httponly` cookie flag, the results are surprising: In our dataset, its presence actually correlates with a *higher* vulnerability ratio compared to cases in which no indicators are found. Note however, that our study focusses on a small data set of 500 domains per years, and hence the results are not statistically significant. Even though the overall numbers are too small to produce significant results, this trend is counter-intuitive. We leave a more detailed investigation of this observation by analyzing a large body of sites to future work. It is noteworthy, however, that previous work from Vasek and Moore [36] investigated risk factors for server-side compromise, and found that `httponly` cookies are negative risk factors. Similar to our work, however, there findings were inconclusive due to a limited sample set. Comparing server- and client-side vulnerabilities with respect to the use of `httponly` cookies, however, is an interesting alley for future work.

The correlation between `httponly` cookies and increased fraction of vulnerabilities might be caused by several reasons: applications that use session cookies are more likely to have a larger code base and thus, more vulnerabilities. For 2011, the year with the highest fraction of vulnerable sites, we therefore analyzed the average number of instructions for domains with `httponly` cookies and found that they only have a code base which is about 10% larger than an average Web site, with a comparable average cyclomatic complexity. Another potential reason for our findings might be the fact that developers underestimate the dangers of an XSS vulnerability. Although taking over a session of a user might be considered the worst-case scenario which is averted by HTTP-only cookies, attackers may leverage an XSS exploit for many other attacks, e.g., XSS worms or stealing passwords from password managers [33].

Early Adopters For X-Frame-Options and HSTS (considered from 2010 and 2013 respectively) we see another trend: early adopters of new security mechanisms are less likely to be susceptible to Client-Side XSS attacks, even though the code bases for these sites are also about 10% larger than an average site. We find that for both XFO and HSTS, the first two considered years show no vulnerabilities. However, until the end of our study in 2016, more and more sites deploy both headers, resulting in vulnerability rates comparable to sites without security indicators. Hence, we find that the late adopters of such new technologies are more likely to introduce Client-Side XSS vulnerabilities in their sites.

CSP Deployment Another insight here is the fact that not a single site using CSP had a vulnerability, even leaving out the 10% threshold discussed above. It is important to note that even a valid CSP policy would not have stopped exploitation of a Client-Side XSS issue: since our analysis was conducted on the Archive.org data, CSP policies would not be interpreted by the browser since they all carried an `X-Archive-Orig-` prefix. The reason for lack of Client-Side XSS on these sites is likely twofold: either companies invested enough in their security to go through the tedious process of setting up CSP in general have better security practices, or this again shows the early adoption effect we observed for XFO and HSTS.

6.4 Going Forward

In our study, we found a number of recurring patterns in the Web's insecurity, i.e., that deployment heavily hinges on the ease of use, optional security mechanisms are rarely used, and that several vulnerabilities are still in existence even though they have been known for many years. Therefore, in the following, motivated by our findings, we discuss how Web security can move forward.

Ease of Use Considering the security technologies we investigated, we find that regardless of the potential benefit, a security measures adoption rate is controlled mostly by the ease of its deployment. While CSP allows very fine-grained control over resources that can be accessed and — more importantly — script code which can be executed, setting up a working CSP policy is often tough. Apart from this hurdle, significant changes on the application itself are required. This high effort must be considered a big roadblock for CSP's success. In contrast, headers like HSTS or XSO, which are easy to deploy and address a single issue, are adopted more swiftly in a shorter timeframe. Thus, we argue that for future techniques *ease of use* should be a primary design concern.

Make Security Mandatory Our findings highlight that if security checks are optional, they are oftentimes not used, as evidenced, e.g., by the lack of origin checking on postMessages. Moreover, if there is an easy way to ensure utility, e.g., through using a wildcard, developers tend to follow that path. Hence, we argue that new technology should ship with mandatory security, which either does not allow for generic wildcards or in that case, following the approach taken by CORS, limit the privileges of an operation. Also, existing APIs could be changed to at least warn developers. As an example, accessing the `data` property of a postMessage without a prior access to the `origin` property could result in a JavaScript console warning for a potentially missing origin check. Future generations of APIs could extend this behavior to throw security exceptions, in case crucial security checks have been omitted.

Ensure Better Developer Training The results of our study indicate that although security measures exist to prevent or mitigate attacks, developers are often unaware of the underlying security issues. Examples for this include (missing) origin checking on postMessages, the ineffective use of HTTP-only cookies, or the inclusion of user-controllable data in the generation of script code, which causes Client-Side XSS. Especially Client-Side Cross-Site Scripting appears a class of vulnerability that remains unresolved — even in light of mitigating technologies like CSP. We therefore argue that research should continue to investigate how developers can be better educated on security issues and how development tools can be designed in a way that they empower their users to build secure applications.

7 Related Work

Our work touches on many areas of Web security. In the following, we discuss research related to our work.

Large-Scale Analysis of Web Security and Privacy Several papers have conducted large-scale analyses of different aspects of Web security. Yue and Wang [40] analyzed inclusions of external scripts and investigated dangerous API calls. In 2010, Zhou and Evans [42] investigated the use of HTTP-only cookies finding that only 50% of the investigated sites make use of the feature. In 2011, two works analyzed the use of crossdomain policies for Flash, as well as other cross-domain communication channels [16, 10], which align with the results we presented for that time. In the same year, Richards et al. [27] provided the first large-scale analysis of the (mis)use of `eval`, showing that while it can be replaced in certain cases, removing it all-together is impossible. In 2012, Nikiforakis et al. [24] examined Javascript inclusions over time of the Alexa top 10.000, pointing out the trend of an evermore increasing amount of external inclusions, which we also observed in our work. In the area of privacy, Lerner et al. [19] conducted an analysis of how trackers evolved over time, also using data from archive.org.

Vulnerability Detection in the Wild In addition to the previously discussed papers, several works have focussed on examining a certain type of vulnerability in the wild. In 2013, Son and Shmatikov [31] analyzed insecure usage of postMessage receivers finding several exploitable issues. In the same year, we presented an automated system to measure the prevalence of Client-Side Cross-Site Scripting in the wild [17]. More recently, Lauinger et al. [14] performed an in-depth analysis of the usage of vulnerable libraries in the wild, showing results comparable to our historical view.

Content Security Policy An area that has gained more attention over the last years is the Content Security Policy. While Doupé et al. [4] showed in 2013 that automatically separating code and data is feasible for ASP.net applications, Weissbacher et al. [38] conducted a long-term study which indicated that CSP was not deployed at scale. Moreover, they discussed the difficulties in setting up CSP for legacy applications. In 2016, Pan et al.

[26] showed that automatically generating CSP policies for the Alexa top 50 is feasible. In the same year, Weichselbaum et al. [37] investigated the efficacy of deployed CSP policies in the wild, highlighting that around 95% of the examined policies are susceptible to bypassing. Moreover, though, the authors propose an extension to CSP to allow for easier deployment.

HTTPS Over the last year, the research community also has focussed more on HTTPS. Clark and van Oorschot [3] systematically explored issues in the area of HTTPS in terms of infrastructure as well as attack vectors against HTTPS in general. Later on, Liang et al. [20] examined the relation between the usage of HTTPS and the embedding of CDNs into Web pages. Most recently, Sivakorn et al. [30] presented an overview of the privacy risks of exposing non authenticating cookies over HTTP, leading to intrusions of end-user privacy.

8 Conclusion

In this paper, we conducted a thorough study on the security history of the Web's client side using the preserved client-side Web code from the Internet Archive. In the course of our study, we were able to observe three overarching developments: For one, the platform complexity of the client-side Web has not plateaued yet: Regardless of the numerical indicator we examine, be it code size, number of available APIs, or amount of third-party code in web sites, all indicators still trend upwards.

Furthermore, the overall security level of Web sites is not increasing noticeably: Injection vulnerabilities found their way onto the client side in the early years of the new millennium and show no sign of leaving. Vulnerabilities that are on the decrease, due to deprecated technology, as it is the case with insecure `crossdomain.xml` policies, appear to be seamlessly replaced with insecure usage of corresponding new technologies, e.g., insecure handling of postMessages.

Finally, we could observe a steady adoption of easy to deploy security mechanisms, such as the `HTTPOnly`-flag or the `X-Frames-Option` header. Unfortunately, this trend does not apply to more complex security mechanisms, such as the `Content-Security-Policy` or sandboxed iframes. Furthermore, we found that while early adopters of dedicated security technologies are overall less likely to exhibit vulnerabilities, this does not apply into the extended lifetime of the mechanism – late adopters appear to have no inherent security advantage over average sites despite their demonstrated security awareness.

Overall, these results paint a sobering picture. Even though Web security has received constant attention from research, security, and standardization communities over the course of the last decade, and numerous dedicated security mechanisms have been introduced, the overall positive effects are modest: Client-Side XSS stagnates at a high level and potentially problematic practices, such as cross-origin script inclusion or usage of outdated JavaScript libraries are still omnipresent. At best, it appears that the growing security awareness merely provides a balance to a further increase in insecurity, caused by the ever-rising platform complexity.

Thus, this paper provides strong evidence, that the process of making the Web a secure platform is still in its infancy and requires further dedicated attention to be realized.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback and our shepherd Nick Nikiforakis for his support in addressing the reviewer's comments. This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345).

References

[1] Devdatta Akhawe, Frederik Braun, François Marier, and Joel Weinberger. Subresource integrity. `https://www.w3.org/TR/SRI/`, Jun 2016.

[2] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.

[3] Jeremy Clark and Paul C van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Security and Privacy*, 2013.

[4] Adam Doupé, Weidong Cui, Mariusz H Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. deDacota: toward preventing server-side xss via automatic code and data separation. In *CCS*, 2013.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[6] Jeremiah Grossman. Clickjacking: Web pages can see and hear you. `http://blog.jeremiahgrossman.com/2008/10/clickjacking-web-pages-can-see-and-hear.html`.

[7] Ian Hickson. HTML5 Web Messaging. https://www.w3.org/TR/webmessaging/, May 2015.

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

[9] Lin-Shung Huang, Alexander Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX*, 2012.

[10] Dongseok Jang, Aishwarya Venkataraman, G Michael Sawka, and Hovav Shacham. Analyzing the crossdomain policies of Flash applications. In *W2SP*, 2011.

[11] jQuery Bug Tracker. SELECTOR INTERPRETED AS HTML. http://goo.gl/JNggpp, 2012.

[12] Amit Klein. DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium, Articles*, 2005.

[13] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in mid-air. In *NDSS*, 2015.

[14] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *NDSS*, 2017.

[15] Eric Lawrence. IE8 security update VI: Beta 2 update. https://blogs.msdn.microsoft.com/ie/2008/09/02/ie8-security-part-vi-beta-2-update/, September 2008.

[16] Sebastian Lekies, Martin Johns, and Walter Tighzert. The state of the cross-domain nation. In *W2SP*, 2011.

[17] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*, 2013.

[18] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The unexpected dangers of dynamic javascript. In *USENIX Security*, pages 723–735, 2015.

[19] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *USENIX Security*, 2016.

[20] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. When HTTPS meets CDN: A case of authentication in delegated service. In *IEEE Security and Privacy*, 2014.

[21] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ronald Deibert, and Vern Paxson. China's great cannon. *Citizen Lab*, 2015.

[22] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1976.

[23] Mozilla Firefox Team. X-frame-options. https://developer.mozilla.org/en/docs/Web/HTTP/Headers/X-Frame-Options.

[24] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *CCS*, 2012.

[25] Erlend Oftedal. Retire.js - identify JavaScript libraries with known vulnerabilities in your application. http://goo.gl/r4BQoG, 2013.

[26] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. CSPAutoGen: Black-box enforcement of content security policy upon real-world websites. In *CCS*, 2016.

[27] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - A large-scale study of the use of eval in javascript applications. In *ECOOP*, 2011.

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

[29] David Ross. Happy 10th birthday cross-site scripting! http://blogs.msdn.com/b/dross/archive/2009/12/15/happy-10th-birthday-cross-site-scripting.aspx, 2009.

[30] Suphannee Sivakorn, Iasonas Polakis, and Angelos D Keromytis. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *IEEE Security and Privacy*, 2016.

[31] Sooel Son and Vitaly Shmatikov. The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In *NDSS*, 2013.

[32] Michele Spagnuolo. Abusing JSONP with rosetta flash. https://miki.it/blog/2014/7/8/abusing-jsonp-with-rosetta-flash/, August 2014.

[33] Ben Stock and Martin Johns. Protecting users against XSS-based password manager abuse. In *AsiaCCS*, 2014.

[34] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *CCS*, 2015.

[35] Apurva Udaykumar. Setting a crossdomain.xml file for HTTP streaming. `http://www.adobe.com/devnet/adobe-media-server/articles/cross-domain-xml-for-streaming.html`.

[36] Marie Vasek and Tyler Moore. Identifying risk factors for webserver compromise. In *Financial Crypto*, 2014.

[37] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live CSP! On the insecurity of whitelists and the future of Content Security Policy. In *CCS*, 2016.

[38] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why is CSP failing? trends and challenges in csp adoption. In *RAID*, 2014.

[39] Mike West, Adam Barth, and Dan Veditz. Content security policy level 2, W3C candidate recommondation. `https://www.w3.org/TR/2015/CR-CSP2-20150219/`, February 2015.

[40] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *WWW*, 2009.

[41] Michal Zalewski. *The tangled Web: A guide to securing modern web applications*. No Starch Press, 2012.

[42] Yuchen Zhou and David Evans. Why aren't HTTP-only cookies more widely deployed. *W2SP*, 2010.