# Careful Who You Trust: Studying the Pitfalls of Cross-Origin Communication

Gordon Meiser, Pierre Laperdrix, Ben Stock

# Careful Who You Trust: Studying the Pitfalls of Cross-Origin Communication

Gordon Meiser
CISPA Helmholtz Center for Information Security
gordon.meiser@cispa.saarland

Pierre Laperdrix
CNRS / Univ. Lille / Inria
pierre.laperdrix@inria.fr

Ben Stock
CISPA Helmholtz Center for Information Security
stock@cispa.de

## ABSTRACT

In the past, Web applications were mostly static and most of the content was provided by the site itself. Nowadays, they have turned into rich client-side experiences customized for the user where third parties supply a considerable amount of content, e.g., analytics, advertisements, or integration with social media platforms and external services. By default, any exchange of data between documents is governed by the Same-Origin Policy, which only permits to exchange data with other documents sharing the same protocol, host, and port. Given the move to a more interconnected Web, standard bodies and browser vendors have added new mechanisms to enable cross-origin communication, primarily domain relaxation, postMessages, and CORS. While prior work has already shown the pitfalls of not using these mechanisms securely (e.g., omitting origin checks for incoming postMessages), we instead focus on the increased attack surface created by the trust that is necessarily put into the communication partners. We report on a study of the Tranco Top 5,000 to measure the prevalence of cross-origin communication. By analyzing the interactions between sites, we build an interconnected graph of the trust relations necessary to run the Web. Subsequently, based on this graph, we estimate the damage caused through exploitation of existing XSS flaws on trusted sites.

**ACM Reference Format:**
Gordon Meiser, Pierre Laperdrix, and Ben Stock. 2021. Careful Who You Trust: Studying the Pitfalls of Cross-Origin Communication. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3433210.3437510

## 1 INTRODUCTION

In their early days, Web applications consisted mostly of static content, allowing neither for the interaction with the user without loading a new page nor enabling inclusion of third-party content. Terms like "iframe," "login area," and "asynchronous JavaScript" were foreign words for site operators and developers. Nowadays, Web applications have turned into rich client-side applications, and third parties supply a growing amount of content to a given site from ads and tracking services to the integration with social media platforms. This accumulation of content sources inevitably leads to a steadily increasing demand for interaction between the related parties. By default, any communication between origins is prohibited by the Same-Origin Policy (SOP), which ensures that only documents with the same protocol, host, and port can access each other. The SOP proved to be too restrictive for the modern Web, and the W3C standardized two new ways of cross-origin communication, namely postMessages and CORS. Together with the long-existing concept of domain relaxation, which enables subdomains of a common parent to interact with each other, these methods allow sites to loosen their security parameter. However, by trusting others to provide data or even enabling code execution, whenever the trusted site is attacked through an XSS, the trusting site also falls victim to the attacker.

While the subject of trust in third parties has been well-studied with respect to inclusion behavior [15, 23, 25], we instead focus on a subtly different problem space. Specifically, we look at the trust relationships between Web sites that the aforementioned cross-origin communication mechanisms entail. The main difference to prior work is that an attacker does not need to compromise a third-party server to manipulate the JavaScript it hosts, but it is rather sufficient to find an XSS vulnerability on a trusted communication partner. According to the 2019 Acunetix Web Application Vulnerability Report, up to 30% of Web application are susceptible to XSS [3]. In addition, academic works on client-side XSS have reported exploitable flaws on up to 10% of the tested sites [19, 20, 28]. In contrast, vulnerabilities that may cause an attacker to take over an entire server, are much less frequent. According to the Acunetix report, only 2% of the Web applications they analyzed were susceptible to RCEs and 5% had some type of overflow vulnerability. Hence, our main focus is to ask: what is the ripple effect caused by a trusted site (in terms of cross-origin communication) that is susceptible to an XSS.

Prior works have already studied the insecurity of CORS configurations [4, 8, 17] and postMessages [13, 14, 26, 38]. Notably, though, we are not interested in utterly insecure configurations, but rather want to investigate the threats caused by *necessary* trust sites put into each other. To that end, we analyze domain relaxation, CORS, and postMessages to build a graph of the connections between sites and the trust they share with each other. Specifically, by crawling the Tranco top 5,000 Web sites, we aim to answer the following three research questions related to trust on the Web:

- *(RQ1): How closely intertwined are modern Web sites and how prevalent are the connections between them?* Additionally, we examine the kind of relationships and the type of the interchanged data. The data from our crawl allow us to investigate which problems could potentially arise, and what could be the impact of insecure usage of cross-domain connections.


*ASIA CCS '21, June 7–11, 2021, Hong Kong, Hong Kong*

ACM ISBN 978-1-4503-8287-8/21/06…$15.00
https://doi.org/10.1145/3433210.3437510

- *(RQ2): Can cross-domain communication lead to an increased attack surface of an otherwise secure site?* In other words, can a Web site be attacked through a trusted, vulnerable one?
- *(RQ3): In the face of a classical Web attacker, how vulnerable can a site be?* Based on the prior results, we consider an attacker who exploits reflected client-side XSS flaws, and analyze the impact they can have while leveraging the established trust relations.

**Contributions** By answering those research questions, our paper makes the following contributions:

- We implement a crawling framework which enables us to collect trust relations as well as the JavaScript code involved in cross-origin communication (Section 4).
- We report on a large-scale empirical study of the Tranco Top 5,000 domains w.r.t. the usage of Cross-Domain communication mechanisms (Section 5.1).
- Leveraging the collected data, we build a graph of the established trust relations (Section 5.2) and analyze the security impact of the different cross-origin communication methods (Section 5.3).
- Based on case studies, we illustrate common scenarios where established trust relations can be misused in unforeseen ways (Section 5.4).
- Finally, we demonstrate the real-world exploitability by synchronizing our previous findings with our defined attacker model of an XSS attacker (Section 5.5).

## 2 THREAT MODEL AND PROBLEM SCOPE

Rather than focussing on the inclusion of third-party code, which is a well-studied problem, we instead aim to understand what risks come with the reliance on cross-origin communication methods on the Web. While at first, the issues appear related, they are in fact quite distinct. An attacker can leverage the trust in included JavaScript through two attacks: either, they can attempt to manipulate the content in transport (MitM attack) or compromise the third-party server, modifying the hosted code. The first attack nowadays is mostly thwarted through the widespread usage of HTTPS[1] and the design choices of browsers, which refuse to include scripting content over insecure connections for sites that are delivered via HTTPS. For the second attack vector to be successful, an attacker needs to somehow compromise the server. This can be achieved by directly attacking the server (e.g., bruteforcing SSH/FTP passwords), attempting social engineering attacks to gain access, or by exploiting vulnerabilities. While compromising a high-profile server has severe consequences to all sites that include content from it, the likelihood is also comparatively small. As discussed by [3], only 2-5% of Web applications suffer from a flaw that allows for code execution.

In contrast, XSS vulnerabilities occur much more frequently. According to [3], 30% of Web applications carry such a flaw, and researchers have shown that around 10% of Web sites are susceptible to reflected client-side XSS flaws [19, 20]. Hence, if an attacker can identify a Web application that is trusted by others to send postMessages (which end up being evaled) or is allowed to read authenticated responses through CORS from others, such an application becomes the target to detect XSS flaws on. Given the much higher success rate in XSS exploitation over a full server-side compromise, such an application becomes the prime target for an attacker. In our paper, we therefore focus on this particular danger, which occurs when sites necessarily trust each other.

[1] https://pokeinthe.io/2019/04/04/state-of-security-alexa-top-one-million-2019-04/

## 3 CROSS-ORIGIN DATA EXCHANGE

The Web's most basic security policy is the **Same-Origin Policy** (SOP), which applies a straightforward principle: its restricts access to resources as soon as the origin [37] (**protocol**, **host**, and **port**) differs from the requesting page's own values. The SOP is a fundamental security mechanism that provides boundaries between Web sites and prevents unauthorized access to sensitive information. Doing so, the SOP creates a security barrier around an application which is bounded by the origin. As the Web evolved and became more complex, the SOP proved to be too restrictive. While browsers already implemented a relaxation of the SOP based on a common parent domain (dubbed *Domain Relaxation*), the W3C also standardized new mechanisms to address this demand, namely postMessages and Cross-Origin Resource Sharing (CORS). In the following, we outline how these three mechanisms work and how we consider them to be relevant when we investigate trust between origins and sites.

### 3.1 Domain Relaxation

If two HTML documents on different subdomains, but common parent domain, want to exchange data, they can *relax* the domain part of their origins. In particular, a document can relax its origin to the registrable domain (eTLD+1). By doing so, it allows other documents which also relaxed their origin to the same value to share an origin, meaning the two documents can now access each other fully (i.e., can execute JavaScript in each other's context and read the DOM). For example, if `https://login.example.com` and `https://example.com` want to communicate, they can both set their `document.domain` to `example.com` so that they are allowed by the browser to exchange information. This mechanism requires both origins to opt in and does not enable cross-protocol (HTTP vs. HTTPS) data exchange. In our example, if only the subdomain were to opt in, it would not be able to access resources on the parent domain. Notably, though, if `https://login.example.com` sets it `document.domain` property to `example.com`, it enables any other HTTPS subdomains of `example.com` to gain access; as all the other origin has to do is also relax its own origin. Hence, as long as a single origin relaxes its domain part, it allows read and execute access from *any* subdomain of the relaxed domain.

### 3.2 postMessages

As soon as the communication exceeds the borders of a registrable domain, domain relaxation cannot be used any longer. To nevertheless enable two documents to communicate, the HTML5 specification standardized the postMessage API [36], which allows to send serialized messages between two documents. This mechanism therefore allows to exchange data across origin *and* site boundaries. The only necessity for this to work is that both documents are

```
// sends message (from https://example.com)
var popupFoo = window.open("https://foo.com", ...);
popupFoo.postMessage("Hi there!", "https://foo.com");

// receives message (on https://foo.com)
window.addEventListener("message", receiveMessage, false);
var allowedOrigins = ["https://bar.com", "https://example.com"]
function receiveMessage(event)
{
  if (!allowedOrigins.includes(event.origin))
    return;
  // ...
}
```

**Figure 1: postMessage() example**

```
GET /resources/private-data/ HTTP/1.1
...
Origin: http://a.com


HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: http://a.com
Content-Type: application/xml

[XML Data]
```

**Listing 1: CORS example (simple request)**

loaded in the same browser and have JavaScript references to each other (e.g., through an opened popup window).

Figure 1 shows an exchange of messages between two documents running at `https://example.com` and `https://foo.com`, respectively. Using the reference popupFoo to the opened popup window, `example.com` sends a message to `foo.com` that will be processed in its message handler. One important security aspect of postMessages is that it can provide strong guarantees on any sent message. The sender can specify which origin is permitted to receive the message, and the receiver can verify the origin of the message to ascertain its integrity. In our example, `example.com` specifies `https://foo.com` as the sole recipient of its message. This ensures that if for some reason the opened popup is navigated away, the possibly sensitive message is not accidentally delivered to another origin. Then, `foo.com` verifies that the received message comes from one of the allowed Web sites present in *allowedOrigins* (here either `https://bar.com` or `https://example.com`). While prior work [26] has shown many message handlers either have no check at all or implement them insecurely, e.g., through substring matching or incorrect regular expressions, in our work we are specifically interested in the dangers of trusting parties. That is to say, we assume a postMessage handler to be securely checking the origin of a message, and in light of that evaluate the dangers of trust put into the allowed senders.

### 3.3 Cross-Origin Resource Sharing

The third mechanisms we are considering is Cross-Origin Resource Sharing (CORS). Contrary to the previous two mechanisms, CORS is a set of server-side headers, enforced by the client, which allow a server to allow read access from JavaScript. That is to say, a document makes a request to a remote URL, and CORS governs access to the response. This is in contrast to the other mechanisms, which only involves client-side documents communicating with each other. In a nutshell, CORS is an extension of the XMLHttpRequest API to allow cross-origin content in the browser through explicit authorization [33]. If a browser visiting site A makes a request for data to site B, the SOP will not grant read access to the received content. It becomes readable only if site B is CORS configured and responds with an *Access-Control-Allow-Origin* (ACAO) header indicating that A is allowed as a communication partner.

The list of trusted parties must be implemented internally and programmatically as the standard only support allowing a single origin within a CORS response. If a site wants to allow access from any origin, it can set this header to the * wildcard. In addition, by default CORS does not allow for credentialed requests, i.e., those that have cookies or specific authentication headers attached. To allow for this, the contacted server must opt in by setting the *Access-Control-Allow-Credentials* (ACAC) header to *true*. Importantly from a security point of view, credentialed requests with a * wildcard are forbidden; even if ACAC is set to true, browsers will ignore the header, failing securely. The only way for an insecure configuration is for a site to blindly reflect back the requesting origin in the ACAO header and set the ACAC header to true. While others have demonstrated that this occurs in practice [4, 8], we are again only interested in explicitly specified trust, not misconfigurations, leading to insecurity.

## 4 COLLECTING TRUST RELATIONS

The SOP protects sites from data abuse over cross-origin borders. However, a large percentage of the Web relies on communication extending beyond the borders of the current origin. To provide the wealth of features expected by users, Web sites rely more and more on other sites to provide advanced functionality and exchange of data therefore becomes indispensable. As shown by Stock et al. [31], the number of sites using CORS and postMessages increased substantially in the past decade. They observed that over 65% of the Alexa top 500 in 2016 either received postMessages or sent them.

With this increasing reliance on other sites, our aim in this study is to measure the current level of interconnection between Web sites and map the trust relations between them, allowing to answer our first two research questions, namely:

(1) How closely intertwined are modern Web sites and how prevalent are the connections between them?
(2) Can cross-domain communication lead to an increased attack surface of an otherwise secure site?

We note here that a different domain does not imply another party, e.g., Youtube and Google are the same party, yet have different domains. We are, however, interested in understanding to what extent a compromise of one site could adversely affected others which trust it (through any of the outlined mechanisms).

### 4.1 Collecting One-to-One Relations

To investigate how Web sites are connected with each other, we crawl the top Tranco 5,000 Web sites [24] and record occurrences of cross-origin communication. The crawler is a NodeJS script utilizing Puppeteer [12] to control and automate a Chrome browser.

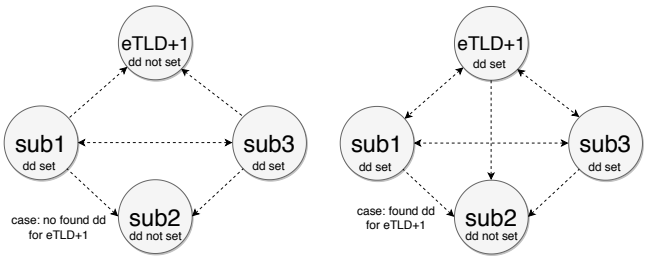


**Figure 2: Trust relationship generation for `document.domain`**

Puppeteer is a library which uses the DevTools Protocol [11] to control the browser. We do not rely on any security exemptions, such as disabling CSP or site isolation, in our crawls. Using the crawler, we search for anchor elements on the visited pages to find subpages, log occurrences of domain relaxation, analyze postMessages, and log CORS-related headers. While we only follow links discovered on the main frame, we record all communication for loaded iframes as well. With the collected data, we build *trust chains* that help us identify and understand security risks originating from these exchanges. In the following we what data we collect and how we combine it to build these chains and subsequently our graph.

*4.1.1 Domain Relaxation.* On each page that we visit, we leverage Puppeteer functionality to inject JavaScript code to modify the setter of *document.domain* and record when it is modified. At the same time, we collect the current *location.href* so that we can precisely identify which origin is relaxing to which domain. When multiple origins in form of subdomains relax their SOP, they gain *execute* privileges equivalent to the rules denoted in Figure 2. Here, the arrows indicate the direction of trust, i.e., in the example to the left, *sub1* trusts *sub2* with *execute* privileges. However, since *sub2* does not set its `document.domain` property (abbreviated as dd in the figure), *sub1* does not automatically gain an *execute* privilege; hence there is no bidirectional trust depicted. In the example on the right, the main page running on the eTLD+1 also sets its `document.domain` property. Hence, it only bidirectionally trusts *sub1* and *sub3*, and unidirectionally trusts *sub2*.

*4.1.2 postMessages.* To record the use of postMessages, we use Puppeteer to inject JavaScript code to hook the *EventListener* of message events. Given the lack of automated tools to analyze postMessage handlers, we rely on manual analysis of the involved JavaScript code. To nevertheless allow for our study's scope, we aim for a certain level of automation. To achieve this, we are interested in extracting (for each pair of incoming message and handler) the snippets of code which handle the specific message. As not all functionality is necessarily implemented in the registered event handler, but rather often the handler itself is just a stub which calls the actual functionality, we need to record all code that interacts with the incoming message. To achieve this, we rely on a JavaScript Proxy object. In particular, instead of passing on the message to the handler, we pass a Proxy around the message. If in the Proxy object, properties like `origin`, `data`, or `source` are accessed, we not only return the values, but importantly also dynamically collect the source code of the calling function. In addition, instead of

```
if(evt.data.msgType == 'successParent'){
 var c = evt.data.msgValue.replace('[[formData]]',
 ↪  evt.data.formData);
 var d = c.replace('[[formData]]','');
 eval(d);
}
```

**Figure 3: Dangerous postMessage handler example**

returning the raw values, we wrap these in yet another JavaScript proxy; this enables us to collect all source code that interacts with the value extracted from the postMessage object (e.g., when the `data` property is assigned to a variable and the variable is passed on). This way, whenever any JavaScript code uses data originating from an incoming message, we can record said code.

This choice of dynamic instrumentation has a functional drawback: if, e.g., an origin check uses the triple-equals comparison, JavaScript first checks the two values for their type. As the origin is now of type Proxy, whereas the value it is compared to is of type String, the comparison always fails. To partially counter this issue, we use mitmproxy [6] to proxy all HTTP connections from the browser. We then check the resource type for being HTML (for inline JavaScript) or JavaScript and for the occurrence of triple-equal signs; if found, we replace them with double-equals. As accessing the *value* of our Proxy object returns the original string and the double-equals comparison omits the type check, the JavaScript code executes as expected. We argue that this does not cause false positives for two reasons: first, coming back to the example of the origin check, the proxy and string *values* still need to be the same. Second, we manually validated all occurrences of problematic handlers in a regular browser without the Proxy functionality, meaning that if during our crawl we incorrectly visited a certain branch of code (due to the changed comparison semantics), that code would not be reached in the test. While this workaround addresses the issues related to triple-equals comparison, JavaScript code can also use explicit `typeof` checks. We opted against replacing those checks, as this might break intended functionality (e.g., if JavaScript code expects a String, but instead receives an Array, simply checking against Object would allow the Array to pass through).

With this mechanism in place, we resort to manual analysis of the collected postMessage handler/message pairs. In particular, we focus on two scenario which are relevant to our work, namely execute and persistent modification:

*Execute privileges*: We check if data originating from postMessages ends up being used in either `eval()`, `innerHTML()`, or `document.write()` (and the derivatives and wrappers such as jQuery's `.html`). To find these cases, we query the database of collected JavaScript code for occurrences of the API calls, and resort to manual analysis of the code. Based on this code review, we then determine if an incoming postMessage can trigger code execution (either directly through invocation of `eval` or by adding script content) and build a proof-of-concept. Figure 3 shows a problematic handler, where the site expects a JSON object with a certain structure (*msgType* must be *successParent*), and evals the *msgValue* attribute. If the execution is successful, we say that the incoming postMessage provides *execute* privileges to the sender.

*Persistent modification*: With the same technique, we can also detect if data from postMessages can cause changes to *arbitrary items* of persistence APIs in the browser, in particular we look for calls to `Local Storage` and `document.cookie` and examine the logged source code. As Steffens et al. [28] have recently shown, numerous sites insecurely use data from these persistent storages in their application, leading to what the authors dubbed Persistent Client-Side Cross-Site Scripting. While these cases are a straightforward security issue, there may exist more subtle problems as well, e.g., modification of a session cookie to conduct a session fixation attack [16] or modification of client-side state in Progressive Web Applications [18]. Hence, we label all cases where a postMessage may cause changes to *arbitrary* storage values as *change persistence*.

Once we have found that an origin grants either privilege in the way described above, we label this as either explicit or implicit trust. We denote *explicit* trust when a document conducts an origin check before handling the message. As we manually examine the source code, we can determine if the trust is for a single domain/origin (e.g., when the origin check compares against a single value) or multiple origins (e.g., through regular expressions). However, there may be cases where there is no origin check at all. As we assume that during our crawls we are not targets of an attack, we rely on collected data to infer *implicit trust*. That is, when origin A handles a postMessage from origin B, we assume A implicitly trusts B as this would be required for proper functionality. The outcome of our analysis therefore is a connection between two origins which either has an execute or change persistence privileges, and where the trust is either explicit or implicit.

*4.1.3 CORS.* A Web developer can use CORS to restrict communication to a set of trusted partners. By looking at each external resource loaded by the browser, we search for *Access-Control-Allow-Origin* (ACAO) and *Access-Control-Allow-Credentials* (ACAC) headers in the responses. If an explicit origin A allowed by B in addition to the credentials header, we assume that the site B trusts A. We denote this trust relation as a *read* trust (A is allowed to read data from B). To create that list, we perform the following two actions:

*CORS data collection:* During our crawls, we can already observe CORS headers of allowed origins. To ensure that these observed headers are not caused by a misconfiguration which blindly reflects the requested origin, we additionally test the CORS-enabled endpoint with a bogus origin (such as `attacker.com`), determine whether they reflect the origin and set the ACAC header, and if so, flag those endpoints as vulnerable. While we exclude them for later analysis, we keep them in the dataset to reason about the usage of CORS on the Web in general.

*CORS allowlist augmentation:* CORS does not immediately deliver the allowlist, but can contain at most one allowed host. Hence, to test for additional hosts, we resort to other cross-domain mechanisms to generate seed origin to check. In particular, when present in a page, we extract the `connect-src` from its Content Security Policy (CSP), which is primarily used to control to which URLs a page can establish an XMLHttpRequest [9]. Hence, if we find that site A allows site B, we test site B's CORS configuration by supplying site A's origin. Next to this, we also rely on policy files for Flash (`crossdomain.xml`) and Silverlight (`clientaccesspolicy.xml`). While we do not consider these technologies relevant to our work, the allowed hosts nevertheless provide additional seeds for CORS testing. Here, for all allowed origins in A's policy files, we test A's CORS configuration using those entries.

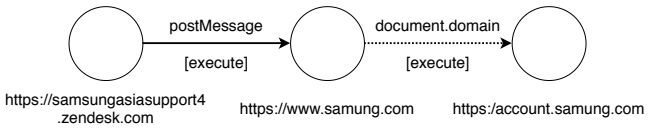


**Figure 4: Example path discovered on samsung.com**

## 4.2 Linking the Collected Data

The results of all our tests thus far are 1-to-1 connections between different origins on the Web. To form a full picture of the Web's trust relations, we use the point-to-point connections to construct a graph. For domain relaxation, all edges between origins are labelled as execute (as discussed in the previous section). For CORS, all relations have the *read* privilege attached to them. For postMessages, we can either label an edge as *execute* if the received message is, e.g., passed to `eval`, or as *change persistence* if the handler allows for arbitrary modification of cookies or Local Storage. Once the individual connections are composed together as a graph, we run our analyses on them.

Figure 4 shows a small subgraph of the collected data. Note the arrows indicates privileges rather than trust, i.e., the arrows' direction shows what an attacker can achieve. Here, we see that https://samsungasiasupport4.zendesk.com can *execute* JavaScript on https://www.samsung.com due to an intended functionality which passes the postMessage's content to an execution sink. An adversary that is able to compromise the zendesk origin can easily pivot to the main page; either by loading `www.samsung.com` in an iframe or opening a popup (note that Samsung does not deploy mechanisms for framing control such as X-Frame-Options or CSP). This now enables the attacker to execute code in the origin of `https://www.samsung.com`. The attacker can now pivot to the account page. Since the account page already relaxes its origin, the attacker's JavaScript code first relaxes its origin to `samsung.com`. Subsequently, the malicious code on `www.samsung.com` now loads the account page in an iframe. Although this has set X-Frame-Options, it explicitly allows Samsung's main page for framing. Since both documents now share the (relaxed) origin, the attacker's code can now exfiltrate account data from the victim. Note that even a defense mechanism that would disallow framing (either X-Frame-Options or CSP) would not help, as the attacker could simply open a popup instead. While this attack is more obvious to a user, it can be run within mere seconds, not allowing the victim to close the popups before their data is stolen. Importantly, there is no connection from Zendesk to Samsung's account domain; yet through the trust relations we observe, compromising the Zendesk page will lead to a compromise of Samsung's account page.

# 5 RESULTS OF LARGE-SCALE ANALYSIS

In this section, we first provide an overview of the state of cross-origin communication, split up by the different mechanisms we

discussed. This data is based on the Top 5k sites (see Section 4.2) of the Tranco Top list [24]. The crawler receives as initial input a list of the first 5,000 Tranco domains, which respond with an HTTP status code between 200 and 399. The crawler is configured to collect links and visit the ones belonging to the same site (registrable domain, eTLD+1) only once and up to depth 2. It enters all new URLs to a relational database until it hits the limit of 1,000 links per site. Given our limit of 1,000 same-site links, we visited a total of 3,078,360 URLs. As 141 sites have not been crawled successfully (e.g. due to timeout errors), we restrict the following analysis to the remaining 4,859 sites. Since our analysis requires manual investigation of postMessage handlers, we decided to only run the crawl once. While this only allows us to capture a single snapshot, we believe the general problematic patterns would occur in every crawl.

## 5.1 Usage of Cross-Origin Mechanisms

Table 1 shows the number of sites making use of any of the cross-origin communication mechanism. In total, we find that 4,509 sites (92.8% of the sites for which we could collect data) use at least one of the mechanisms we investigate on at least one of their subdomains. The word *use* has a different meaning for every element in the table. Concerning domain relaxation this refers to the percentage of sites that set `document.domain` on at least one of their subdomains. For CORS, we distinguish between receiving data and sharing data. Here, receiving data refers to the case when a site makes a cross-origin request and receives a CORS header granting access in the response. Sharing data instead refers to the case where a URL on a given site sends a CORS response header authorizing the request (i.e., either responds with a wildcard or whitelists the requesting origin via `Acccess-Control-Allow-Origin`). Note that the two numbers do not align as there are many more initiators of cross-origin requests than there are sites these requests go to (especially in the top 5,000 sites). For postMessages, the table shows the number of sites that sent and received postMessages, respectively. Here, we also see that more sites receive postMessages than send them. We attribute this to the widespread usage of postMessages for advertisements; here, the ad frame often transmits parameters to the main page, but does not receive any answers.

Most of the sites use CORS (91.6%) or postMessages (84.6%) to exchange data, a smaller part uses document.domain (12.5%) to provide full execution privileges to other subdomains under the same parent domain. Our findings indicate that the trend of increasingly relying on cross-origin communication [31] continues. More importantly, the vast majority of high-profile sites is making use of data from other origins in one way or the other.

*5.1.1 Exchanged Data.* Table 2 shows the five highest-ranking sites which use domain relaxation. We note that the mechanism is very popular, as evidenced not only by the 12% usage in the top 5,000, but importantly even in the highest-ranking sites. Notably, four of the five domains only make use of the mechanism on at least one subdomain, but not the main domain, i.e., its use is related to subdomains being enabled to communicate between each other, but not with the main page. Interestingly, we only observe the usage of `document.domain` on both Twitter and Microsoft on merely a single subdomain; begging the question of the necessity in the first place (as there seems to be no sibling domain to communicate with).

**Table 1: Usage number of cross-origin mechanisms, showing number of sites in the initial dataset and fraction relative to the 4,859 successfully crawled sites**

| Mechanism | # sites | fraction |
|---|---|---|
| document.domain set | 607 | 12.5% |
| CORS data received | 4,435 | 91.3% |
| CORS data shared | 2,225 | 45.8% |
| *CORS union* | *4,454* | *91.6%* |
| postMessages sent | 3,604 | 74.2% |
| postMessages received | 4,084 | 84.1% |
| *postMessages union* | *4,111* | *84.6%* |
| Union of all mechanisms | 4,509 | 92.8% |

**Table 2: Five highest ranking domains using `document.domain` on at least one subdomain**

| site | rank | on main page |
|---|---|---|
| facebook.com | 2 | yes |
| twitter.com | 5 | no |
| microsoft.com | 6 | no |
| tmall.com | 7 | no |
| baidu.com | 10 | no |

We now take a closer look at postMessages, with Table 3a showing the five sites with the most outgoing postMessages. Not surprisingly, social media platforms like Facebook (rank 1) and YouTube (rank 3) can be found here because they use postMessages in order to resize iframes, interchange data about the login status of the user, and send status/control messages for their video players. Representatives of web/data analytics pages like Hotjar (rank 2) and Consensu (rank 4) are also contained within the top 5. Finally, Google uses postMessages for recaptchas, to relay oauth2 requests to *accounts.google.com*, as well as for ads delivery.

Table 3b shows the top 5 sites with the most incoming postMessages. Again, social media platforms (Facebook, Twitter) and web/data analytics sites (Hotjar, AddThis) are the most prevalent, accompanied by Google. This is not particularly surprising, as the nature of the businesses requires a bidirectional exchange of data. However, sites receiving data from this many other parties cannot rely on meaningful origin checks, as they would have to whitelist a sheer endless number of origins. Hence, the secure usage of provided data, especially in those "high-volume" sites is of utmost importance. In particular, as many of these sites act as hubs that both send and receive messages, compromising such a hub allows an adversary to abuse existing trust relations to numerous others.

Table 3c lists the top 5 sites that share data via CORS, meaning that they replied to requests with a CORS header, either whitelisting the requesting origin or responding with a wildcard. We find that of the top 5, all but Facebook make use of CORS with the wildcard `*`. We attribute this to the fact that they all host scripts which are included from others. In particular, if a script's integrity is meant to be ensured through Subresource Integrity (SRI) [10], the hosting site needs to enable cross-origin read access. Complementary to the sites which enable cross-origin read with CORS, Table 3d shows sites that are CORS-whitelisted by most others. The top-rated are

**Table 3: Sites making the most use of postMessage and CORS for cross-domain data exchange, showing both total number of involved sites and *only those within the initial top 5,000 sites***

**(a) Top senders of postMessages**

| | receiving sites (*in top 5,000*) |
|---|---|
| facebook.com | 1,599 (*1,313*) |
| hotjar.com | 1,218 (*824*) |
| youtube.com | 1,001 (*901*) |
| consensu.org | 439 (*373*) |
| google.com | 413 (*375*) |

**(b) Top receivers of postMessages**

| | sending sites (*in top 5,000*) |
|---|---|
| facebook.com | 2,800 (*2,038*) |
| twitter.com | 2,719 (*2,207*) |
| google.com | 2,642 (*2,220*) |
| hotjar.com | 1,200 (*817*) |
| addthis.com | 1,083 (*833*) |

**(c) Top sites sharing data via CORS**

| | sharing with (*in top 5,000*) |
|---|---|
| facebook.com | 2,745 (*1,928*) |
| fontawesome.com | 981 (*731*) |
| cloudflare.com | 910 (*658*) |
| jquery.com | 783 (*633*) |
| criteo.com | 534 (*408*) |

**(d) Top sites reading data via CORS**

| | received from (*in top 5,000*) |
|---|---|
| lemonde.fr | 52 (*13*) |
| pennlive.com | 51 (*17*) |
| telegraph.co.uk | 51 (*17*) |
| lefigaro.fr | 50 (*14*) |
| cleveland.com | 49 (*17*) |

news-related, and we determined that their heavy reliance on CORS stems primarily from ads and SRI-pinned scripts.

## 5.2 Collected Trust Relations

We now model the 1-to-1 relations collected in our experiment into our graph database. In total, this amounts to 291,218 nodes representing distinct origins. Notably, only 113,758 of them are located within the top 5,000 sites we considered. This indicates that a significant fraction of cross-domain communication occurs with sites outside of the highest-ranking sites. Between the nodes, we then first add 1,070,915 domain relaxation edges. Although usage of `document.domain` is not as widespread as other mechanisms, for $n$ subdomains which set `document.domain`, they allow other $n-1$ subdomains access (and vice versa). Importantly, even if a subdomain *sd* sets `document.domain`, yet none of its sibling domains does, *sd* still implicitly allows any sibling domain to gain access (yet not vice versa). Additionally, we add 372,809 CORS relationships, and 154,179 postMessage edges to the graph.

As a next step, we update the graph and create the *execute*, *change persistent* and *read* connections, as described in Section 4.2. All domain relaxation relations become *execute* connections. For postMessages, we manually examined 385 handlers which used eval(), document.write(), and innerHTML() to determine if they would allow a message sender to execute code. Out of these 385 candidates, we determined that 145 handlers enable execution of arbitrary JavaScript code. Of those, 48 contained an origin check, thereby making their trust *explicit*. For the remaining 97, no origin check occurred. Based on our notion of *implicit* trust, we label sites using the handlers as trusting all parties from which they received messages when we crawled the applications. Since several handlers are used across different sites, we in total add 422 *execute* connections to the graph, affecting 70 different sites.

Additionally, we examined the source code of 129 postMessage handlers that use Local Storage and 283 handlers setting cookies. We found that 49 of the Local Storage and 50 of the cookie postMessage handlers allow storing arbitrary values in the Local Storage and cookies, respectively. 10 of the Local Storage handlers used an origin check and 20 of the cookie handlers used checks. For the remaining cases, we resort to our notion of implicit trust. In total, we added 559 corresponding Local Storage relations and 927 cookie relations to the graph, affecting 35 and 30 sites, respectively. Due to two sites allowing both modification of cookies and Local Storage, in total 63 sites trust others with *change persistence*.

Due to the nature of CORS we cannot collect full whitelists at runtime, as CORS is meant to be used to explicitly whitelist a single origin. To that end, we augment the data as described in Section 4.1.3 using CSP, Flash, and Silverlight policies as indicators of whitelist candidates. Using CSP's connect-src, this enabled us to find an additional 923 whitelist entries (from 216 different sites) which were whitelisted on 217 sites. Furthermore, the policy files yielded additional 3,737 whitelist entries (across 241 sites), which enabled 481 additional sites to read data cross-origin. Note that we discard all those entries for sites which blindly reflected the Origin contained in the request. To achieve this, we pick a random origin and if that is reflected, we mark the site as irrelevant for us; this is because we are not interested in insecure configuration (as Chen et al. [4] did), but rather in intended trust relations. As a final check, we ensure that the sites we test also set ACAC header to *true*, as only credentialed requests are within scope of our analysis.

## 5.3 Impact Analysis

`document.domain` enables pages under a common parent domain to access each other, hence the trust is not placed on other parties. Nevertheless, whenever a domain sets the corresponding property, it opens itself up to attacks from any subdomain of the set domain. In our data, we found that out of the 607 sites that use domain relaxation at least on a single subdomain, 231 use it on their main page (`domain.com` or `www.domain.com`) and set the property to the registrable domain. Hence, in those cases, the main page is susceptible to being attacked by any subdomain that can be compromised by an attacker. While the overall number relative to the dataset we crawled is small, it contains a number of high-profile domains (such as Alibaba, American Express, and Facebook).

We now focus on cross-*site* trust, i.e., both postMessages which enable *execute* or *change persistence* as well as CORS *read* accesses across sites. Table 4a shows the results for outgoing *execute* relations, i.e., which sites are most trusted by others. We find that `kameleoon.com`, an advertisement company, has the most trust put into it by others. We analyze this case in more detail in Section 5.4.2. We also find that Facebook and Twitter are highly ranked. This, however, is an artifact of our notion of implicit trust, as we observed handlers enabling code execution on sites that also received messages from Twitter and Facebook.

Table 5a shows the sites that allow most other sites to gain the *execute* privilege on them. We find that there are a number of sites which trust at least 12 other sites to provide code or markup to

**Table 4: Sites most trusted by others with *execute, change persistence, and read* privileges**

| site | # origins | # sites |
|---|---|---|
| kameleoon.com | 44 | 19 |
| facebook.com | 28 | 17 |
| twitter.com | 21 | 13 |
| googlesyndication.com | 17 | 12 |
| 2mdn.net | 11 | 9 |

**(a) *execute***

| site | # origins | # sites |
|---|---|---|
| facebook.com | 50 | 11 |
| twitter.com | 11 | 10 |
| addthis.com | 5 | 5 |
| rcsobjects.it | 24 | 5 |
| instagram.com | 5 | 5 |

**(b) *change persistence***

| site | # origins | # sites |
|---|---|---|
| alicdn.com | 30 | 11 |
| ampproject.org | 9 | 9 |
| alipayobjects.com | 10 | 7 |
| google-analytics.com | 12 | 6 |
| bostonglobe.com | 5 | 3 |

**(c) *read***

**Table 5: Sites trusting most others with *execute, change persistence, and read* privileges**

| site | # origins | # sites |
|---|---|---|
| filgoal.com | 15 | 15 |
| mynet.com | 16 | 13 |
| khabaronline.ir | 13 | 12 |
| hemmings.com | 13 | 12 |
| vnexpress.net | 13 | 12 |

**(a) *execute***

| site | # origins | # sites |
|---|---|---|
| wp.pl | 66 | 33 |
| appledaily.com | 56 | 28 |
| kompasiana.com | 19 | 18 |
| buzzfeed.com | 24 | 17 |
| playbuzz.com | 16 | 14 |

**(b) *change persistence***

| site | # origins | # sites |
|---|---|---|
| tmz.com | 24 | 20 |
| sears.com | 22 | 16 |
| kmart.com | 24 | 14 |
| daraz.pk | 16 | 9 |
| yandex.ua | 57 | 6 |

**(c) *read***

them. These have a significantly increased attack surface over sites that operate in isolation, as compromising any of a site's trusted communication partners is sufficient to attack the site itself. Fortunately, we found that blindly `eval`ing postMessages is a technique which is decreasing in popularity in favor, e.g., of parsing JSON with the `JSON.parse` functionality of JavaScript.

Similar to the *execute* privilege, Table 4b and Table 5b show the sites which are trusted by most others and which trust most others to modify storage (i.e., set arbitrary cookies or Local Storage entries), respectively. Not surprisingly, we again find both Facebook and Twitter in the top trusted sites, primarily based on the implicit trust in those parties. Overall, we find that even the most trusted sites only affected a handful of origins, which indicates that prior work on insecurity of postMessages [26] has primed developers to not blindly trust data originating from third parties. We note here explicitly that in our investigation, we also found instances of handlers where an adversary could only control the content of a predefined cookie or Local Storage item; however, we excluded these from our analysis (as this does not allow for arbitrary changes). On the receiving end, we find that highly-interconnected sites like `wp.pl` allow up to 66 origins from 33 other sites arbitrary modifications of their persistence APIs.

Last, but not least, we focus on CORS read privileges. We note here that a site is marked as having provided CORS-enabled read access as long as a single URL sends appropriate CORS headers. In particular, given the blackbox nature of our analysis, we cannot determine whether the data read from said URL has a specific security impact or not. Notably though, this is in line with other works on CORS, which merely checked the existence of CORS headers when scanning the start pages [4, 8]. In terms of sites that are most trusted by others (shown in Table 5c), `tmz.com` trusts most others, almost all of them some form of news site. Notably, these relations were all found with our extended method of using CSP, Flash, and Silverlight policies to augment the analysis; i.e., the trust relations could not be observed during mere crawling. Nevertheless, as long as an attacker can compromise any of the trusted sites, they may gain read access to `tmz.com` (and likewise for others). Focussing on the sites that have read access to most other sites (Table 4c), `alicdn.com` stands out. However, the large number of domains is only related to the fact that `alicdn.com` is whitelisted by numerous shops belonging to the Alibaba network. While we could not find any vulnerabilities on `alicdn.com` itself, it is security best practice to outsource non-vital functionality to a CDN; as a client-side compromise of such a CDN seemingly has no security impact then. However, as this example shows, finding an XSS flaw in the CDN would in fact enable authenticated read access from numerous shops.

## 5.4 Case Studies

In this section, we discuss case studies which highlight the dangers associated with the trust put into other parties. Note that these are hypothetical cases that highlight patterns of problems. We discuss real-world exploitability through a Web attacker in Section 5.5.

*5.4.1 American Express.* As discussed in the previous section, American Express is among the high-profile sites that use domain relaxation on its main page. When investigating this further, we discovered that in our crawls, we visited a total of 35 different origins belonging to `americanexpress.com`. Notably, only the main page (namely, `www.americanexpress.com`) made use of domain relaxation, relaxing to the registrable domain. This is interesting from both a security and a functionality perspective. Given that no other origin we observed in our crawls applies domain relaxation, there is seemingly no reason for the main page to relax its domain. Since the desired goal is that subdomains with a common parent can communicate, one might expect to observe at least one additional subdomain relaxing its domain as well. Hence, the usage seems to be somewhat moot from a functionality point of view. Naturally, our analysis does not guarantee full coverage of all subdomains and hence might have missed the necessity of domain relaxation. However, there are 34 subdomains which are trusted with *execute*. Looking up the IPs associated with these domain names, we find that they are spread across 12 Autonomous Systems, belonging

```
if (0 == event.data.indexOf("checkKameleoonScriptPresent")){
 if ("https://back-office." +
 ↪  Kameleoon.Internals.configuration.DOMAIN == event.origin) {
  var obtainScriptInstallationCode =
  ↪  event.data.replace("checkKameleoonScriptPresent", "");
  eval(obtainScriptInstallationCode);
 }
}
```

**Figure 5: Excerpt of Kameleoon's postMessage handler**

to 8 different entities. This adds ample attack surface to compromise a single subdomain, allowing an attacker to pivot to the main page. Furthermore, it highlights the danger associated with `document.domain`, as different parties may be hosting subdomains which are implicitly trusted by using the API. Next to this, the subdomain `travel` trusts four origins from other sites (two of which belong to American Express) to execute code via a postMessage, hence opening a second attack vector through the subdomain.

*5.4.2 Kameleoon.* During our manual inspection of potentially exploitable postMessage handlers, we came across JavaScript code from a certain domain which is included on multiple origins. The distributor is `kameleoon.com` [2] and the script is included within top ranked sites like `lefigaro.fr`, `nespresso.com`, and `welt.de`.

Every customer of Kameleoon needs to add scripts to its page to be able to use the functionality offered by Kameleoon. This collection of JavaScript code also contains a postMessage handler (see Figure 5) which is using the `eval` to execute code sent by the Kameleoon backend `https://back-office.kameleoon.com`. So, on the one hand, Kameleoon can execute arbitrary JavaScript in the origin of the customers (which seems to be intended). On the other hand, this will also lead to a mass exploitation of Kameleoon customers if the backend server hosting the configuration Web page for customers can be exploited using any kind of XSS flaw.

It is worth noting that the problem is introduced through a script hosted by Kameleoon and included by its customers. While this already implies that the customers trust Kameleoon to deliver non-malicious code to them (as was explored as the threat model in prior works [15, 23, 25]), an attacker merely has to find an XSS in Kameleoon's backend, rather than needing to compromise their server. As indicated earlier, Acunetix found XSS flaws in about 30% of the tested applications for their 2019 security report [3], whereas only 2% and 5% suffered from RCE or overflows, respectively, which would allow for a server-side compromise.

Overall, 44 origins across 19 sites are affected. During our crawl, we only had a fraction of all possible Kameleoon customers in the initial 5,000 sites. The impact of such a problematic and dangerous method is much more significant when taking all customers of Kameleoon into account. Likely the customer list [1] is just an excerpt and thus can be seen as a lower bound. We also note that the handler is registered via third-party code; i.e., many customers may be unaware of the extended attack surface to their application.

*5.4.3 Advertisement Sandbox Domains.* One common assumption is that the impact of a Cross-Site Scripting flaw can be mitigated through usage of sandbox domains. In particular, assuming that on such a domain there is no relevant information (e.g., cookies) which could be stolen, an XSS is not a problem. However, if another origin or site puts trust into a sandboxed domain, compromising that domain again becomes interesting for an attacker. One such origin is `https://ads.pubmatic.com`. Pubmatic is a supply-side platform (SSP) that allows publishers to sell their ad space to customers. Pubmatic uses its domains to deliver the content to the customers, and to facilitate the realtime bidding for ads, so the customers need to communicate with the SSP. In total, `ads.pubmatic.com` is trusted by 36 origins across 10 sites, which are susceptible to an attack given that Pubmatic is compromised (e.g., through an XSS).

Next to Pubmatic, we found several other instances of sandbox domains which are trusted to provide code to others. Among them, we found seemingly contentless domains such as `c.betrad.com`, `assets.bounceexchange.com` , and `cdn3.doubleverify.com`.

*5.4.4 Trusting Lower-Ranked Sites.* As discussed by Van Goethem et al. [32], the rank of a site is often a proxy to their level of security. Thus, a lower-ranked site might be easier to attack through an XSS or compromised inclusion than a higher-ranking site. To that end, we checked our data set for trust relations towards sites with a lower rank. Out of the 9,611 nodes with outgoing trust relations, we can detect 195 nodes (pertaining to 22 sites) which can act as a starting point to reach higher-ranked nodes. Using these nodes we can reach 52 sites using an *execute* relation, 95 sites using a *read* relation and 34 sites using a *change persistence* relation. This highlights that highly-ranked sites in several instances trust lower-ranked sites, thereby enabling an enlarged attack surface for their own application.

## 5.5 Real-World Exploitability

Conducting the crawl and the additional analyses as discussed, we now have a graph of origins with the individual trust relations between them. This now allows us to answer our third research question: *In the face of a Web attackers, how vulnerable can a site be?* To that end, we leverage our graphs and detect vulnerable paths. Vulnerable paths are defined as several subsequent *execute* relations with an optional *read* or *change persistence* at the end. For every node, we look for relationships to other nodes and save them in a list, including their parent. If we reached a node via *execute*, we recursively apply the algorithm for all nodes related to it (except its parent). This way, when the algorithm terminates, we can assess the impact that compromising each node in the graph could have.

For our attacker, we consider a Web attacker in the classical sense, meaning they can lure a victim to their site and hence force the victim's browser to load any page in an iframe or a popup. Therefore, if the attacker has knowledge about an XSS flaw in any document belonging to some origin $O$, they can force the victim's browser to load the corresponding URL with the exploit payload. Hence, when $O$ is compromised, the attacker can traverse the graph by following all outgoing *execute* privilege edges. This process can be recursively done until no more outgoing *execute* edges are available for traversal. Apart from this, any node in the graph which can be reached via *execute* privileges and has outgoing *read* edges moreover enables the attacker to read content from these sites.

While it is infeasible to know all sites that are vulnerable to an XSS, we can nevertheless resort to using publicly available tools to detect XSS. Arguably, though, testing for server-side XSS flaws may require multiple requests as the server-side components are a

blackbox. Additionally, this may have unwanted side effects, such as accidentally storing an XSS payload in case of persistent server-side XSS flaws. Hence, we resort exclusively to looking for reflected client-side XSS. We rely on the taint engine from Lekies et al. [19] and the exploit generator from Steffens et al. [28], which are both available on Github [5], for this purpose. In particular, we scanned our dataset of 5,000 domains with the original parameters, i.e., up to 1,000 subpages and a depth of 2. Overall, we detected XSS flaws on 1,376 origins, out of 333 have at least one other origins that trust them with *execute*, *change persistence* or *read* privileges.

Figure 6 shows an excerpt of the resulting graph, specifically those nodes that can be compromised through the Web attacker, but do not themselves carry an XSS. Note that the graph does not show *all* `document.domain` relations, simply because of domains such as `focus.cn`, which has an XSS in eight origins, which can affect 290 additional origins through `document.domain`. Focussing on those cases where an attacker can leverage the XSS to pivot to an XSS-free origin, two sandbox origins highlight the dangers of trust relations, specifically `https://ads.pubmatic.com` and `https://eus.rubiconproject.com`. The existing XSS flaws can be used to pivot to three (Pubmatic) and five (Rubicon) other domains' origins, respectively. For Rubicon, the ability to attack, e.g., `finance.detik.com` also enables the attacker to conduct a subsequent read to `connect.detik.com`, which is otherwise out of reach. Similarly, compromising `d3.sina.com.cn` allows the attacker to pivot to `finance.sina.com.cn`, which in turn enables read access to `licashi.sina.com.cn`.

Furthermore, we find that the vulnerabilities in the Russian and Turkish version of Yandex' sites enables a cross-origin read to many of their other origins. Note shown in the graph are other attack vectors like for `case.edu`, in which the XSS on `webapps.case.edu` can be used to attack the job application portal `employment.case.edu` through the usage of `document.domain`. Finally, the change persistence privilege occurs less frequently overall. Nevertheless, we find that both the aforementioned ad sandbox domains have this privilege (shown in green in the graph).

We would also like to stress that our results must be considered lower bounds, as we only scanned in a fully automated fashion for a single class of XSS. Considering that even `google.com` was recently prone to an XSS [22], we strongly believe that a significant fraction of sites that others put trust into can be compromised; leading to the chain reactions described in our work.

### 5.6 Limitations

Like every work using crawler, we are limited by three main factors: lack of logins, unknown coverage of available URLs, and unknown code coverage of the client-side code. The first specifically impacts our analysis regarding CORS, as having read access to unprivileged data (as the crawler is not logged in) might not prove to be dangerous. We share this limitation with prior work related to CORS [4, 8]. In addition, we naturally could not crawl all pages on a given site due our limitation of depth 2 and up to 1,000 links. Third, and maybe most importantly, our analysis does not guarantee code coverage of the JavaScript, i.e., we might have missed relevant postMessages (and the functions processing the received data) as well as invocations of `document.domain`.

In addition, our notion of implicit trust may over-approximate the gravity of the situation. Given that some postMessage handlers did not have an origin check, we estimate trust by observing exchanged messages. This is based on the assumption that we are not being attacked when crawling. Importantly, though, if a page has multiple event handlers, not all messages are necessarily meant for *all* handlers. Although our methods allow us to trace which functions were executed when a given postMessage was handled, we found that there is often no difference in called functions between messages seemingly meant for that handler and those which are not. This is because either a dispatcher just forwards all messages, or all functionality is handled within the receiver. In both cases, there is no difference in the code execute in handling an incoming message. We nevertheless argue that given the complete lack of origin checks, this is a best-effort approximation.

## 6 DISCUSSION

In Web security, the main security concept is often to ensure that only benign code is executed. This includes mechanisms such as Content Security Policy, which ensures only developer-specified code can be executed, or Subresource Integrity (SRI) which aims at ensuring that remote scripts will only be executed if their cryptographic hash matches what is expected. This way, SRI prevents that a compromised third-party server delivers malicious JavaScript to the including site. In particular, SRI therefore defends against a server-side compromise, yet is powerless in the case of our attack scenario, in which an attacker can abuse the trust put into the client-side code running in a particular origin. This holds true for the case of postMessages, where the sender needs to trust another origin to deliver only benign messages, as well as for CORS, where a server allows an origin which executes JavaScript code in the browser of the user.

While our work may well over-approximate the trust put into others (given that in case of a non-existing origin check we assume a handler would handle messages from all communication partners observed in our crawl), we nevertheless believe this to be an overlooked problem in prior work. The Web's dynamic nature requires methods of exchanging data between origins and the postMessage API provides meaningful ways to ensure that trust can be made explicit through origin checks. However, as we have shown, this leads to security problems in practice, especially if other sites are allowed to send code which gets executed. This is aggravated by the fact that many such snippets (like the example of Kameleoon) are third-party provided, meaning the first party may not even be aware of it. Hence, to mitigate such issues, instead of allowing for string-to-code conversion through usage of `eval`, postMessage handlers should be used to call well-defined APIs. Looking in particular at the messages exchanged between Kameleoon and their customers, we find that the critical functionality using `eval` is used in many messages to load entire libraries. While this enables Kameleoon to be very flexible in delivering updates to their code base, it must be considered bad practice.

*Countermeasures.* Rather than implementing library loading by `eval`ing a postMessage, loading of additional code could also be implemented through the programmatic addition of a remote script. This still leaves the sending party with the flexibility to execute any

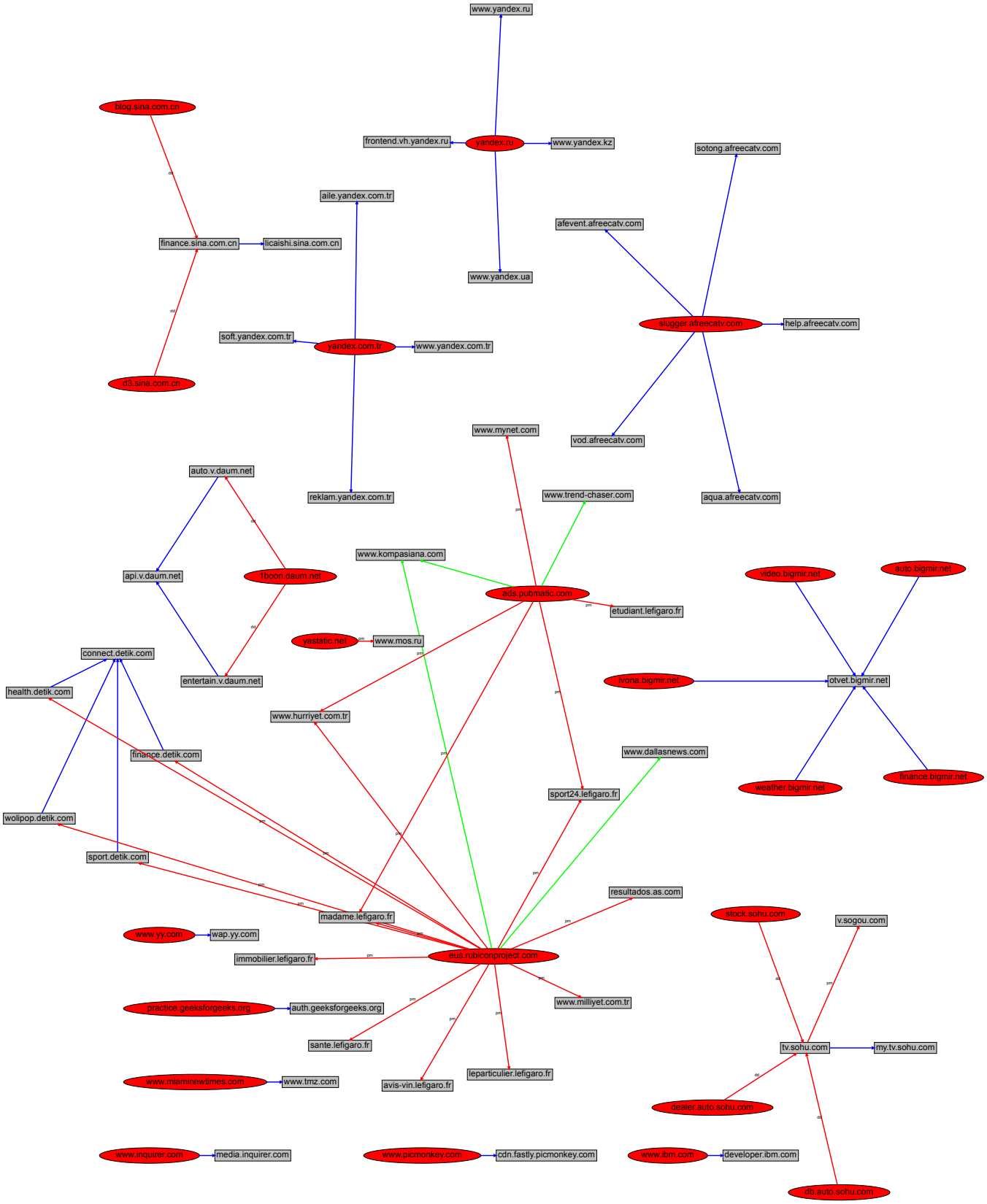


**Figure 6: Real-World Exploitability Graph, showing domains with confirmed XSS (red) and their capabilities towards other nodes, specifically execute (red edges), read (blue edges), and change persistence (green edges). Graph does not contain attack surface through document.domain if this cannot be used to pivot elsewhere.**

code of their choosing, yet the postMessage handler could simply check the script's URL against a list of allowed origins. While this may still leave pages vulnerable in case of open JSONP endpoints on those allowed origins [34], it significantly reduces the attack surface. For cases in which `eval` is used to parse JSON, the obvious solution is to use `JSON.parse` instead.

In addition to the classical cases of direct code execution through postMessages handlers employing `eval` [26], we also investigated whether incoming postMessages may lead to change in persistence APIs. This is particular dangerous given the fact that prior works [18, 28] have shown that client-side application increasingly rely on such persistent storages to persist state or even cache code. Although we could not find any sites that were prone to a persistent client-side XSS *and* had a postMessage handler which enabled arbitrary storage manipulation, our manual analysis might not have revealed all relevant handlers. In addition, we believe that the trends identified by these prior works hint at the fact that usage of client-side storage is on the rise, likely leading to exploitable flaws in the future. Similarly to the execute of incoming messages through `eval`, it also seems that allowing a remote party to arbitrarily set both key and value of a persistence item is merely a feature aimed at being as flexible as possible. Hence, if a postMessage handler really needs to modify such storages, the keys should not be selectable by the postMessage sender, but rather be well-defined upfront. In general, though, compared to the results of Son and Shmatikov [26], the security of postMessage handlers seems to have increased, especially with respect to conducting origin checks.

As far as the usage of the dreaded `document.domain` API goes, Google engineers have already called for its deprecation [35]. We second that opinion, given that postMessages enable a well-defined mechanism to check for sending origins, and also enable fine-grained handling of incoming messages; where domain relaxation directly enables a malicious subdomain to execute arbitrary code. As our data has shown, not only do a number of high-profile sites relax their domain on the main page, but more importantly the example of American Express has shown that not all subdomains of a given domain may reside on the same servers. Hence, usage of this mechanism may expose a Web application to much more harm than the good it does for cross-origin communication.

*Responsible Disclosure.* Given that our XSS findings were discovered with pre-existing tools and notifications had been conducted by prior works already [20, 28], we only reported those flaws for which explicitly name the origins in the paper. As for the dangerous trust relations, we plan to inform those sites that are trusted (such as Kameleoon) to reconsider their dangerous practices, as the sites themselves have little control over this functionality.

## 7 RELATED WORK

In this section, we describe prior work focused on analyzing security mechanisms aimed at protecting cross-origin communication and describe how our work differs from them.

### CORS

The first one to point out the insecurity of the CORS configurations in the wild was Kettle [17]. He discussed several potential misconfigurations, ranging from blind reflection of origins to `null` origins. His results were picked up by Müller [21], who built a scanner aimed at detecting several of these misconfigurations [8]. Building on these works, Chen et al. [4] investigated both the state of CORS on the Web and the implementation difference in browsers, showing that over a fourth of all sites had some CORS issue.

All these works, however, focus on the insecurity of CORS through improper configuration and implementation issues in major browsers. In contrast, our work investigates the explicit trust enabled by CORS, i.e., we explicitly disregard misconfigurations. Given this threat model, we rely on additional sources of information for a more targeted CORS configuration checking, such as CSP, Flash, and Silverlight policies.

### postMessages

In 2013, Son and Shmatikov [26] performed an in-depth analysis of the postMessages mechanism in the wild. Similar to the works on CORS, their analysis focussed on finding missing or incorrectly implemented origin checks in postMessage handlers, finding 84 high-profile sites to be susceptible to an XSS. Stock et al. [31] investigated the top 500 sites over time using the Internet Archive, finding that over half the sites that received a postMessage have at least one handler without an origin check. Yang et al. [38] showed that postMessages are a danger in Webviews on Android, enabling so-called origin stripping attacks. Guan et al. [14] investigated the dangers of having multiple postMessage event handlers in the same page. In their attack scenario called *DangerNeighbor*, they showed that a malicious handler may intercept secret information. To stop such attacks, they propose to use encryption of messages, where only the intended handler knows the key. In concurrent work, Steffens and Stock [27] proposed PMForce, an automated system to test postMessage handlers for vulnerabilities. PMForce was not available at the time of our experiments, but future work can rely on it (instead of manual analysis) to reproduce our results.

Our work is orthogonal to these works, as we again explicitly look for trusted sites exchanging data. Notably, the proposed solution of using cryptographic means is infeasible to protect against our threat model, as the key material would have to be available in the client-side code of the vulnerable trusted site (as postMessages are usually generated from data only available at runtime).

### Other Types of Trust on the Web

Prior research has also focussed on understanding the implications of trust on other layers of the Web. In 2012, Nikiforakis et al. [23] showed how sites increasingly rely on and trust others to deliver their JavaScript. More recently, Ikram et al. [15] investigated the chain of implicit trust, i.e., which other parties' JavaScript is included by directly trusted third parties. With respect to JavaScript on the server side, Zimmermann et al. [39] showed how dependencies in the node.js ecosystem endanger the security of widely-used libraries. More generally, Simeonovski et al. [25] investigated how trusted others to host infrastructure (e.g., email servers) can endanger sites' security.

In contrast to our work, these papers concern themselves with JavaScript inclusions and dependencies. Instead, we focus on client-side mechanisms for data exchange and the dangers associated with trusting other sites to deliver code and data via these mechanisms.

## Client-Side XSS

Arguably, a postMessage handler that allows for execution of JavaScript from an incoming message constitutes an XSS flaw. As it is purely caused by client-side code, we here list relevant works in detection of client-side XSS. The topic of reflected client-side XSS has been studied by many papers [7, 19, 20, 29, 30]. In our work, we rely on the taint engine used in [19] as well as the exploit generator made publicly from Steffens et al. [28]. Our paper augments the findings of prior work by showing that even though origin checks are done, sites can still be attacked through those sites they trust.

# 8 CONCLUSION

Much research in previous years has focussed on securing individual applications or defending against server-side compromise of third parties. In our work, we looked specifically at attacks which are enabled by the trust that sites need to put into others. Coming back to our research questions, we first find that the overwhelming majority of sites rely on some form of cross-origin communication method. Second, we have shown that this trust can lead to situations in which sites can be compromised by other sites they put their trust in. This holds for communication via postMessages as well as for usage of domain relaxation. In particular, we could show that the mechanism of domain relaxation opens a site up to additional attack vectors if subdomains are hosted by other parties. Finally, we map our results to real-world exploitability through automated detection of client-side XSS flaws, showing that even seemingly unimportant sites such as advertisement sandbox domains can cause severe damage to other sites. Given the insights of our work, we especially call on third-party JavaScript vendors to build their mechanisms such that they use well-defined APIs rather than simply `eval`ing code coming from a trusted site.

# ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback. In particular, we thank our shepherd Manuel Egele for his guidance and support in preparing the final version of the paper. The authors would also like to thank Marius Steffens and Sebastian Roth for their help in manual validation of potentially vulnerable postMessage handlers.

# REFERENCES

[1] Kameleoon Customers Page - Site with references to customers of Kameleoon. https://www.kameleoon.com/en/customers/, 2019.
[2] Kameleoon Main Page - Platform for AI personalization and A/B testing. https://www.kameleoon.com/, 2019.
[3] Acunetix. Web application vulnerability report 2019. https://cdn2.hubspot.net/hubfs/4595665/Acunetix_web_application_vulnerability_report_2019.pdf, 2020.
[4] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS. In *USENIX Security*, 2018.
[5] CISPA. "Exploit generator and Taint Engine to find persistent (and reflected) client-side XSS". https://github.com/cispa/persistent-clientside-xss.
[6] Aldo Cortesi, Maximilian Hils, and Thomas Kriechbaumer. mitmproxy - free and open source interactive HTTPS proxy. https://mitmproxy.org.
[7] Stefano Di Paola. DominatorPro: Securing Next Generation of Web Applications. https://dominator.mindedsecurity.com/, 2012.
[8] Horst Görtz Institute for IT-Security. CORS misconfigurations on a large scale. https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html, 2017.
[9] Mozilla Foundation. connect-src, Content Security Policy Level 3, 2016. *Online at https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/connect-src*, 2019.
[10] Mozilla Foundation. Subresource Integrity - MDN. *Online at https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity*, 2019.
[11] Google. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/, .
[12] Google. Puppeteer Documentation. https://github.com/GoogleChrome/puppeteer/blob/master/docs/api.md, .
[13] Chong Guan, Kun Sun, Zhan Wang, and WenTao Zhu. Privacy breach by exploiting postmessage in html5: Identification, evaluation, and countermeasure. In *AsiaCCS*, 2016.
[14] Chong Guan, Kun Sun, Lingguang Lei, Pingjian Wang, Yuewu Wang, and Wei Chen. DangerNeighbor attack: Information leakage via postMessage mechanism in HTML5. *Computers & Security*, 2019.
[15] Muhammad Ikram, Rahat Masood, Gareth Tyson, Mohamed Ali Kaafar, Noha Loizon, and Roya Ensafi. The chain of implicit trust: An analysis of the web third-party resources loading. In *The World Wide Web Conference*, pages 2851–2857, 2019.
[16] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable protection against session fixation attacks. In *ACM SAC*, 2011.
[17] James Kettle. Exploiting CORS misconfigurations for Bitcoins and bounties, 2016. https://portswigger.net/research/exploiting-cors-misconfigurations-for-bitcoins-and-bounties/.
[18] Jiyeon Lee, Hayeon Kim, Junghwan Park, Insik Shin, and Sooel Son. Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications. In *CCS*, 2018.
[19] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of DOM-based XSS. In *CCS*, 2013.
[20] William Melicher, Anupam Das, Mashmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting. In *NDSS*, 2018.
[21] Jens Müller. A simple CORS misconfiguration scanner. https://github.com/RUB-NDS/CORStest, 2017.
[22] Tomasz Andrzej Nidecki. Mutation xss in google search. https://www.acunetix.com/blog/web-security-zone/mutation-xss-in-google-search/, 2019.
[23] 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 *ACM CCS*, 2012.
[24] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco - A Research-Oriented Top Sites Ranking Hardened Against Manipulation https://tranco-list.eu/list/VQXN. In *NDSS*, 2019.
[25] Milivoj Simeonovski, Giancarlo Pellegrino, Christian Rossow, and Michael Backes. Who controls the internet? analyzing global threats using property graph traversals. In *WWW*, 2017.
[26] Sooel Son and Vitaly Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS*, 2013.
[27] Marius Steffens and Ben Stock. Pmforce: Systematically analyzing postmessage handlers at scale. In *CCS*, 2020.
[28] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *NDSS*, 2019.
[29] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise client-side protection against DOM-based cross-site scripting. In *USENIX Security Symposium*, 2014.
[30] 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.
[31] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of client-side web (in)security. In *USENIX Security*, 2017.
[32] Tom Van Goethem, Ping Chen, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Large-scale security analysis of the web: Challenges and findings. In *TRUST*, 2014.
[33] W3C. Cross-Origin Resource Sharing - W3C Recommendation. https://www.w3.org/TR/cors/, January 2014.
[34] 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 *ACM CCS*, 2016.
[35] Mike West. https://twitter.com/mikewest/status/1143431982560493568, 2019.
[36] WHATWG. HTML Living Standard - 9.4 Cross-document messaging. https://html.spec.whatwg.org/multipage/web-messaging.html, 2020.
[37] WHATWG. Origin Specification. https://html.spec.whatwg.org/multipage/origin.html, May 2019.
[38] Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *IEEE S&P*, 2018.
[39] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*, 2019.