JSONPS: Secure an inherently insecure practice with this one weird trick!

Sebastian Lekies
Google
Zürich, Switzerland
slekies@google.com

Damien Engels
Google
Zürich, Switzerland
engels@google.com

Metodi Mitkov
Saarland University
Saarbrücken, Germany
s8memitk@stud.uni-saarland.de

***Abstract*—JSONP, or JSON with Padding, is a cross-origin data exchange mechanism that was proposed in 2005 as a reaction to the extremely strict security model employed by browsers of the day [10]. At that time, the Same-Origin Policy (SOP) prohibited any kind of cross-site communication for security purposes. However, at the same time, this prevented legitimate cross-origin scenarios in which two websites mutually exchange data with each other. JSONP is a workaround that circumvents the SOP by leveraging cross-origin script inclusions to pass data from a backend to a cross-origin webpage. In modern browsers, Cross-Origin Resource Sharing and the PostMessage API offer more powerful and more secure ways to exchange data. However, due to its popularity, its framework support, and a large amount of legacy applications and APIs, JSONP is still very popular today. According to our study, at least 10 % of the Alexa top sites still leverage JSONP. This is problematic since JSONP is inherently insecure: To load data from a cross-origin backend, a webpage has to include and execute a script within its security context. Due to the SOP, the website cannot verify the contents of the script and thus has to blindly grant the third-party server code execution capabilities. To counter this problem, this paper presents JSONPS – a light-weight sandbox for JSONP scripts. By leveraging browser features such as iframes and the PostMessage API, the sandbox enables a website to execute a JSONP script outside of its security boundaries, while still being able to retrieve the data in a secure way. The sandbox can be deployed to large legacy code bases and modern application frameworks with minimal code changes. Furthermore, it considerably increases security when the usage of JSONP is inevitable for backward-compatibility reasons. The sandbox is designed to work in older browsers: At Google, we have deployed the sandbox to major applications serving traffic to millions of users.**

***Index Terms*—Web Application Security, JSONP, Cross-Origin Communication, XSS**

1. Introduction

Modern browsers are capable of concurrently executing different mutually distrusting web applications, while guaranteeing a high level of security and isolation. For example, it is perfectly safe to do online banking via your bank's web site, while other untrusted web applications are opened within separate tabs, windows, or iframes within the same browser. The underlying security policy that enables this level of isolation is the Same-Origin Policy (SOP). The Same-Origin Policy [8] restricts communication of active content to objects loaded from the same origin – the tuple of port, protocol, and hostname used to fetch the resource [3]. The SOP can be considered the most fundamental security policy within browsers as it prevents web applications from different origins from interacting with each other in undesired and insecure ways. However, at the same time, the SOP prevents legitimate use cases in which two websites want to mutually exchange data with each other. In modern browsers, Cross-Origin Resource Sharing [24] and the PostMessage API [8] enable these use cases in a secure fashion. However, both of these techniques have only been added recently and for a very long time there was no browser-supported method for exchanging data despite the SOP. In 2005, Bob Ippolito thus proposed JSON with Padding or JSONP [10], a technique that deliberately bypasses the Same-Origin Policy to allow websites to exchange data with each other. The idea behind JSONP is based on the fact that HTML script tags are capable of loading and executing cross-origin scripts. Thus, if two websites want to exchange data with each other, the receiving website can include a script from the data-providing website. This script contains valid JavaScript code which passes the data as a JSON object to a function defined in the receiving website. After its proposal, JSONP became very popular and wide-spread on the internet, despite its known security problems:

"Your page is still toast if the remote host decides to inject malicious code instead of JSON data"

— Bob Ippolito in the original proposal of JSONP

By including and executing a JSONP script, the including website grants the remote host highly privileged code execution capabilities within its origin. While JSONP has been superseded by other, more secure technologies, it is still frequently used in practice. Our study shows that at least 10 % of the 10,000 Alexa top sites still rely on JSONP. The use of JSONP is still common because it works in old browsers, developers are used to the pattern, and because there is a large number of legacy APIs which only support JSONP. Thus, for the foreseeable future, JSONP will be found in sensitive production web applications. To mitigate the security issues introduced by JSONP, this paper presents a sandbox that can be used as a drop-in replacement for large legacy code bases. By including and executing a JSONP endpoint in a sandboxed origin, a potentially malicious JSONP script can be securely executed without granting privileged access to it. In this context, we will demonstrate different design trade-offs in terms of backward compatibility, ease-of-deployment, and security. Furthermore, we will report on our experience in deploying the sandbox to a large code base consisting of hundreds of million lines of code served to millions of users. With only a few minor code changes, we were able to fully remove potential security threats caused by legacy JSONP endpoints. In summary, this paper makes the following contributions:

- We provide an introduction to JSONP and its security properties. Thereby, we highlight that JSONP scripts are inherently insecure.
- We provide an empirical study to demonstrate that despite its security weaknesses, JSONP is still widely used on the web. Our study finds that at least 10 % of the Alexa top sites still use this inherently insecure legacy technique.
- We present a light-weight sandbox for JSONP scripts which is designed to mitigate the major security flaw of JSONP. The library has been built with large legacy code bases in mind and can serve as a drop-in replacement to secure these code bases with changes to only a few lines of code. Furthermore, the sandbox is fully backward compatible with older browsers.
- We report on the experience of deploying the sandbox to millions of users in a code base of hundreds of million lines of code.

The rest of the paper is structured as follows: Section 2 provides the necessary technical background required for the rest of the paper. Namely, it introduces the Same-Origin Policy, JSON & JSONP, and modern cross-origin sharing mechanisms such as CORS and the PostMessage API. Subsequently, Section 3 presents the results of an empirical study on the usage of JSONP scripts on the Alexa top 10,000 websites. Section 4 then presents the design of our sandbox and reports on the experience of rolling it out to a large legacy code base. Section 5 then summarizes related work, before Section 6 concludes the paper.

2. Technical Background

2.1. The Same-Origin Policy

The Same-Origin Policy is the fundamental security policy of a web browser. Its main purpose is to isolate web applications from each other. More specifically, the Same-Origin Policy prevents objects served from different origins from interacting with each other. A ”web origin” is defined as the tuple of protocol, hostname, and port used to fetch the resource or object[3]. Thus, web applications from different origins are isolated from each other when executed within the same browser. However, there is a noteworthy exception to the SOP: HTML tags such as iframes, objects, scripts, etc. are capable of fetching cross-origin resources despite the SOP: the e.g.:

```
<script src="//remote.com/script.js">
</script>
```

Listing 1: Remote script include

While such remote resources will be fetched and executed by the browser, the access to the resource content is again governed by the SOP. For example, the contents of a cross-origin iframe are inaccessible to JavaScripts on the including page. Hence, the including page can trigger the execution of a script or iframe, but cannot access their source code or contents. However, there is another important exception for script tags: When a script executes, it runs in the context of the embedding origin and not in the origin of the remote page. Hence, a script included from a remote origin runs with the same privileges as a locally fetched script and can thus interact with any other scripts or objects in the embedding origin.

2.2. JSON & JSONP

The JavaScript Object Notation (JSON) is a standardized data exchange format derived from the ECMA-script programming language [5]. The format is based on JavaScript/ECMA-script object literals and data types. The following listing depicts a simple JSON Object:

```
{
  "name": "John Doe",  // String
  "age": 25,  // Number
  "address": { // Object
    "zip_code": 12345,
    "street": "Example Street 1"
  },
  "children":
    ["Janie Doe", "Johnny Doe"], // Array
  "has_job": false, // Boolean
  "job_description": null // Null
}
```

Listing 2: A simple JSON object

Since the format is derived from JavaScript, JSON objects are valid JavaScript object literals as well (note that the opposite is not always true). Thus, the format is very easy to use from within a JavaScript program.

JSON with Padding or JSONP is a mechanism for sharing a JSON object between two websites residing on different origins despite the isolation imposed by the SOP. To achieve this, the website providing the JSON object provides an HTTP endpoint which wraps the JSON object within a function call to a so-called callback function provided via a URL parameter:

```
// URL: ./jsonp.js?cb=foo

foo({"name": "John Doe"});
```

Listing 3: A simple JSONP endoint

Subsequently, the receiving page can define a callback function and include the JSONP endpoint via a script tag:

```
<script>
  function foo(jsonObject) {
    // receives the JSON object.
  }
</script>
<script
  src="//remote.org/jsonp.js?cb=foo">
</script>
```

Listing 4: A JSONP script include

The script tag will fetch the JSONP endpoint with the user's authentication cookies and execute the content as JavaScript. As a result, the callback function is called and the JSON object is passed to it as a parameter.

For a long time, JSONP was the only viable method for exchanging data between two different origins. Thus the technique became very popular, especially in the context of APIs. However, the technique has a major security problem: Due to the SOP, the receiving page cannot inspect the source code of the JSONP endpoint before executing it as a script. This means that the receiving page cannot verify the legitimacy of the script. At any time, the sending page could include arbitrary, potentially malicious code and execute it in the security context of the receiving origin. As included scripts inherit the origin of the embedding page, the JSONP script has fully privileged access to any data on the origin or to trigger any actions in the name of the user.

2.3. Modern cross-origin data sharing: CORS & the PostMessage API

Since JSONP is inherently insecure, browser developers introduced new mechanisms to enable secure cross-origin data sharing. In this paper we cover two separate mechanisms:

- Cross-Origin Resource Sharing (CORS) for secure cross-origin browser-to-server communication
- The PostMessage API for secure cross-origin in-browser communication

2.3.1. Cross-Origin Resource Sharing. Similar to JSONP, CORS is a mechanism to work around the restrictions enforced by the SOP, but in a safe way with browser support. To enable CORS communication, a data provider (a server) and a data receiver (a website within the user's browser) need to cooperate to exchange data across origins. Due to SOP, the browser usually blocks access to cross-origin HTTP responses. However, CORS enables a server to allowlist other origins to access cross-origin responses for certain endpoints. By setting an HTTP response header called *ACCESS-CONTROL-ALLOW-ORIGIN: <origin>*, a server can indicate to a browser that the specified origin is allowed to access the contents of the response. If the specified origin matches the origin of the receiving page, the browser provides access to the response. If no header is provided or if the origins do not match, access is denied. As opposed to JSONP, CORS works with standard XML HTTP requests and doesn't require the inclusion of a cross-origin script. Thus, CORS is a more modern and secure technique for sharing data across origins. However, at the same time JSONP is still in use in a large number of legacy APIs not supporting CORS. Retrofitting old applications and APIs with CORS is problematic since it might require considerable changes to both the client/web application and the server/API. Especially since client and server are usually maintained by different parties, it can sometimes be difficult to change the underlying technology. We generally recommend the usage of CORS over JSONP whenever possible. However, in this paper we will focus on complex legacy applications that cannot easily switch to CORS for the reasons outlined above.

2.3.2. The PostMessage API. Another mechanism that was introduced to enable cross-origin communication is the PostMessage API [8]. Instead of focusing on browser-to-server communication, the PostMessage API enables cross-origin communication within the browser for applications running within different windows, tabs, or frames. More specifically, the API allows two web pages to exchange messages with each other. To do so, the receiving page registers an event handler for message events. Subsequently, the sending page can send a message to the handler via the postMessage function on a window handle object. The following code listings depict how to receive and send messages.

```
// message handler function
var foo = function(msg) {
  // only accept messages from
  // trusted origins
  if (isTrustedOrigin(msg.origin)) {
    // process message data
    msg.data...
  }
};

window.addEventListener("message", foo);
```

Listing 5: Receiving a PostMessage

```
<iframe
  id="frame"
  src="//remote-origin.com">
</iframe>
<script>
// receive a window handle
var windowHandle =
  document.getElementById("frame");

// The object to send.
var msg = {
  "name": "john doe"
};

windowHandle.contentWindow.postMessage(
  msg, "http://remote-origin.com");

</script>
```

Listing 6: Sending a PostMessage

As messages are serialized, the API supports standard JavaScript objects and basic data types. It does not support complex data such as JavaScript functions or DOM nodes. However, as it supports standard JavaScript objects, it implicitly supports JSON objects as well.

When receiving a message, a page will deliver the message event to all event handlers registered within the page. Hence, messages are essentially broadcast to all handlers within the receiving page. This can lead to confusion in mashup-style applications in which multiple handlers are waiting for messages from different senders. For this reason, the PostMessage API also supports so-called MessageChannels. MessageChannels are a bi-directional communication channel used to send messages to specific message handlers instead of broadcasting them. Later on, we will use MessageChannels in our JSONP sandbox implementation.

3. JSONP on the Web

To study the usage of JSONP on the web, we conducted a small empirical study of the Alexa top 10,000 web sites. In this section, we first present our methodology, before we then analyze the results.

3.1. Methodology

To assess how wide-spread JSONP still is on the web, we crawled the landing pages of the Alexa top 10,000 websites. Furthermore, we additionally crawled up to 10 randomly chosen sub-pages residing on the same origin. For each of the fetched pages, we attempted to detect cross-origin JSONP scripts. To do so, we recorded all cross-origin request-response pairs with a response content type indicating JavaScript content. Subsequently, we identified JSONP scripts in the following way: For each identified cross-origin script, we went through each GET parameter key-value pair and tried to find a function call within the script content with the same name as the parameter value. To do so, we took the parameter value and concatenated it with "({" and matched the resulting string against the response body. If the response body contained such a string, we replayed the requests with a changed parameter value. If the new response also contained the new parameter value
+ "({", the script was marked as a JSONP script.

3.2. Study Results

During the crawl, the crawler successfully visited 9,997 of the top 10,000 websites. On the landing pages and on the up to 10 sub-pages, we recorded a total of 1.2 million cross-origin script includes. Subsequently, we applied our methodology to classify scripts as JSONP endpoints. In total, our crawler identified 7,002 JSONP requests originating from 974 of the visited domains. Thus, at least 9.74 % of the Alexa top 10,000 websites still use JSONP. However, our numbers need to be considered as lower bounds, as our study did not deep crawl the sites and did not authenticate to uncover more complex parts of the applications. Our results demonstrate that JSONP is still widespread in practice and that there is a considerable amount of legacy applications and APIs which still rely on this inherently insecure cross-origin data sharing technique. To secure such applications, we thus present JSONPS, a drop-in sandbox for securing large legacy code bases.

4. JSONPS: A simple JSONP sandbox

This section outlines the design principles behind JSONPS and implementation details related to the browser features it leverages. It also discusses how we rolled out the new mechanism to a large number of applications with little effort.

4.1. Design Overview

The design goal of JSONPS is to provide a safe alternative for libraries that provide JSONP functionality.

To provide safety, we need to ensure that the JSONP endpoint has no additional privileges when compared to a CORS-enabled JSON endpoint serving the same data. In particular, this means that we can't allow the JSONP script to execute in the same origin as the client application. More subtly, we also want to ensure that different JSONP endpoints cannot interfere with one another.

To ensure that deployment of JSONPS is practical, we also preserve full API compatibility with existing JSONP libraries. This allows JSONPS to be a drop-in replacement for JSONP without requiring any changes to the serving endpoint or impacting normal application logic. To achieve this, we provide an isolation mechanism that does not rely on additional server-side resources in most cases. In Section 4, we discuss how to expand compatibility for older browsers and stricter Content Security Policies, in which case there is a fixed upfront cost to serve a static file.

At a high level, JSONPS works by creating a local sandboxed environment which loads JSONP scripts in a document with a synthetic origin containing only the definition of the callback function required by the JSONP script. The callback function will receive the data through the normal JSONP mechanism and serialize the data to ensure that it is a valid JSON object. The serialized data will then be sent to the main application where it can be safely de-serialized.

4.2. Sandboxing Javascript

The easiest way to sandbox JavaScript execution to a different origin is to create an iframe with a synthetic origin whose document contains the script we want to execute. While there are other, non-native mechanisms that try to provide sandboxing within the same origin [17], these approaches have been shown to be prone to bypasses [6] and would require loading an additional, non-trivial amount of code. Loading additional code would in turn make deployment more difficult, as a typical JSONP library is composed of very little code.

There are three main mechanisms to create a local iframe document with a synthetic unique origin:

1) Setting the iframe's "srcdoc" attribute with the content of the document in conjunction with a sandbox attribute with the value 'allow-scripts'. For browsers which support these attributes, this is the most convenient way, as it is not subject to potential Content Security Policy restrictions on the "child-src" and "frame-src" directives.

2) Setting the iframe's "src" attribute with the "data:" URI which encodes the contents of the document. This is a practical alternative for browsers which do not support the "srcdoc" or "sandbox" attributes. The "srcdoc" attribute takes priority over the "src" attribute, which means this is only used as a fallback when the browser does not support the "srcdoc" attribute. Since the "srcdoc" attribute is a more recent feature than the "sandbox" attribute in all browsers, this also ensures that the "sandbox" attribute is set correctly whenever the iframe loads from its "srcdoc" attribute. Even in rare cases where the browser might not support the "sandbox" attribute, most browsers will load documents from 'data:'-uris in their own synthetic origin.
3) Setting the iframe's "src" attribute with an object URL pointing to a blob containing the document in conjunction with a sandbox attribute with the value 'allow-scripts'. While this method allows loading larger documents (data-uris are limited to 4KB in older Edge versions), we do not need it as the behaviour of the sandboxed document can be expressed in less than 4KB characters. There are other drawbacks to using documents backed by blobs, as these are typically loaded in the same origin as the parent application and might leak memory if their URLs are not properly revoked.

We use a combination of "src", "srcdoc", and "sandbox" in JSONPS to maximize compatibility with older browsers while preserving security guarantees:

```
const script = /* ... */;
const html = `<script>${script}</script>`;
const url = 'data:text/html;' +
         'charset=UTF-8;base64,' +
         btoa(html);

const iframe =
    document.createElement('iframe');
iframe.sandbox.value = 'allow-scripts';
iframe.setAttribute('srcdoc', html);
iframe.setAttribute('src', url);
```

Listing 7: Creating a sandboxed iframe

4.3. Communicating via Message Channels

Once we have a safe execution environment, we still need to communicate with it to ensure we can retrieve the data that has been loaded. To do this, we set up a dedicated MessageChannel between our library and the iframe. The purpose of doing it in this manner is twofold: on the one hand, we ensure that no other code can interfere with the communication with the sandboxed iframe, and on the other hand, we avoid interfering with potentially existing message listeners in the main application.

Since the sandboxed iframe is designed to run JSONP scripts, it could potentially be tricked into loading malicious code by receiving a message from another window. This might in turn lead to the exfiltration of query parameters, or worse, tampering with the loaded data. By establishing a dedicated MessageChannel and sending one of the ports along with the first message, we can establish a chain of trust with the iframe and ensure no tampering has occurred. To root this chain of trust, we need the iframe to know which initial message to trust. We can achieve this by creating a random nonce, passing it to the iframe as part of the loaded document, and sending it again in the initial message. The iframe's startup behaviour consists of dropping any messages that do not contain the nonce and no longer listening for messages once the dedicated MessageChannel is established.

An additional benefit we get from establishing the MessageChannel outside of the sandboxed iframe is that the iframe never has to use postMessage on the main application's window. This in turn ensures that we never accidentally trigger a generic message listener and potentially change the application's logic.

```
const nonce = /* ... */;
iframe.addEventListener('load', () => {
  const chan = new MessageChannel();
  const msg = {nonce};
  iframe.contentWindow.postMessage(
      msg, '*', [chan.port2]);
  chan.port1.onmessage = (msg) => {
    /* Handle message from sandbox */
  };
}, {once: true});
```

Listing 8: Setting up MessageChannel in parent

```
const nonce =
  document.body.getAttribute('nonce');

function handle(msg) {
  if (msg.data.nonce !== nonce) {
    /* Ignore message */
    return;
  }
  /* Stop listening on window */
  removeEventListener('message', handle);

  const port = msg.ports[0];
  /* Channel established */
}

addEventListener('message', handle);
```

Listing 9: Setting up MessageChannel in sandbox

4.4. Fetching data

Once we have established a trusted channel, we can start issuing JSONP requests securely. Most JSONP utilities will accept a URL for the JSONP endpoint, then asynchronously return the results through a provided callback or using a Promise-based API. While there are often several additional arguments that can be provided, we ignore them here for the sake of simplicity. We also use async await notation for code brevity, but the same can be accomplished using Promise-based APIs or callbacks.

```
async function fetch(uri) {
  const iframe = await loadSandbox();
  const port = setupChannel(iframe);
  port.postMessage({uri});
  let callback;
  const result = new Promise((resolve) => {
    callback = resolve;
  });
  port.addEventListener('message',
      (msg) => {
    callback(JSON.parse(msg.data));
    iframe.remove();
  });
  return result;
}
```

Listing 10: Sending a fetch request to the sandbox

```
port.onmessage = (msg) => {
  // Setup JSONP handler
  window['jsonp'] = (data) => {
    // Ensure data is serializable
    const msg = JSON.stringify(data);
    port.postMessage(msg);
  };

  // Load JSONP script
  const script =
      document.createElement('script');
  script.src = msg.data.uri
      + '?callback=jsonp';
  document.head.appendChild(script);
  document.head.removeChild(script);
};
```

Listing 11: Fetching data in the sandbox

4.5. Limitations and tradeoffs

While the mechanism presented so far allows for a drop-in replacement of current JSONP implementations in most cases, we are aware of some limitations in specific cases. We present these limitations in the current section and propose solutions. One such limitation comes from the mechanism behind JSONP. Despite not being intentionally part of the design, JSONP allows loading more than just JSON objects. Since it can load arbitrary JavaScript, JSONP can also be used to transfer object types that are outside of the JSON specification. One common example of such objects are Date objects.

```
// Valid JSONP
jsonp({d: new Date('December 17')})

// Output from JSONPS
{"d": "2001-12-16T23:00:00.000Z"}
```

Listing 12: Example of valid JSONP, but invalid JSON

This limitation is intentional, as trying to support it fully wouldn't be possible in JSONPS due to the structure-clone algorithm that is run on postMessage. This behaviour can also cause brittleness in applications that rely on it as there might be unintended side effects (e.g. inconsistent timezone information). During our rollout of JSONPS, we have not encountered any application where this limitation was an issue.

There are two other limitations that we are aware of, both of which relate to our use of local HTML documents:

1) Internet Explorer supports neither the "srcdoc" attribute, nor "data:"-uri or Object URLs with an HTML mime-type. To our knowledge, there is no mechanism to load a local resource as an HTML document in Internet Explorer.
2) When loading embedded resources, the browser applies the Content Security Policies that are delivered with the resource, or automatically propagates the policies of the embedding context if the resource is not loaded from the network. In practice, this means that while the JSONP call itself can be made secure, the Content Security Policy of the application still has to contain the JSONP endpoint, which could be used as a CSP bypass.

Both of these limitations can be mitigated by hosting a static version of the iframe's document and loading it from the network. The nonce can then be passed as a URL parameter. While the iframe's default behaviour is dangerous if served on its own, it can be safely served from any origin as long as it is served with a Content Security Policy of 'sandbox: "allow-scripts"'. For additional protection, the sandbox can also check that the sandboxing was successful before proceeding with normal operation.

```
function isSandbox() {
  if (self.origin) {
    return self.origin === 'null';
  }
  // for IE/Edge
  return location.host === '';
}

if(!isSandbox()) {
  throw Error('sandboxing error');
}
```

Listing 13: Throw error if sandboxing failed

Alternatively, depending on an application's needs, it is also possible to detect when the code is running on Internet Explorer and gracefully degrade the security expectations while preserving the functionality of the application.

4.6. Rolling out JSONPS to a large code base

To validate the approach behind JSONPS, we first rolled it out to a common charting library whose use of JSONP caused several known vulnerabilities in sensitive Google applications. This integration was then submitted as a Vulnerability Research Grant [7] to the researcher who found the vulnerabilities, resulting in a few findings that helped us improve the initial design [14].

We then rolled out JSONPS to 40 callsites across the Google codebase which were relying on the Google Closure library's JSONP utility [2] with a non-literal uri paramater. These callsites, in aggregate, affected millions of users and no production issue has been reported as a result of these changes.

5. Related Work

The potential security threats of including JavaScript from remote hosts have been frequently studied in the past. Nikiforakis et al. [19] measured how even top sites often depend on numerous inclusions, which can include expired domains. Stock et al. [23] documented that the trends identified by Nikiforakis et al. [19] grew over time, with sites relying on others more and more to include script content. They also indicated that JSONP was increasingly used until 2016 in the top 500 sites they studied at the time. Arshad et al. [1] developed techniques to detect malicious scripts in what they referred to as inclusion chains. Ikram et al. [9] in turn analyzed how often such malicious inclusions occur in practice, showing that around 1.2% of domains are included (implicitly) within the top sites. In a similar vein, but more generally geared towards the increasing reliance on third parties, Kumar et al. [11] outlined security challenges, such as lacking HTTPS adoption, which originate from implicit trust.

With respect to security problems arising from third party inclusions, Lauinger et al. [12] showed that more than one third of the top 75,000 sites at the time included libraries with vulnerabilities. Furthermore, work on client-side XSS has shown that third parties often cause vulnerabilities in non-library code [16, 18, 22]. Most recently, Steffens et al. [21] studied how the inclusions of third parties may interfere with a first party's ability to deploy security mechanisms such as CSP and SRI.

In contrast to these works, Lekies et al. [13] showed that relying on script inclusions to bypass the Same-Origin Policy has even more pitfalls. Specifically, they studied Cross-Site Script Inclusion, an attack in which the attacker relies on the fact that the resource (which contains sensitive data) can be included from any origin. Based on this, the authors were able to find vulnerabilities in high-profile services. Our threat model, however, differs from their work, as we aim to avoid the dangers of a compromised third party rather than attacking said third party.

In the space of cross-origin communication, recent works have shown that the seemingly secure CORS mechanism is also often configured incorrectly [4]. Additionally, Meiser et al. [15] and Squarcina et al. [20] showed that even the necessary cross-origin communication can lead to severe risks for the communication partners.

6. Summary & Conclusion

JSONP is an inherently insecure workaround to enable cross-origin communication within the boundaries of the extremely strict security model of early browsers. While today the technique has been superseded by more secure technologies such as Cross-Origin Resource Sharing or the PostMessage API, there is still a substantial amount of legacy applications relying on JSONP. According to our study, at least roughly 10 % of the Alexa top 10,000 websites still leverage JSONP for cross-origin communication with other applications. To secure old legacy code bases, we have developed JSONPS, a light-weight sandbox for JSONP scripts. The library can serve as a drop-in replacement for frameworks to secure inherently insecure usages of JSONP. In this paper, we have presented the trade-offs of different design alternatives and identified the best solution based on backward compatibility needs, ease-of-deployment, and security. To validate our approach, we deployed the sandbox to a legacy code base consisting of hundreds of million lines of code serving traffic to millions of users. Our experience demonstrates that such code bases can be secured with only minor code changes and effort.

References

[1] S. Arshad, A. Kharraz, and W. Robertson, "Include me out: In-browser detection of malicious third-party content inclusions," in *Financial Crypto*, 2016.

[2] T. C. L. Authors, "goog.net.Jsonp," Online https://google.github.io/closure-library/api/goog.net.Jsonp.html, 2021.

[3] A. Barth, C. Jackson, and I. Hickson, "The web origin concept," RFC 6454, December, Tech. Rep., 2011.

[4] J. Chen, J. Jiang, H. Duan, T. Wan, S. Chen, V. Paxson, and M. Yang, "We still don't have secure cross-domain requests: an empirical study of {CORS}," in *USENIX Security*, 2018.

[5] ECMA, *ECMA-404: The JSON Data Interchange Format*. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), 2013. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-404.htm

[6] Google, "Caja," https://github.com/googlearchive/caja, 2020.

[7] ——, "Vulnerability Research Grant Program Rules – Application Security – Google," Online https://www.google.com/about/appsecurity/research-grants/, 2021.

[8] I. Hickson, "Html5 specification," World Wide Web Consortium, Tech. Rep. 1.5610, 2012.

[9] M. Ikram, R. Masood, G. Tyson, M. A. Kaafar, N. Loizon, and R. Ensafi, "The chain of implicit trust: An analysis of the web third-party resources loading," in *The Web Conference*, 2019.

[10] B. Ippolito, "Remote json - jsonp," *http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/*, 2005.

[11] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey, "Security challenges in an increasingly tangled web," in *WWW*, 2017.

[12] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *NDSS*, 2017.

[13] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript." in *USENIX Security Symposium*, 2015.

[14] LiveOverflow, "Script Gadgets! Google Docs XSS Vulnerability Walkthrough," Online https://www.youtube.com/watch?v=aCexqB9qi70, 2020.

[15] G. Meiser, P. Laperdrix, and B. Stock, "Careful who you trust: Studying the pitfalls of cross-origin communication," *ACM AsiaCCS*, 2021.

[16] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out domsday: Toward detecting and preventing dom cross-site scripting," in *NDSS*, 2018.

[17] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe active content in sanitized javascript," *Google, Inc., Tech. Rep*, 2008.

[18] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns, "Scriptprotect: Mitigating unsafe third-party javascript practices," in *ASIACCS*, 2019.

[19] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote javascript inclusions," in *CCS*, 2012.

[20] M. Squarcina, M. Tempesta, L. Veronese, S. Calzavara, and M. Maffei, "Can i take your subdomain? exploring related-domain attacks in the modern web," in *USENIX Security*, 2021.

[21] M. Steffens, M. Musch, M. Johns, and B. Stock, "Who's hosting the block party? studying third-party blockage of csp and sri," in *NDSS*, 2021.

[22] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, "From facepalm to brain bender: Exploring client-side cross-site scripting," in *CCS*, 2015.

[23] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in) security," in *USENIX Security*, 2017.

[24] A. van Kesteren. (2013, Jan.) Cross-origin resource sharing. [Online]. Available: http://www.w3.org/TR/cors/