We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS

It hasnt been published, please keep it secret and do not disseminate. Thank you.

Abstract

The default Same Origin Policy essentially restricts access of cross-origin network resources to be "write-only". However, many web applications require "read" access to contents from a different origin, so developers have come up with workarounds, such as JSON-P, to bypass the default Same Origin Policy restriction. Such ad-hoc workarounds leave a number of inherent security issues. CORS (cross-origin resource sharing) is a more disciplined mechanism supported by all web browsers to handle cross-origin network accesses. This paper presents our empirical study about the real-world uses of CORS. We find that the design, implementation, and deployment of CORS are subject to a number of new security issues: 1) CORS relaxes the cross-origin "write" privilege in a number of subtle ways that are problematic in practice; 2) CORS brings new forms of risky trust dependencies into web interactions; 3) CORS is generally not well understood by developers, possibly due to its inexpressive policy and its complex and subtle interactions with other web mechanisms, leading to various misconfigurations. Finally, we propose protocol simplifications and clarifications to mitigate the security problems uncovered in our study.

1 Introduction

Same origin policy (SOP) is the foundation for client-side web security. It guards web resources from being accessed by scripts from another origin. The default SOP does not provide an explicit access control authorization mechanism to share cross-origin network resources. Under the SOP, client-side scripts are free to send GET or POST requests to third-party servers by referencing other websites' resources or submitting cross origin forms, but they have no simple and safe mechanism to read those responses, even from an origin willing to share. Because many web applications have the need to read cross-origin network resources and browsers did not have any good support for it, developers proposed some ad-hoc mechanism to serve the need. For example, JSON-P [12] uses the exception that an imported cross-origin JavaScript is accessible to workaround the restriction. But such a workaround approach introduces a number of inherent security issues.

Cross origin resource sharing (CORS) is proposed to solve the problems of JSON-P, and to provide a protocol support of authorized access cross-origin network resources. This protocol has been adopted by major browsers (e.g., Chrome, Firefox, IE) since 2009, and has been widely used in mainstream websites. Our work aims to provide a comprehensive security analysis of CORS in its protocol design, implementation, and deployment process, and to identify new types of security issues about the deployments of CORS in real websites.

The issues we found in this study can be classified into three categories: *a) Overly permissive cross origin sending permissions.* The CORS protocol enables new default sending permissions inadvertently, giving attackers more capabilities that lead to new security issues. We found that by leveraging this relaxed sending permission, an attacker could exploit previously unexloitable CSRF vulnerabilities, remotely infer victim's accurate cookie size of *any* website, or use a victim's browser as a stepstone to attack binary protocol services inside victim's internal network. *b) Inherent security risks of CORS.* The functionality of CORS needs resource servers to trust third-party domains and share resources. Such a trust dependency on third-party websites increases attack surfaces and introduces new security risks. We found that an attacker can leverage this inherent risk to launch MITM attack against HTTPS sites or steal secrets on strongly secured target sites by exploiting vulnerabilities on weak websites. *c) Complex CORS details*

and various misconfigurations. While CORS's general process is simple, there are certain error-prone details leading to a number of misconfigurations and security issues in the real world. By conducting a large-scale measurement on Alexa top 50,000 websites including their 97,199,966 distinct sub-domains, we found insecure CORS misconfigurations in 132,476 sub-domains, accounting for *27.5%* of all the CORS configured sub-domains. Some of these domains serve popular websites, such as sohu.com(Alexa 18), mail.ru(Alexa 50), sogou.com(Alexa 183), fedex.com, washingtonpost.com. These misconfigurations could cause privacy leakage, information theft and even account hijackings.

We further delve into these security issues and analyze the underlying causes behind them. We found that, although some are developer's mistakes, many security issues are caused by various error-prone details in the CORS protocol design and implementation. We propose some improvements and mitigation measures to address these problems.

To sum up, this paper makes the following contributions:

- We conducted a comprehensive security analysis on CORS protocol in its design, implementation, and deployment process.
- We discovered a number of new CORS security issues and demonstrated their consequences with practical attacks. For example, remotely exploiting victim's internal binary-protocol services, remotely obtaining victim's accurate cookie size on *any* website.
- We conducted a large-scale measurement of CORS configurations in popular websites, and found **27.5%** of all the CORS configured domains have insecure misconfigurations.
- We analyzed the underlying design reasons behind those security issues, and proposed protocol simplifications and clarifications to mitigate them.

We organize the rest of this paper as follows. Section 2 describes the development of cross origin network access and CORS. In Section 3 we present an overview of this study, including methodology and summary of discovered CORS issues. In the next three sections (Section 4 to 6), we detail three categories of CORS security issues separately and also demonstrate their security implications with case studies. We discuss root causes and protocol simplifications in Section 7 and related research regarding CORS and SOP security in Section 8. We conclude in Section 9.

2 Background

Cross-origin resource access can be classified into two categories: cross-origin local resources access (e.g., for DOM, cookie) and cross-origin network resources access (e.g., for XMLHttpRequest). The former has been studied in previous research [24, 38], and the latter is the focus of this paper. More specifically, we study the access control mechanisms for both sending cross-origin requests and reading cross-origin responses.

2.1 Cross-Origin Network Access

Cross-origin reference is a core feature of the web at its birth, and there is no explicit cross-origin access control mechanisms built into the HTTP protocol. In other words, any website can refer to resources of any other website using HTML tags, implying that any website can manipulate a visitor's browser into issuing GET requests to any resource servers. This does not directly cause security concerns when HTML does not support active content. Contents retrieved by HTTP requests are rendered by the browser. Websites referring the resources do not have direct access to the contents.

JavaScript changes the threat model of the Web, and introduces significant risks to the cross-origin access. In order to ensure that different web applications cannot interfere with each other, Netscape introduced the *Same Origin Policy (SOP)*, the fundamental isolation strategy for client-side web application security. This policy defines the security boundary of a resource by its *origin*, the URI scheme/host/port tuple. Although SOP prevents JavaScript from reading the response of a cross-origin request (except a few cases such as imported script), it does not prevent client-side JavaScript from sending cross-origin POST requests (e.g., using automatic form submission without user awareness). While this permissive sending capability provides rich features for Web interactions, it also introduces security problems.

2.2 The Risks of Cross-Origin Sending

Automatic submission of POST requests provides more permissions to a malicious website, enabling two types of attacks.

The first category of attacks is **Cross Site Request Forgery (CSRF)** [35]. CSRF is a serious threat to the Web, and has been an OWASP top-10 security issue since 2007 [20]. Besides the possibility of automatic POST submission, two other mechanisms in web lead to the severity of CSRF. First, POST is the standard method for non-idempotent request that changes server state. Second, cookies are commonly used in web applications as authentication tokens, attached by default with HTTP re-

quests. Combining the three factors, a malicious website can control a victim's browser to issue POST requests with the victim's identity to other websites. Without sufficient application-level defenses, this could cause disastrous consequences, such as automatic money transferring from the victim to the attacker account.

The second category is **HTML Form Protocol Attack (HFPA)** [28]. HFPA allows an adversary to use a victims' browser as a stepping-stone to attack text-based protocol services (such as SMTP) otherwise unreachable, e.g., located within an internal network. By carefully crafting HTML forms, an attacker can encapsulate other textual protocol data into the body of cross-origin POST requests. Since textual protocol implementations are often permissive in accepting input, they simply ignore the unknown lines in POST requests and execute the known commands crafted by an attacker. Below is an example showing how SMTP commands are encapsulated into a POST request:

```
POST  /  HTTP/1.1
Host: 192.168.1.1
Content-type: multipart/form-data; boundary=--123

--123
Content-Disposition: form-data; name="foo"

HELO example.com
MAIL FROM:<somebody@example.com>
RCPT TO:<recipient@example.org>
--123--
```

There are currently no effective protocol-level solution for these two types of attacks. Proposed solutions for CSRF attacks, such as Origin header [7] and same-site cookies [16], are not widely deployed due to incomplete browser support. The mainstream CSRF defense still relies on CSRF tokens, implemented by individual web applications. To mitigate HFPA attacks, browsers restrict port numbers in cross-origin requests, e.g., by disallowing cross-origin requests to port 25 to protect SMTP services. However, such blacklisting approaches are incomplete, since services may be configured to use different port numbers, and new services are constantly emerging. Thus, browsers often block only a small subset of port numbers, leaving the majority of them exposed. For example, Chrome disables 63 port numbers in total, while Edge and IE browser only forbid 8 of them. None of the browsers protect port 6379 (redis) or 11211 (memcache), for example, leaving those services vulnerable to HFPA attacks [10].

2.3 The Need for Cross-Origin Reading

Many web applications need JavaScript to have the capability to read responses of cross-origin resources. Initially developers invented JSON-P (JSON with Padding) [12] to bypass SOP, by leveraging the exception that an imported cross-origin JavaScript using the <script> tag is accessible to the hosting page. A resources server can encapsulate shared data in JSON format into JavaScript by padding, and a third-party domain can include the JavaScript through <script> tag to obtain the embedded data. Although JSON-P solves some cross-origin resource sharing problems, it still has limitations. For example, it only supports resource sharing through cross-origin GET requests and doesn't support other methods such as POST. Further, it introduces two inherent security problems [9, 21]. First, importing a third-party JSON-P resource requires complete trust of the third-party. Because JSON-P resource is executed immediately as JavaScript; the importing origin cannot perform any input validation on the content. Second, a JSON-P resource needs to have application-level access control to prevent unauthorized read, which complicate web application implementations.

In order to provide a safer and more powerful solution for authorized cross-origin resource sharing, W3C designed Cross-Origin Resource Sharing (CORS) [31] protocol to replace JSONP. Since the first proposal in 2005, CORS has had several iterations in terms of protocol design. In August 2011, CORS was included in *Fetch* standard [30] by Web Hypertext Application Technology Working Group(WHATWG) [33], another web standard organization founded by browser vendors including Mozilla, Opera and Apple. Since then, CORS was independently updated in the Fetch standard, and has minor differences from the W3C standard. Browser vendors such as Mozilla gave priority to the WHATWG's standard [4], resulting in the obsolescence of W3C CORS standard in August 2017 [5]. Today, CORS is implemented in all major browsers and is still evolving. Figure 1 summarizes the development history of cross-origin access and CORS.

2.4 The Complexity of CORS

In general, CORS consists of three steps:

1. A domain issues a cross-origin request to a resource server. For each CORS request, an Origin header is automatically added by the browser to indicate the origin of the requesting domain.

2. The resource server generates an access control policy in HTTP response headers (Access-Control-Allow-Origin) indicating the origins allowed to read its resources.

3. The browser enforces the received access control policy by checking if the requesting origin matches

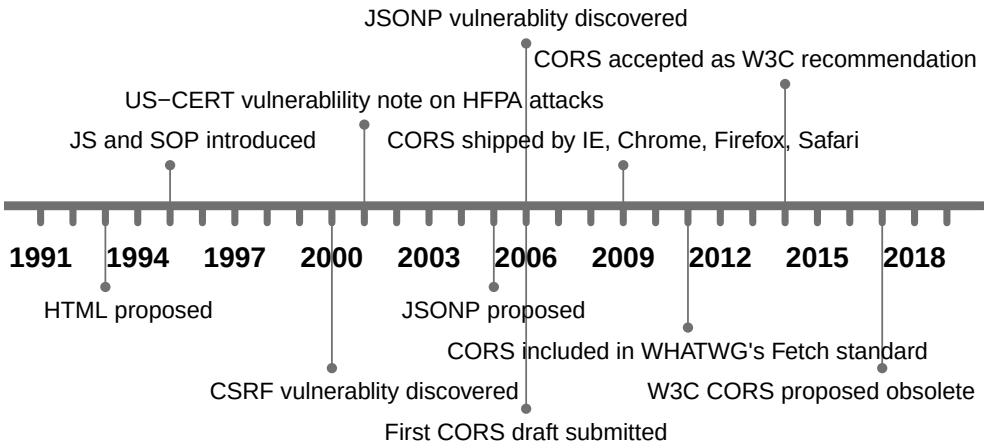

Figure 1: Timeline of cross-origin network access and CORS development.

the allowed origins as specified by Access-Control-Allow-Origin header. Only if yes is the requesting domain allowed to read the response content.

CORS may seem straightforward, but its details are complex. In addition to the access control for *origins*, CORS also provides fine-grained access control for HTTP *methods*, HTTP *headers*, and *credentials* (including cookies, TLS client certificates, and proxy authentication information). Partly for backward compatibility, CORS classifies cross-origin requests into two categories based on request methods and headers, *simple requests* and *non-simple requests*. A simple request must satisfy all of the following three conditions. Otherwise, a request is considered non-simple.

a) Request method is HEAD, GET or POST.

b) Request header values are not customized, except for 9 whitelisted headers: Accept, Accept-Language, Content-Language, Content-Type, DPR, Downlink, Save-Data, Viewport-Width, and Width.

c) Content-Type header value is one of three specific values: "text/plain", "multipart/form-data", and "application/x-form-uri-encoded".

A simple cross-origin request is considered safe and will be sent out directly by the browser. A non-simple request is considered dangerous, thus requires a *preflight* request to obtain permission from the resource owner to send the actual cross-origin request. The preflight request is initiated with an OPTIONS method, and includes Origin, Access-Control-Request-Method, Access-Control-Request-Headers headers. The resource server includes Access-Control-Allow-Origin, Access-Control-Allow-Method and Access-Control-Allow-Headers in its HTTP response to indicate the allowed origins, methods, and headers respectively. The browser then checks whether the policy in the response headers allow for sending the actual cross-origin request.

To reduce the performance impact due to preflight requests, CORS provides the Access-Control-Max-Age response header to allow a browser to cache the results of preflight requests. Further, additional features are also defined, e.g., Access-Control-Allow-Credentials controls whether or not a cross-origin request should include credentials such as cookies.

3 Overview of CORS Security Analysis

Essentially, the CORS protocol is an access control model regulating access to cross-origin network resources (including sending requests and reading responses) between browsers and servers. In this model, a requesting website script initiates a resource access request from a user's browser, which automatically adds an Origin header to indicate the requester's identity; then the third-party website returns the access control policy; Finally, the browser enforces the access control policy to determine whether the requester can access the requested network resources. This section presents an overview of our study.

3.1 Threat Model

We consider two types of attackers: web attackers and active network attackers. Web attackers only need to trick a victim into clicking a link to execute malicious JavaScript in the victim's browser, while active network attackers need to manipulate the victim's network traffic. Unless otherwise specified, attacks in this paper can be launched by web attackers.

3.2 Methodology

We studied specifications including W3C's CORS standard [31], WHATWG's Fetch standard [30], and CORS-related discussions in W3C mailing lists [15] to learn how CORS is designed and its security considerations. We also examined CORS implementations including 5 major browsers and 11 popular open-source web frameworks to understand how CORS features are implemented in practice. In the course of doing so, we identified potential interactions between CORS features and known attacks (specific and general) and their implications.

Furthermore, we measure CORS policies of real-world websites to evaluate CORS deployment in the wild. We conducted a large scale measurement on Alexa Top 50,000 websites, including their 97,199,966 distinct sub-domains. For each domain, we sent cross-origin requests with different requesting identities to examine their CORS policies in response headers.

3.3 Summary of Analysis Results

Through the analysis, we found a number of CORS-related security issues, which we can classify into three high-level categories, per Table 1.

1) Incomplete reference monitor. CORS allows "simple" requests to be sent freely by default, to keep consistent with previous policy (cross-origin GET and POST requests are allowed by default). Yet, the scope of simple CORS requests is in fact beyond previous capabilities in a number of subtle ways. It turns out that the new by-default sending capability of CORS can be exploited by web attackers to launch a variety of attacks that are previously not able to carry out in a web attacker setting.

2) Trust dependency. A domain with strong security mechanisms may allow CORS access from a weaker domain. A web/network attacker can compromise a weak domain and issue CORS requests to obtain sensitive information from the strong security domain.

3) Policy complexity. Because the CORS itself policy cannot be expressed in the simple form, many websites implement error-prone dynamic CORS policy generation at the application level. We found that a variety of misconfigurations of CORS policies are due to these complex policies.

In the following three sections, we will describe these three categories of problems in detail.

4 Overly Permissive Sending Permission

The cross-origin sending permission of default SOP already poses significant security challenges, leading to vulnerabilities such as CSRF and HFPA attacks (Section 2.2). Absent consideration of backward compatibility, CORS could have addressed all cross-origin access to solve and unify the defenses against CSRF, HFPA, and other cross-origin network resource access at the protocol level. But instead CORS kept compatibility with the previous policy.

CORS allows "simple" requests to be sent freely by default in its new JavaScript interfaces (e.g., XMLHttpRequest Level2, fetch). However, these new interfaces (referred to as "CORS interfaces" subsequently) in fact implicitly further relax sending permissions, unintentionally allowing malicious customization of HTTP headers and bodies in CORS simple requests.

4.1 Crafting Request Headers

Before the advent of CORS, cross-origin requests could only be sent using header fields and values fixed by the browser. CORS interfaces provide new capabilities that allow JavaScript to modify 9 CORS whitelisted headers (See Section 2.4). Further, CORS imposes few limitations on the values and sizes of these headers. Thus, an attacker can craft these headers with malicious content to deliver attack payloads.

CORS imposes few limitations on header values. RFC 7231 [22] provides clear BNF format requirements for 4 out of 9 CORS whitelisted headers: Accept, Accept-Language, Content-Language and Content-Type. For example, standard-compliant Accept header values should be like "text/html,application/xml". CORS imposes no format restrictions on any whitelisted headers, except Content-Type. CORS works on the top of HTTP, so when implementing CORS interfaces, browsers should restrict at least those 4 whitelisted header values according to HTTP's BNF rules. However, in our testing of five mainstream browsers (Chrome, Edge, Firefox, IE, Safari), all except Safari lack any restrictions on any headers other than Content-Type. For example, their values can be set to "(){:;};", an attack payload for exploiting the Shellshock vulnerability [17]. Safari restricts the values of Accept, Accept-Language and Content-Language, disallowing some delimiter characters like "(","{".

In addition, although the five browsers follow CORS standards in limiting Content-Type to three specific values ("text/plain", "multipart/form-data","application/x-form-url-encoded"), these restrictions can be bypassed. We found that all of them prefix-match the three values and ignore the remaining values beyond the first comma or semicolon. Thus, an attacker can still craft malicious content in Content-Type headers by appending an attack payload to a valid value.

These implementation flaws open new attack surface

Table 1: Overview of CORS security problems

Categories	Problems	Attacks
Overly permissive sending permission	Overly permissive header formats and values	RCE via crafting headers
	Few limitations on header size	Infer privacy information for any website
	Overly flexible body format	File upload CSRF
	Few limitations on body value	Attack binary protocol services
Risky trust dependency	HTTPS domain trust their own HTTP domain	MITM attacks on HTTPS websites
	Trust in other domains	Information theft or account hijacking
Policy complexity	Poor expressiveness of access control policies	Information theft or account hijacking
	Forgeable "*null*" Origin values	Information theft or account hijacking
	Security mechanism complexity	Information theft or account hijacking
	Complex interactions with caching	Cache poisoning

in that a web attacker can manipulate a victim's browser to craft exploitation payloads using a CORS "simple" request, using the browser as stepping-stone to compromise vulnerable yet nominally internal-only services.

Case study: In order to demonstrate the threat, we conducted an experiment to exploit an internal service by crafting a malicious Content-Type header. We set up a Apache Struts environment in our local network, one with the s2-045 vulnerability (CVE-2017-5638) [18]. This vulnerability was caused by incorrect parsing of Content-Type header, and led to remote code execution. As the vulnerable service was deployed in our internal network, it is supposed to be unexploitable by web attackers from an external network. However, with the help of CORS, we confirmed that an attacker can set up a web page that sends cross-origin requests with crafted malicious payload via a Content-Type header. Once an intranet victim visits this page, the vulnerability is triggered. In our experiment, this attack enabled us to obtain a shell on the internal server.

CORS imposes few limitations on header sizes. There is no explicit limit on request header sizes in either the HTTP or CORS standards. We tested five major browsers and found all of them allow for at least 16MB of one or more headers in CORS interfaces. When we set headers to very large values (e.g., 1 GB), the browsers produced "not enough memory" errors, rather than "header size too large" errors. This is much larger than request size limit enforced by other web components (e.g., web servers). Table 2 summarizes different header size limitations for five major browsers and popular web servers in default configurations.

Case Study: web attackers can exploit header size differences between browsers and web servers to launch side-channel attacks, remotely determining the presence of a victim's cookies on *any* website. To carry out this attack, an attacker first measures the header size limit of a target web server by directly issuing requests

Table 2: Header size limitations for browsers and servers (single/all headers)

Browser	Limitation	Server	Limitation
Chrome	>16MB/>16MB	Apache	8KB/<96KB
Edge	>16MB/>16MB	IIS	16KB/16KB
Firefox	>16MB/>16MB	Nginx	8KB/<30KB
IE	>16MB/>16MB	Tomcat	8KB/8KB
Safari	>16MB/>16MB	Squid	64KB/64KB

with increasing-size headers until receiving a `400 Bad Request` response. Then the attacker sends "simple" request in the victim's browser with crafted header values so that the header size is slightly smaller than the measured limit. If a cookie is present, the cookie will be automatically attached in the request. The total header size will exceed the limitation, resulting a `400 Bad Request` response. In the absence of cookies, the target server will return a `200 OK` response.

In fact, the attacker cannot directly observe whether a response is *200* or *400* because browsers have normalized such low-level information for security considerations. However, the attacker can utilize timing side-channels to differentiate the response status. One general timing channel is response time. If the attacker issues the "simple" request towards a large file or a time-consuming URL, a *200* response will be significantly slower than a *400* response. In Chrome, the *Performance.getEntries()* API directly exposes whether or not a request is successful: if a response has status code 400, the API will return empty response time.

Attackers can further infer more details about victim's cookies, such as the size of cookies with specific path attribute by comparing cookie size under different directories, or the size of cookies with the *secure* flag by comparing the cookie size in HTTP and HTTPS requests. As web applications usually use different amounts and

attributes of cookie to keep different states for clients, cookie size information in different dimensions can potentially indicate a victim's detailed status on target website, such as whether the user has visited, logged-in, or is administrator on the target website.

The presence of a cookie can leak private information about the victim. For example, an attacker might remotely infer the victim's health conditions by looking for visits to particular disease or hospital websites; infer political preferences by visits to candidate websites; or infer financial considerations by whether the victim has an account on lending or investment websites.

4.2 Crafting Request Bodies

Before CORS, JavaScript could only send cross-origin POST requests via automatic form submission. The browser will automatically encode the body of a request before sending, limiting the format and value of POST body data. CORS allows JavaScript to issue cross-origin "simple" requests with neither format nor value limitations on request bodies, allowing attackers to craft binary data in any format.

CORS lacks limits on body format. Standard HTML forms restrict the format of POST data. HTML form data is automatically encoded by browsers in three encoding types: "application/x-www-form-urlencoded", "text/plain", or "multipart/form-data". For the first type, the browser separates the form data with "=" and joins it with "&", such as "name1 = value1&name2 = value2"; for the second, the browser splits the form data with "=" and joins it with CRLF; for the third, the browser divides each instance of form data into different sections, each separated by a boundary string and a Content-Disposition header like *Content-Disposition: form-data; name = "title"; filename = "myfile"*.

CORS does not impose any format restrictions on request bodies. We tested five browsers and found that all of them allow JavaScript to send cross-origin requests with body data in any format. Such flexibility in composing request body can lead to new security problems.

Case Study: We show that an attacker can exploit a *file upload CSRF* vulnerability which was previously unexploitable. In an HTML form, the "filename" attribute of file select control cannot be controlled by JavaScript, and is automatically set by browsers only if the user makes a selection in the file dialog. Before CORS, checking the presence of "filename" attribute on server-side is sufficient to prevent file upload CSRF. However, CORS breaks this defense, allowing attackers to craft the body to set "filename" attribute therefore able to launch file upload CSRF attacks. We found such a case in the personal account pages of JD.com (Alexa Rank 20), which has CSRF defenses in every input place except for uploading a file to change the user's avatar. This vulnerability is unexploitable without CORS. We confirmed that, with CORS, an attacker can exploit this CSRF vulnerability to modify the victim's avatar.

CORS has few limitations on body values. Before CORS, browsers restrict binary data in the body of cross-origin POST requests by filtering or converting some special values. For example, in Firefox, Edge and IE, form data is truncated by "\x00" and the data after "\x00" will not be sent. In Chrome and Safari, a "\x0a\x0d" sequence is converted to a single character "\x0d". This limits an attacker's ability to accurately construct malicious binary data. However, both CORS standards and CORS interfaces in browsers impose no limitations on the values of request body, which gives attacker greater flexibility.

Case Study: We found that it is possible with the new flexibility to exploit *binary-based protocol* services. Apple Filing Protocol (AFP) [34] is a file-sharing protocol from Apple that provides file sharing services for MacOS. It is a binary-based protocol with its own data frames and formats. We tested the MacOS built-in AFP server and found that it always parses data using 16-byte alignment, ignoring any unrecognized 16-byte frames and continuing to parse the next 16-byte frame. Before CORS, this protocol is not vulnerable to HFPA attacks due to the format and value limitations of HTML form. By taking advantage of the CORS interfaces, an attacker can craft a cross-origin request, making its header size a multiple of 16 bytes, which is ignored by the AFP server, and constructing its binary body in AFP protocol format for communication with the AFP Server. We demonstrated this attack in our experiments: by sending a cross-origin request from a public website, we can create new files on an AFP server located in our otherwise-protected intranet.

5 Risky Trust Dependency

CORS provides web developers an authorization channel to relax the browser's SOP and share contents with other trusted domains. However, this trust relationship makes the target site dependent on the security of third-party websites, increasing attack surfaces. An attacker can first enter a weakly secured trusted domain, and then abuse this trust relationship to attack a strongly secured target site.

We study two typical types of trust relationship and the risks they pose:1) HTTPS site trusting their own HTTP domain. 2) Trusting other domains. In the first case, an active network attacker can read sensitive information and launch CSRF attacks against HTTPS websites by hijacking HTTP website contents. In the second case, a web attacker can carry out similar attacks on

a strongly secured website by exploiting XSS vulnerabilities on a weak website. Furthermore, our measurements on popular websites showed that those two risks were largely overlooked by developers. We found that about 12.7% CORS-configured HTTPS websites (e.g., fedex.com) trust their own HTTP domain, and 17.5% CORS-configured websites (e.g., mail.ru) trusted all of its subdomains.

5.1 HTTPS Site Trust HTTP Domain

HTTPS is designed to secure communication over insecure networks. Therefore, a man-in-the-middle attacker cannot read the content of an HTTPS website. However, if an HTTPS site is configured with CORS and trusts its own HTTP domain, then an MITM attacker can first hijack the trusted HTTP domain, and then send a cross-origin request from this domain to the HTTPS site, and indirectly read the protected content under the HTTPS domain.

Case Study: Fedex.com (Alexa Rank 470), has fully deployed HTTPS and enabled the *secure* and *httponly* flag in its cookies to protect against MITM attacks. But it configures CORS and trusts its HTTP domain, so an MITM attacker can first hijack the HTTP domain and then send cross-origin requests to read the HTTPS content. We verified this attack in our experiments: it allowed attackers to read detailed user account information, such as user names, email addresses, home addresses, credit cards on Fedex.com.

5.2 Trusting Other Domains

Other domains can be divided into two types, their own subdomains and third-party domains.

Trusting all of its own subdomains. The harm of cross-site scripting (XSS) vulnerability [36] on a subdomain is often limited, because it cannot read sensitive contents on other important subdomains directly due to SOP restrictions, nor steal cookies that use the *httponly* flag. But if an important subdomain is configured with CORS and trusts other subdomains, the harm of a subdomain XSS can be enhanced.

Case study: Russia's leading mail service mail.ru (Alexa global rank 50) provides strong security protection for the primary domain (https://mail.ru), such as deploying CSP (Content Security Policy) [27] to prevent XSS, and enabling *httponly* flag in its cookies. But its primary domain is configured to trust any subdomain, and mail.ru subdomains are less secured, so an attacker can exploit any XSS vulnerability present on its subdomains to read the contents of the primary domain.

We verified this attack as follows. We found an XSS vulnerability on its subdomain, https://lipidium.lady.mail.ru. By exploiting[1] this XSS vulnerability, we could successfully read sensitive content of the top domain, including the user name, email address, and the number of unread mails information.

Trusting third-party domains. If a secure site is configured with CORS and trusts a third-party domain, an attacker could exploit the vulnerability on the third-party domain to indirectly attack the secure site.

Case study: The Korean e-commerce site (faceware.cafe24.com) and the Chinese house decoration website (www.jiazhuang.com) trust third-party websites crossdomain.com and runapi.showdoc.cc respectively, but the third-party websites have security issues. crossdomain.com's domain name has expired and can be registered by anyone, and runapi.showdoc.cc has an XSS vulnerability on its site. So an attacker could exploit these vulnerabilities on third-party sites to indirectly attack the target sites.

5.3 CORS Measurement

To understand the real-world impact of the aforementioned problems, we conducted measurements of CORS deployments on popular sites. We targeted the Alexa Top 50,000 domains and extracted all of their subdomains from an open-to-researchers passive DNS database [1] operated by a large security company [2]. In total, we collected 97,199,966 different subdomains over 49,729 different SLDs.

For each subdomain, we repeatedly changed the Origin header value to different error-prone values in different testing requests, and inferred their CORS configurations according to response headers. For example, to understand whether an HTTPS domain (e.g., https://example.com) trusts its HTTP domain, we set the request Origin header to be "Origin: http://example.com". If the response headers from the HTTPS domain contains "Access-Control-Allow-Origin: http://example.com", we know that the HTTPS domain trusts its HTTP domain. We use the same approach in other subsections.

We found that 481,589 domains were configured with CORS, of which 61,347 HTTPS domains (about 12.7%) trusted the HTTP domain and 84,327 domains (about 17.5%) trusted any of its own subdomains, as shown in Table 3.

We further investigate the reasons behind the high proportion of these two security risks. By analyzing CORS standards, web frameworks, and web software, we found three reasons for the first risk: 1) The standards don't explicitly emphasize the security risk. 2) Some web frame-

[1]Note, this exploitation was wholly contained to manipulating our own browsers; no third party was manipulated via XSS.

Table 3: Measurement of insecure CORS configurations

Categories	Count	Percentage	Examples
HTTPS trust HTTP	61,347	12.7%	fedex.com, global.alipay.com, www.yandex.ru
Trust all subdomains	84,327	17.5%	mail.ru, mobile.facebook.com, payment.baidu.com
Reflecting origin	15,902	3.3%	account.sogou.com, analytics.microsoft.com, account.nasdaq.com
Prefix match	1,876	0.4%	tv.sohu.com, myaccount.realtor.com, manage.renren.com
Suffix match	32,575	6.8%	m.hulu.com, www.php.net, account.zhihu.com
Substring match	430	0.1%	subscribe.washingtonpost.com, hrc.byu.edu
Not escaping "."	890	0.2%	www.nlm.nih.gov, about.bankofamerica.com
Trust *null*	3,991	0.8%	mingxing.qq.com, aboutyou.de, login.thesun.co.uk
Total	132,476	27.5%	

works fail to check protocol types. For example, the popular web framework django-cors-headers only checks the domain and neglects the protocol type when examining a request's Origin header in order to return the CORS policy. 3) Some web applications allow both http and https protocol types for better compatibility. We analyzed the popular CMS software Wordpress and found that its trust list was hard-coded to allow both HTTP and HTTPS domains when returning CORS policies. This approach improves compatibility and can make Wordpress run in both HTTP and HTTPS environment without any extra configuration, but it introduces new security risks.

We also do not find any explicit security warnings for the second risk (trusting third-party domains) in either of the standards (W3C or Fetch). Another reason for the second risk is that trusting arbitrary third-party subdomains simplifies web developer configuration, especially when a resource needs to be shared among multiple different subdomains.

6 Complex Policies and Misconfigurations

The core function of CORS is that the policies generated by resource servers instruct client browsers to relax SOP restrictions and share cross-origin resources. If the server-side policies are incorrect, it may trust an unintended domain, bypassing the browser's SOP enforcement. To understand this risk, we analyzed open-source web framework implementations and real-world CORS deployments. We discovered a number of CORS misconfiguration issues. We found that 10.4% of CORS-configured domains trust attacker-controllable sites. We also found that 7 out of 11 popular CORS frameworks undermine CORS's security mechanisms and could generate insecure policies.

While some mistakes were caused by negligence, others arose due to the complex details and pitfalls in CORS' design and implementation, which make CORS unfriendly to developers and prone to misconfigurations. We can classify the reasons into four categories: 1) The expressiveness of access control policy is poor. Many websites need to implement error-prone dynamic CORS policy generation at the application-level. 2) Origin *null* value could be forged in some corner cases. 3) Developers do not fully understand the CORS security mechanisms, leading to misconfigurations. 4) Interactions between CORS and web caching bring new complexity.

6.1 Poor Expressiveness of CORS Policy

The W3C CORS standard states that an Access-Control-Allow-Origin header value can be either *an origin list*, "null", or "*", whereas in the WHATWG's Fetch standard, it can only be *a single origin*, "null", or "*". Our test on five major browsers shows that they all comply with the WHATWG's Fetch standard.

This access control policy is not expressive enough to meet common web developer usage patterns. For example, it is difficult for web developers to share resources across multiple domain names through simple server configurations. Instead, they need to write specific code or use the web framework to dynamically generate different CORS policies for requests from different origins. This approach increases the difficulty of CORS configuration, and is error-prone in practice. We found a number of misconfigurations are rooted in this category.

In general, we can classify the misconfigurations into two sub-categories: 1) blindly reflect requester's origin in response headers; 2) attempt to validate requester's origin but make mistakes.

1). Reflecting origin. When web developers have to dynamically generate polices, the simplest way to configure CORS is to blindly reflect the Origin header value in Access-Control-Allow-Origin headers in responses. This configuration is simple, but dangerous, as it is equivalent to trusting any website, and opens doors for attacker websites to read authenticated resources. In

our measurement, 15,902 websites (about 3.3%) out of 481,589 CORS-configured websites have this permissive configuration, including a number of popular websites such as account.sogou.com, analytics.microsoft.com, account.nasdaq.com.

2). Validation mistakes. Due to the poor expressiveness of CORS policies, web developers have to dynamically validate the request Origin header and generate corresponding CORS policies. We find the validation processes prone to errors, resulting in trusting unexpected attacker-controllable websites. These errors can be classified into four types. **i) Prefix matching**: When a resource server checks whether the Origin header value matches a trusted domain, it trusts any domain prefixed with the trusted domain. For example, a resource server wants to trust example.com, but forgets the ending character, resulting in allowing example.com.attacker.com. We found this mistake on popular websites like tv.sohu.com, myaccount.realtor.com. **ii) Suffix matching**: When a resource server checks whether the Origin header value matches any subdomain of a trusted domain, the suffix matching is incomplete, accepting any domain ending with the trusted domain. For example, www.example.com wants to allow any example.com subdomain, but it only checks whether the Origin header value ends with "example.com", leading to allow attackexample.com, which can registered by attackers. Such mistakes are found on websites like m.hulu.com. **iii) Not escaping '.'**: For example, example.com wants to allow www.example.com using regular expression matching, but its configuration omits escaping ".", resulting in allowing wwwaexmaple.com. Websites like www.nlm.nih.gov are found to make this mistake. **iv) Substring matching**: We also found that some websites like subscribe.washingtonpost.com have validation mistakes, resulting allowing ashingtonpost.co, which can be registered by anyone. In our measurement, a total of 50,216 domain names (about 10.4%) were found to have these validation mistakes, as shown in Table 3.

6.2 Origin Forgery

An important security prerequisite for CORS is that the Origin header value in a cross-origin request cannot be forged. But this assumption does not always hold in reality.

The Origin header was first proposed for defense against CSRF attacks [7]. RFC 6454 [6] states that if a request comes from a privacy-sensitive context, the Origin header value should be *null*, but it does not explicitly define what is a privacy-sensitive context.

CORS reuses the Origin header, but CORS standards also lack clear definition of *null* value. In browser implementations, *null* is sent from multiple different sources, including local file pages, iframe sandbox scripts. When developers want to share data with local file pages (e.g., hybrid applications), they configure "Access-Control-Allow-Origin: null" and "Access-Control-Allow-Credentials: true" on their websites. However, an attacker can also forge the Origin header with *null* value from any website by using browser's iframe sandbox feature. Thus, sites configured with "Access-Control-Allow-Origin: null" and "Access-Control-Allow-Credentials: true" can be read by any domain in this way. In our measurement, we found 3,991 domains (about 0.8%) with this misconfiguration, including mingxing.qq.com, aboutyou.de.

Table 4: Different CORS framework implementations

Framework	* and "true" to reflection	no Vary
ASP.net CORS (ASP.net)	Yes	
Corsslim (PHP)		Yes
Django-cors-headers (Python)	Yes	
Flask-cors (Python)	Yes	
Go-cors (Golang)	Yes	
Laravel-cors (PHP)	Yes	
NelmioCorsBundle (PHP)		Yes
Plack::Middleware ::CrossOrigin (Perl)	Yes	Yes
Rack-cors (Ruby)		
Tomcat CORS filter (Java)	Yes	
Yii2 CORS filter (PHP)		Yes

6.3 Complexity of Security Mechanisms

For web developers' convenience, CORS allows Access-Control-Allow-Origin to be configured with the wildcard "*", which allows any domain. Given these overly-loose permissions, CORS later added an additional security mechanism: "Access-Control-Allow-Origin: *" and "Access-Control-Allow-Credentials: true" cannot be used at the same time. This means that "Access-Control-Allow-Origin: *" can only be used to share public resources.

We found this security mechanism is not well-understood by either application developers or framework developers: 1) Many application developers were not aware of this additional requirement and still configured both "Access-Control-Allow-Origin: *" and "Access-Control-Allow-Credentials: true". In our measurement, 7,444 out of 481,589 CORS-configured domains (about 1.5%) manifested this mistake, including

popular domain names such as api.vimeo.com, security.harvard.edu. 2) To avoid the above configuration errors, some web frameworks actively convert the combination into reflecting origin. This causes the protocol security mechanism to be bypassed, allowing any domain to read authenticated resources. We analyzed 11 popular CORS middleware and found that 7 of them converted this combination to reflecting origin, as shown in Table 4.

In addition, this mechanism also increased browser complexity. We tested five major browsers and found that they have implementation pitfalls on this issue. Browsers are supposed to be always return error when a server replies "Access-Control-Allow-Origin: *" and "Access-Control-Allow-Credentials: true" for a credential-included cross-origin request. However, we found that this configuration combination can pass the browser's security check when they are in the response for preflight requests. Thus, if a web site misconfigured "Access-Control-Allow-Origin: *", "Access-Control-Allow-Credentials: true" and "Access-Control-Allow-Method: DELETE", an attacker can still pass the preflight check and send a cross-origin DELETE request with credentials for that site.

6.4 CORS and Cache

There is another error-prone corner case when CORS interacts with an HTTP cache. When a resource server needs to be shared with multiple domain names, it needs to generate different CORS policies for different requesting domains. But most web proxies cache HTTP contents only based on URLs, without taking into consideration the associated CORS policies. If a resource shared with multiple domains is cached with CORS policy for one domain, others domains will not be able to access the resource because of CORS policy violation. For example, a resource from c.com needs to be shared with both a.com and b.com from browsers sharing a same cache. If the resource is first accessed by a.com and is cached with header "Access-Control-Allow-Origin: a.com", b.com will not be able to access the resource since the cached content has a CORS policy that does not match with b.com.

HTTP provides the Vary header for this situation. A resource server needs to configure "Vary: Origin" in its response headers, which instructs web caches to cache HTTP contents based on both URLs and Origin header value. Thus, when a server returns different CORS policies for different requesting domains, these resources will be cached in different entries.

Many developers are not aware of this corner case. In our measurements, 132,987 domains (about 27%) allowed for multiple different domains, but didn't configure "Vary: Origin", such as azure.microsoft.com and global.alipay.com. We analyzed 11 samples of CORS middleware, finding 4 that were not aware of this issue and did not generate Vary headers, as shown in Table 4.

6.5 Responsible Disclosure

We are in the process of reporting all vulnerabilities to the affected vendors. Some websites (e.g., sohu.com, mail.ru) have acknowledged and fixed the issues.

7 Discussion

We first analyze the underlying causes behind the CORS security issues and then propose corresponding mitigation and improvement measures.

7.1 Root Cause

Backward compatibility needs to be just right. Although backward compatibility is important in designing new systems, over consideration can deteriorate system security and increase burden in system development and deployment. Prior to CORS, cross origin request attacks have become serious problems for web security. To keep backward compatibility, CORS can choose not to solve the existing form submission problem, but it is not necessary to allow default sending permission in its newly opened interfaces. Although CORS made attempt to restrict the default sending permission such as restricting Content-Type to three white-list values, it unintentionally relaxed the permissions in subtle ways, leading to various new cross-origin attacks.

Under web rapid iterative development model, new protocols aren't fully evaluated before deployed. New features are quickly implemented by browsers and shipped to users before they are fully evaluated, some immature design are difficult to change after these features are widely used in Web. Starting in the second half of 2008, CORS protocol has major changes and is still under discussion in the W3C. Due to web developers' requirements or browsers' competitions, in January 2009, some vendors have implemented this immature protocol into browsers as new features, which include some immature design, such as CORS policies only support a single origin [8]. Although the new CORS standard in 2010 required Access-Control-Allow-Origin to support origin list [29], these requirements haven't been supported in any browsers. One reason is compatibility issues. Browser modification could lead to different versions of browsers supporting different levels of access control policies, CORS configuration will be further complicated. Another reason is that, currently web developers can dynamically generate CORS configuration

to complete their goals. Therefore, this design kept unchanged, which increased web developers configuration difficulty.

The protocol security considerations haven't been effectively conveyed to the developers. The CORS protocol has many error-prone corner cases in its design and implementation, as presented in Section 5 and Section 6, but these cases are not effectively conveyed to developers. An important reason is that these security risks aren't clearly highlighted in the two CORS specifications. First, the W3C CORS standard lacked timely updates, its latest version was still in 2014 [31]. In August 2017, the W3C CORS standard was proposed for obsolescence in the W3C mailing list [5], suggesting the use of WHATWG's Fetch standard. Web developers who didn't subscribe to the W3C mailing list would likely still take W3C CORS standard to be the latest standard. Second, WHATWG's Fetch standard had no separate security consideration section and did not emphasize these security risks either.

7.2 Improvement for CORS

We found the CORS protocol can be improved in four aspects:

The default sending permission should be more restrictive. A fundamental cause for cross origin request attack is that a browser allows to directly send cross origin requests, which could contain malicious data, without asking permission from the server.

One solution is to send a preflight request for all cross origin requests that allow users to modify headers and body, and then send the real request after negotiating with the server. To reduce the additional preflight round trip, developers can use Access-Control-Max-Age to cache preflight requests. Note the "always-preflight" solution may break websites that deploy CORS yet not support preflight, e.g., those supporting only CORS simple request. While not many, those sites do exist based on our experiments.

Another mitigation is to limit the format and value of white-list headers and bodies in CORS simple requests, similar to restricting Content-Type header to take only three specific values. However, this approach also increases the complexity of CORS protocol and may bring unexpected security troubles. For example, originally, CORS limited Content-Type to three specific values excluding "application/json", so many web applications used this restriction as CSRF defenses against JSON APIs. Later, Chrome opened new API SendBeacons() for new features, which can send "Content-Type: application/json" in cross origin requests directly [32]. This behavior break many websites' CSRF defense and brought controversy [3].

CORS configuration should be simplified. The poor expressiveness of CORS policy increase the configuration complexity, web developers have to dynamically generate corresponding CORS policies, which are prone to mistakes. Therefore, browsers should support advanced CORS policies, such as origin list, subdomain wildcard, to simplify developers' CORS configuration in common usages.

The *null* definition should be clear. In CORS standards, the *null* value definition is not clear, and in actual practice, browsers send *null* values in different sources. Developers who don't know this corner cases may misconfigure CORS. Therefore, the CORS standard needs to clearly define *null* values, preferably using different values for different sources.

Security risks should be clearly summarized in standards. The standard should explicitly point out the risk of trust dependencies brought by CORS. Also, many CORS misconfigurations are caused by various subtle corner cases. These security risks should be clearly delivered to developers, for example, summarizing best practices for CORS configuration, highlighting various CORS error-prone details, and updating them in the latest CORS standards.

8 Related Work

CORS is a relatively new web security mechanism. Although a few researchers have found some CORS security issues [37, 23, 11, 14, 13], there is no systematic study and assessment for CORS security. Our work aims to fill in this gap, providing a comprehensive security analysis of CORS in design, implementation and deployment process. Through this analysis, we discovered a number of previously unknown security issues. We believe our study can help the community have a deeper understanding of CORS security and know the status quo better.

8.1 Cross-Origin Sending Problems

A few researchers noticed some cases about CORS-related security issues [37, 23], but they only briefly touched individual cases rather than studied CORS systematically. Wilander opened an issue on Github [37], suggesting that Fetch standard should restrict Accept, Accept-Language, and Content-Language value according to RFC 7231, as an attacker may abuse these three headers to delivery malicious payloads. We found that even though Safari adopted his advice to limit the three headers from using some insecure values, this problem was still not completely solved. Revay found POST body format was relaxed in XMLHttpRequest API, which

could lead to file upload CSRF [23], and we further provided a real world case to demonstrate this threat. In summary, a few researchers found some new security issues brought by CORS, but none provide an overall study on this problem. We systematically studied the new permissions introduced by CORS and their security risks, and further demonstrated their harmful consequences with practical attacks.

8.2 CORS Misconfiguration Problems

We were aware of some known CORS misconfigurations attacks and studies [11, 14, 13, 19]. Gurt found a CORS configuration mistake in the one of Facebook Message domain, resulting in any malicious web sites can read victim's chat information [11]. Kettle discovered and summarized various CORS misconfigurations he encountered in his penetration testing experience [14]. Inspired by his work, and we studied and measured CORS misconfiguration semantically, and further analyzed their root causes. Johnson measured the reflecting origin misconfiguration in the Alexa top 1M sites [13], and Mller [19] measured different misconfigurations mentioned in Kettle's work. With the help of passive DNS database, we further performed an in-depth evaluation on their different subdomains. We also analyzed different CORS frameworks to understand those misconfigurations.

8.3 Other Cross-Origin Problems

From a broad perspective, our work can also be viewed as an analysis of access control policies in the Web. Singh studied inconsistent access control policies for different resources in web browsers, but not including CORS [25]. Schwenk tested the SOP for DOM between different browsers and found many inconsistencies [24]. Zheng studied the SOP for cookies and found that various cookie-related security issues [38]. Son studied the usage of PostMessage, a client-side cross-origin communication mechanism, on the Alexa top 10,000 websites and found many vulnerable websites [26].

9 Conclusion

We conducted an empirical security study on CORS. We examined CORS specifications and implementations in both browsers and Web frameworks, and discovered a number of new security issues. By conducting an large scale measurement on CORS deployment in real-world websites, we found that CORS was not well-understood by developers, 27.5% of all the CORS configured domains had insecure misconfigurations. We further analyzed the underlying reasons behind these issues and found that while some are developer's negligence, many security issues are rooted in the CORS protocol design and implementations. Finally, we proposed some improvements and clarifications to address these problems.

The reality of CORS security is an unfortunate epitome of web security. As the Web keeps adding new, in many cases, premature features, unexpected interactions cause new security threats. Mitigation of new threats further require new features, which if not designed properly will again introduce new risks. Backward compatibility further complicate the problem. We hope that web community can take more principled approach to security in future web protocol design and implementation.

References

[1] 360, Q. Network security research lab at 360. `http://netlab.360.com/`, 2017. [accessed Feb-2018].

[2] 360, Q. Qihoo 360 technology co. ltd. `http://www.360.cn/`, 2017. [accessed Feb-2018].

[3] AYREY, D. Json api's are automatically protected against csrf, and google almost took it away. `https://github.com/dxa4481/CORS`, 2017. [accessed Feb-2018].

[4] BARON, D. W3c proposed recommendation: Html5. `https://groups.google.com/forum/#!msg/mozilla.dev.platform/BnY1261cNJo/MdkaT_EX6MOJ`, 2014. [accessed Feb-2018].

[5] BARON, D. Transition request: Proposed obsolete for cors. `https://lists.w3.org/Archives/Public/public-webappsec/2017Aug/0010.html`, 2017. [accessed Feb-2018].

[6] BARTH, A. Rfc 7231-the web origin concept. december 2011. *URl: https://tools.ietf.org/html/rfc6454* (2011).

[7] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 75–88.

[8] BATEMAN, A. Access-control-allow-origin: * and ascii-origin in ie8. `https://lists.w3.org/Archives/Public/public-webapps/2009JanMar/0090.html`, 2009. [accessed Feb-2018].

[9] GROSSMAN, J. Advanced web attack techniques using gmail. `http://blog.jeremiahgrossman.com/2006/01/advanced-web-attack-techniques-using.html`, 2006. [accessed Feb-2018].

[10] GRGOIRE, N. Trying to hack redis via http requests. `http://www.agarri.fr/kom/archives/2014/09/11/trying_to_hack_redis_via_http_requests/index.html`, 2014. [accessed Feb-2018].

[11] GURT, Y. Critical issue opened private chats of facebook messenger users up to attackers. `https://www.bugsec.com/news/facebook-originull/`, 2013. [accessed Feb-2018].

[12] IPPOLITO, B. Remote json - jsonp. `http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/`, 2005. [accessed Feb-2018].

[13] JOHNSON, E. Misconfigured cors, stealing user data from the alexa 1m. `https://ejj.io/misconfigured-cors/`, 2016. [accessed Feb-2018].

[14] KETTLE, J. Exploiting cors misconfigurations for bitcoins and bounties. http://blog.portswigger.net/2016/10/exploiting-cors-misconfigurations-for.html, 2016. [accessed Feb-2018].

[15] MAIL LISTS, W. Public-webapps@w3.org mail archives. "https://lists.w3.org/Archives/Public/public-webapps/", 2018. [accessed Feb-2018].

[16] MIKE WEST, M. G. Same site. https://tools.ietf.org/html/draft-west-first-party-cookies-07, 2016. [accessed Feb-2018].

[17] MITRE. Cve-2014-6271. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271, 2014. [accessed Feb-2018].

[18] MITRE. Cve-2017-5638. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5638, 2017. [accessed Feb-2018].

[19] MLLER, J. Cors misconfigurations on a large scale. https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html, 2017. [accessed Feb-2018].

[20] OWASP. Owasp top 10 secuirty issues. https://www.owasp.org/index.php/Top_10_2007, 2007. [accessed Feb-2018].

[21] POPESCU, P. Practical jsonp injection. https://securitycafe.ro/2017/01/18/practical-jsonp-injection/, 2017. [accessed Feb-2018].

[22] RESCHKE, J., AND FIELDING, R. Rfc 7231-hypertext transfer protocol (http/1.1): Semantics and content. june 2014. *URl: http://tools.ietf.org/html/rfc7231*.

[23] REVAY, G. Here it is, the file upload csrf. http://gerionsecurity.com/2013/04/here-it-is-the-file-upload-csrf/, 2013. [accessed Feb-2018].

[24] SCHWENK, J., NIEMIETZ, M., AND MAINKA, C. Same-origin policy: Evaluation in modern browsers.

[25] SINGH, K., MOSHCHUK, A., WANG, H. J., AND LEE, W. On the incoherencies in web browser access control policies. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 463–478.

[26] SON, S., AND SHMATIKOV, V. The postman always rings twice: Attacking and defending postmessage in html5 websites. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[27] STAMM, S., STERNE, B., AND MARKHAM, G. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web* (2010), ACM, pp. 921–930.

[28] TOPF, J. The html form protocol attack. http://www.remote.org/jochen/sec/hfpa/hfpa.pdf, 2001. [accessed Feb-2018].

[29] VAN KESTEREN, A., ET AL. Cross-origin resource sharing. *W3C Working Draft 27 July 2010* (2010).

[30] VAN KESTEREN, A., ET AL. Fetch. https://fetch.spec.whatwg.org/, 2011. [accessed Feb-2018].

[31] VAN KESTEREN, A., ET AL. Cross-origin resource sharing. *W3C Recommendation 16 January 2014* (2014).

[32] VELA, E. sendbeacon let's you send post requests with arbitrary content type. https://bugs.chromium.org/p/chromium/issues/detail?id=490015, 2015. [accessed Feb-2018].

[33] WHATWG. Web hypertext application technology working group. "https://whatwg.org/", 2018. [accessed Feb-2018].

[34] WIKIPEDIA. Apple filing protocol — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/Apple_Filing_Protocol", 2018. [accessed Feb-2018].

[35] WIKIPEDIA. Cross-site request forgery — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/Cross-site_request_forgery", 2018. [accessed Feb-2018].

[36] WIKIPEDIA. Cross-site scripting — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/Cross-site_scripting", 2018. [accessed Feb-2018].

[37] WILANDER, J. Cors-safelisted request headers should be restricted according to rfc 7231. https://github.com/whatwg/fetch/issues/382, 2016. [accessed Feb-2018].

[38] ZHENG, X., JIANG, J., LIANG, J., DUAN, H.-X., CHEN, S., WAN, T., AND WEAVER, N. Cookies lack integrity: Real-world implications. In *USENIX Security Symposium* (2015), pp. 707–721.