

Introduction

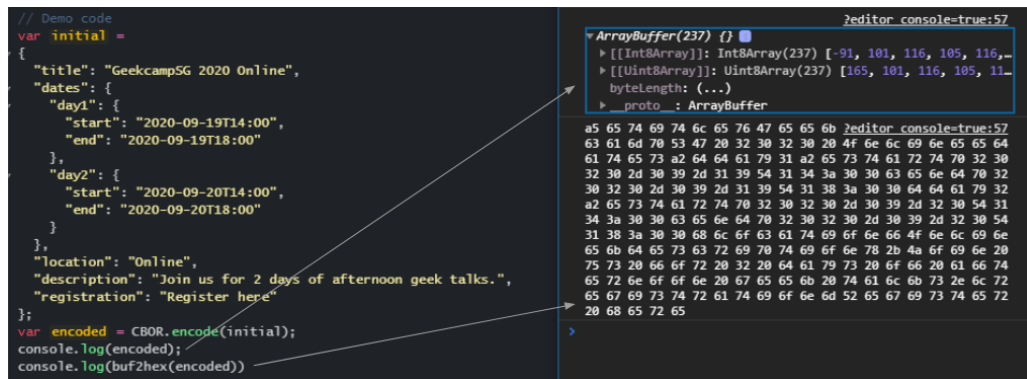
Recently, a new technology on the Web was introduced by one of the big Internet giants, Google. This technology, like many that Google have created, have been criticized for privacy and security issues. The technology is called Web Bundles, which was formerly known as Bundled HTTP Exchanges. In particular, this one aims to allow communication to happen offline or over Bluetooth, but it can be used towards other webpages (Web.dev). What this new technology allows is for the creation of web bundles that contain full webpage in a single file. These are part of a recent standard called Web Packaging, which has similar goals, including Signed HTTP Exchanges, which allows for checking the integrity of WebBundles. In this single file, we can contain the HTML files, CSS stylesheets, Javascript files, images, etc. that would otherwise be requested and sent in responses through HTTP requests to load a single webpage (Web.dev). This allows for the ability to build a self-contained web app that's easy to share and usable without an internet connection, a feature that regular applications do not possess (Web.dev). However, there have been some concerns raised about the possible exploits of this new technology that we will go into further in this paper.

Overview

A web bundle is a Concise Binary Object Representation (CBOR) file with a “.wbn” extension. CBOR is a binary format loosely based on JSON. It's an archive format with a parsable index and easy-to-read individual files based on their offset in the bundle, allowing for

© 2021 siliconninja, Thomas Nandola and Ricardo Hernandez. Paper (except for Figures 1-2) licensed under the CC-BY 4.0 license, available at <https://github.com/siliconninja/CSSE340-Web-Bundles-Research/LICENSE>.

significantly faster processing and transfer speeds at the cost of human readability (Knakal, 2020). It is not a standard compression format, like .tar or .zip, but it can compress and serialize JSON data (Mohamed, 2020) and other types of data, illustrated in Figure 1 below.



```
// Demo code
var initial =
{
  "title": "GeekcampSG 2020 Online",
  "dates": {
    "day1": {
      "start": "2020-09-19T14:00",
      "end": "2020-09-19T18:00"
    },
    "day2": {
      "start": "2020-09-20T14:00",
      "end": "2020-09-20T18:00"
    }
  },
  "location": "Online",
  "description": "Join us for 2 days of afternoon geek talks.",
  "registration": "Register here"
};
var encoded = CBOR.encode(initial);
console.log(encoded);
console.log(buf2hex(encoded))
```

```
ArrayBuffer(237) {}
  ▶ [[Int8Array]]: Int8Array(237) [-91, 101, 116, 105, 116, ...]
  ▶ [[UInt8Array]]: UInt8Array(237) [165, 101, 116, 105, 116, ...]
  ▶ byteLength: (...)
  ▶ proto: ArrayBuffer
a5 65 74 69 74 6c 65 76 47 65 65 6b 2editor console: true: 57
63 61 6d 70 53 47 20 32 30 32 30 20 4f 6e 6c 69 6e 65 65 64
61 74 65 73 a2 64 64 61 79 31 a2 65 73 74 61 72 74 70 32 30
32 30 2d 30 39 2d 31 39 54 31 34 3a 30 30 63 65 6e 64 70 32
30 32 30 2d 30 39 2d 31 39 54 31 38 3a 30 30 64 64 61 79 32
a2 65 73 74 61 72 74 70 32 30 32 30 2d 30 39 2d 32 30 54 31
34 3a 30 30 63 65 6e 64 70 32 30 32 30 2d 30 39 2d 32 30 54
31 38 3a 30 30 68 6c 6f 63 61 74 69 6f 6e 66 4f 6e 6c 69 6e
65 6b 64 65 73 63 72 69 70 74 69 6f 6e 78 2b 4a 6f 69 6e 20
75 73 20 66 6f 72 20 32 20 64 61 79 73 20 6f 66 20 61 66 74
65 72 6e 6f 6f 6e 20 67 65 65 6b 20 74 61 6c 6b 73 2e 6c 72
65 67 69 73 74 72 61 74 69 6f 6e 6d 52 65 67 69 73 74 65 72
20 68 65 72 65
```

Figure 1: The encoding of JSON into the CBOR format. (Mohamed, 2020)

A web application is a software application that includes a client-side and server-side that the client can request and run in their browser. Generally, the web applications operates so that the client/user issues a request from their browser/user interface over the internet, the web server then forwards this request to the appropriate web application server, the web application server then processes that request, the web application server then transfers its response to the web server that then responds to the client with the requested information that is then usually displayed on the user's display (Gibb, n.d.). On the contrary, Web Bundles, formerly known as Bundled HTTP Requests (WICG, see commit message at top), encapsulates one or more HTTP resources in a single file (WICG). It can include one or more HTML files, JavaScript files, images, or stylesheets.

With a web bundle, your browser reads a byte stream from a bundle and uses provided metadata to simulate HTTP exchanges and access resources inside it, as illustrated in Figure 2 below.

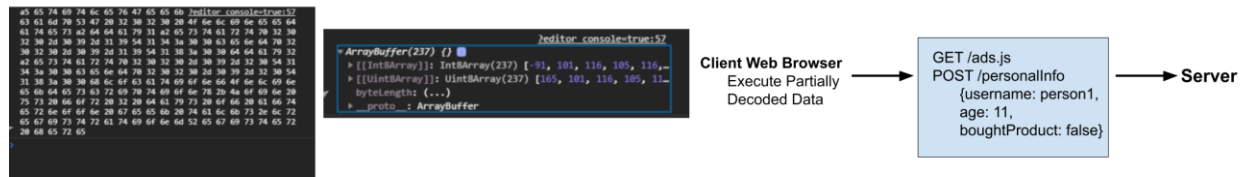


Figure 2: A simple overview of how Web Bundles execute in the browser. (Mohamed, 2020)

The compression increases the throughput of communications at the expense of additional CPU time (Willis et al.). In fact, this allows bundles to be sent through Bluetooth (which is slow) or low-bandwidth connections (Web.dev), which can then be used offline. Web bundles aim to treat web applications more like native applications while also keeping the convenience of a web application, but without the need of an internet connection.

A while back, Mozilla attempted this with the Prism project, which became known as WebRunner. This allowed web applications to be launched from the desktop and configured independently from the web browser. However, around 2010, the project went inactive, as one of the developers described that the Prism project took a lot of time to maintain and was less important to the team and the other projects at Mozilla. The time commitment caused the developers to neglect the project as they mentioned they'd, “now need significant work to turn it into a hit product” (Gertner, 2011). Moreover, the project faced deep architectural challenges due to changing technologies at Mozilla.

Description

One of the main exploitation techniques for WebBundles that don't work on web apps is to obfuscate URLs, so anyone who clicks on a page that uses a WebBundle could load malicious code from another website (Knakal, Brave). The way to do this in web apps is to use obfuscated JavaScript, which can make the code on a page unreadable, hiding what the code is doing (Katz). These relate to confidentiality, because the code is unreadable, and availability, as security auditors cannot use debug tools to ensure the loaded JavaScript is secure, and JavaScript security tools (such as EasyPrivacy (Brave)) can't detect what's happening without actually executing the code. This "can be done even without Web Bundles" (Knakal), but they make it easier to do so by having a "independent namespace", which means a set of URLs that are harder to read, as they can be obfuscated in different ways depending on the WebBundle encoder (Knakal). For example, in Figure 2 above, different parts of the URL `/ads.js` could be encoded with different major types, such as arrays (shown as `Int8Array` above), or a byte string (Bormann et al., section 3.1), and then obfuscated using something like Base64 (Katz). A simplified example of how these could be represented is shown in Figure 3 below. An index is the location of the string inside the CBOR file.

Major Types	Unsigned integer + text string	Array + text string	Array + Map (JSON object)
Representation	Index 1: 97100115 (<code>'ads'</code> encoded into ASCII base 10) Index 1000: <code>'js'</code>	Index 1: [<code>'a'</code> , <code>'d'</code> , <code>'s'</code>] Index 1000: <code>'js'</code>	Index 1: [<code>'a'</code> , <code>'d'</code> , <code>'s'</code>] Index 1000: { <code>'j'</code> : 115} (<code>'s'</code> encoded into ASCII base 10)

Figure 3: *There are many data types that CBOR could use to represent the same URL (such as `ads.js`)*

(Bormann et al., section 3.1), which could be combined.

This makes it harder to determine what the data is, as you have to figure out how to decode the URL through multiple data types. Thus, WebBundles make it harder to detect malicious URLs if the data is encoded using different data types.

As Peter Snyder discusses in his blog post on Brave, “WebBundles Harmful to Content Blocking, Security Tools, and the Open Web”, a main concern about using the Web Bundles is that they make URLs not unique. You can randomize, reuse, and hide unwanted URLs that would otherwise be blocked by ad-blocking or other privacy tools. For example, a webpage might use the following script: `<script src="http://adnetwork.com/js/ads.js"></script>`, and normally we could use an ad-blocker to block `*/ads.js`. However, when downloading a bundle you download everything within that bundle, this same file can have an arbitrary name, says “forcedAds.js”. Additionally, since a web bundle is a set of HTTP exchanges, with a URL identifying the primary resource of the bundle. Therefore, can also redirect where the URL’s since there is no enforced relationship between the URL used to look up resources in the package, and where the resource came from online (Knakal, 2020).

The main pain point of all these problems is that Web Bundles create a local independent namespace. Within web bundles, URLs are no longer global references to resources on the Web, just arbitrary indexes into the bundle. Because of this, WebBundles make it easy for sites to dodge privacy tools by randomizing URLs for unwanted resources for the web application. Now clients are no longer able to blacklist the resources that they do not want loaded when opening the web application, so therefore they’re being forced to open the web bundle and all that it includes. Peter Snyder gives a great analogy comparing web bundles to PDFs. When we download and open a PDF we have no choice of what is included in the PDF. When a client executes a web bundle, there is no way for the user to exclude a part of the package. It’s almost

impossible to access individual elements within the PDF because the PDF comes as an all or nothing package. Peter Snyder gives a great example, “What on the current Web is referred to everywhere as, say, `example.org/tracker.js`, could in one WebBundle be called `1.js`, in the next `2.js`, in the third `3.js`, etc” (Synder, 2020). All of these versions of `tracker.js` are pointing to the same resource but named differently in the included all-or-nothing bundle, so that it becomes increasingly more difficult for someone to explicitly pinpoint the resource that they want to block. In his article, Knakal raises the idea that if you can build bundles on-the-fly per request, then you really can make almost any content unblockable. Making HTTP requests on-the-fly involves you opening the web bundle, then as you're executing it, it executes the HTTP requests that are compressed in the bundle, but it makes it so that you have to decompress to inspect them, and the algorithm behind decompression doesn't allow for easy inspection without fully decompressing it.

Existing security tools don't work with WebBundles because internally, they use Concise Binary Object Representation (CBOR) to store the data that will be executed, which is as hard to reverse engineer as an executable program. According to cbor.io, CBOR is “... a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation”. Web bundles being a packaged file, are archives, and all resources in them are parsable. There are already several tools for parsing CBOR files. For example, a tool has been published on github called the Jackson Data Format Module. The Jackson, a suite of data-processing tools for Java, is a data format module that supports reading and writing CBOR encoded data. The module includes a streaming API which includes a `CBORParser` and `CBORGenerator` allowing to encode and decode the CBOR data into supported formats. This allows us to decode the entire CBOR file into individual data

types, for enhanced readability. Therefore, I would argue that inspecting a web bundle isn't difficult because we have tools that have been made available to us that can decode the CBOR data and we can analyze the data/components of individual web bundles. However, the data might be obfuscated, such as a set of binary data which could represent assembly instructions, or random JSON (Bormann et al., section 3.1). It's harder than reverse engineering JavaScript because existing deobfuscation tools won't work, so new tools would need to be made, and the barrier to entry is higher, as the types of instructions that can be put in a WebBundle aren't confined to valid JavaScript code.

Another good way to protect against attacks with arbitrary code is known as sandboxing. For the purposes of this paper, we will assume the sandbox lives in the web browser. In this technique, the sandbox is a program that prevents reading other parts of the computer, such as existing data stored in memory (such as passwords) or disk (such as documents) (Natvig). The sandbox is a "virtual machine", which means it's given a specific set of memory addresses and its own disk space (Natvig). So, if the WebBundle encoder or decoder is integrated with the browser's sandbox, it may load malicious URLs, but it can't read files or information from other programs. However, there are ways to escape sandboxes with WebBundles. For example, the WebBundle can execute arbitrary JavaScript to use an exploit against the browser sandbox. One such exploit involves Chrome's sandbox, called Caja, which took advantage of how it parses JavaScript (Heyes). But it's not hard to detect an escape: the operating system (OS) can look at the files that were changed and alert the user accordingly. Thus, sandboxing works as a security mechanism, as long as the WebBundle encoder/decoder is directly integrated into the sandbox, and the OS is not compromised.

Some existing solutions, such as ad blockers and blacklists, don't work with WebBundles. An adblocker won't work because it usually blocks certain URLs on the webpage (called a blacklist) (Knakal), such as images or scripts which are given a certain URL. The same strategy will not work with WebBundles: resources inside the WebBundle may not immediately match what's in the blacklist, as they are encoded in binary form. And when the WebBundle is decoded, the HTTP requests inside the bundle are executed, which can allow any URL from the adblocker's "ads list" to work, bypassing the adblocker (Knakal). However, while adblockers won't work, a blacklist implemented at the OS level only works if the blacklist is updated. This is because a malicious web developer could register a new domain name that the blacklist doesn't detect until the WebBundle is actually executed on someone's computer. As soon as the network request is captured on the computer's networking stack, it can be added. Both these solutions have a fatal flaw, which is that malicious URLs can be executed.

On the other hand, a whitelist (a specific set of URLs that can have traffic go through them) can be used at the OS level. This is impractical though, as users have to manually add URLs they deem non-malicious, but gives the user the most control. Existing browser extensions, such as NoScript, do this by default for JavaScript code (InformAction), but a lot of burden is placed on the users to whitelist websites. There are a few possible solutions which place the burden on different people. The first is where users could use existing whitelists like ad blockers do, but that assumes the writer of the list doesn't put malicious URLs in them. The second is where the whitelist could be applied at the network level, which also makes it harder for malware on the same computer to ignore the whitelist. However, this puts more work on network admins. A network solution also wouldn't allow users to control which URLs they deem non-malicious, if they were to build their own website, or access a legitimate network that the

network admins don't know about, because the administrator makes the whitelist based on their wants and needs. Thus, this violates the "unmotivated user" usability property, because the user has to rely on the administrator to whitelist non-malicious websites. Another problem is that a URL on the whitelist could be malicious in the future if someone were to hijack the servers hosting it. Thus, one way to achieve reasonable security could be to combine filter lists applied at the OS level and antivirus software with heuristics at the OS level. An antivirus could scan the behavior of web traffic to detect suspicious activity. This compromise places less burden on the user, retains most user control, and stops most behavior that seems malicious.

Conclusion

There is a lot of arbitrary code and attack surface in the WebBundle implementation, but about as much as JavaScript. Additional attack vectors though can be worse than using something simpler. As they say, "keep it simple". Low bandwidth is being traded off for new potential attack vectors. This is because the spec behind WebBundles trusts the encoder and decoder implementation to not be compromised. This makes it harder for cybersecurity experts to detect flaws, violating the principle of Simplicity. Also, the "confidentiality" pillar of security is being applied towards the wrong thing, which is towards encrypted URLs on the client side rather than just encrypting URLs in transit. Therefore, unless the browser properly integrates the WebBundle encoder and decoder with the browser's sandbox, users can't secure their own machines without resorting to solutions like whitelists and antiviruses. Sandboxes and antiviruses are a fail-safe stance in case something should go wrong, and whitelists can be used on the network to apply the principle of least privilege in terms of network communications. However, sandboxes are not attractive to ad companies, and so there may be pressure on web browser

© 2021 siliconninja, Thomas Nandola and Ricardo Hernandez. Paper (except for Figures 1-2) licensed under the CC-BY 4.0 license, available at <https://github.com/siliconninja/CSSE340-Web-Bundles-Research/LICENSE>.

developers to hide flaws in them to create a market of exploits. Thus, it's important to notice how WebBundle sandboxes will develop in browsers to ensure security for everyone without additional hassle. Additionally, WebBundles do create a harder way to block ads according to recent reports, which decreases privacy, as there is less user control without resorting to less usable solutions like whitelists.

Works Cited

Andone, Claudiu. “Google Makes It Harder to Block Ads and Trackers in Chromium.” *Windows Report / Error-Free Tech Life*, WindowsReport, 31 Aug. 2020, windowsreport.com/google-web-bundles/. Accessed 25 January 2021.

Barth, A. “The Web Origin Concept.” *RFC Editor*, Dec. 2011, www.rfc-editor.org/info/rfc6454. Accessed 29 January 2021.

Bormann, Carsten, and Paul Hoffman. “RFC 8949.” *RFC 8949: Concise Binary Object Representation (CBOR)*, www.rfc-editor.org/rfc/rfc8949.html. Accessed 25 January 2021.

Brinkmann, Martin. “Google Proposed Web Bundles Could Threaten the Web as We Know It - GHacks Tech News.” *GHacks Technology News*, 30 Aug. 2020, www.ghacks.net/2020/08/30/google-proposed-web-bundles-could-threaten-the-web-as-we-know-it/. Accessed 25 January 2021.

Brave. “WebBundles Harmful to Content Blocking, Security Tools, and the Open Web (Standards Updates #2).” *Brave Browser*, 26 Aug. 2020, brave.com/webbundles-harmful-to-content-blocking-security-tools-and-the-open-web/. Accessed 25 January 2021.

“FasterXML/Jackson-Dataformats-Binary.” GitHub, 30 Jan. 2021 github.com/FasterXML/jackson-dataformats-binary/tree/master/cbor. Accessed 30 January 2021.

Gertner, Matthew. “Discontinuing WebRunner.” *Salsitasoft*, 8 Sept. 2011, www.salsitasoft.com/2011/09/08/discontinuing-webrunner. Accessed 31 January 2021.

© 2021 siliconninja, Thomas Nandola and Ricardo Hernandez. Paper (except for Figures 1-2) licensed under the CC-BY 4.0 license, available at <https://github.com/siliconninja/CSSE340-Web-Bundles-Research/LICENSE>.

“Get Started with Web Bundles.” *Web.dev*, Google, web.dev/web-bundles/. Accessed 25 January 2021.

Gibb, Robert. “What Is a Web Application?” StackPath, www.blog.stackpath.com/web-application/. Accessed 30 January 2021.

Heyes, Gareth. “Escaping JavaScript Sandboxes with Parsing Issues.” *PortSwigger Research*, PortSwigger Research, 10 July 2020, portswigger.net/research/escaping-javascript-sandboxes-with-parsing-issues.

Katz, Or. “Akamai Security Intelligence & Threat Research Subscribe.” *Catch Me If You Can - JavaScript Obfuscation - Akamai Security Intelligence and Threat Research Blog*, <https://blogs.akamai.com/sitr/2020/10/catch-me-if-you-can---javascript-obfuscation.html>. Accessed 29 January 2021.

Knakal, Martin. “Web Bundles: What Are They and Do They Pose a Threat to the Web?” *CDN77*, CDN77, 5 Oct. 2020, www.cdn77.com/blog/web-bundles. Accessed 25 January 2021.

Mohammed, Isham, director. *CBOR: For Faster M2M Communication - GeekcampSG 2020*. YouTube, www.youtube.com/watch?v=QGdBTtEWEmc. Accessed 7 February 2021.

“NoScript - JavaScript/Java/Flash Blocker for a Safer Firefox Experience! - Faq - InformAction.” *NoScript*, InformAction, noscript.net/faq#qa1_5.

Natvig, Kurt. “Sandbox Technology Inside AV Scanners.” *Virus Bulletin*, Sept. 2001, vxug.fakedoma.in/archive/other/VxHeavenPdfs/Sandbox%20Technology%20Inside%20AV%20Scanners.pdf.

© 2021 siliconninja, Thomas Nandola and Ricardo Hernandez. Paper (except for Figures 1-2) licensed under the CC-BY 4.0 license, available at <https://github.com/siliconninja/CSSE340-Web-Bundles-Research/LICENSE>.

© 2021 siliconninja, Thomas Nandola and Ricardo Hernandez. Paper (except for Figures 1-2) licensed under the CC-BY 4.0 license, available at <https://github.com/siliconninja/CSSE340-Web-Bundles-Research/LICENSE>.

Sawood, Alam, et al. “Supporting Web Archiving via Web Packaging.” *Department of Computer Science, Old Dominion University*, 17 June 2019, arxiv.org/pdf/1906.07104.pdf.

Accessed 25 January 2021.

TTT Studios. “Web Bundles: The Solution to the Internet's Shrinking Attention Span.” *Medium*, Medium, 15 Nov. 2019, [tttstudios.medium.com/web-bundles-the-solution-to-the-internets-shrinking-attention-span-210d90d0f86a](https://www.tttstudios.medium.com/web-bundles-the-solution-to-the-internets-shrinking-attention-span-210d90d0f86a). Accessed 25 January 2021.

“Web Packaging Format Explainer.” *GitHub*, 8 Nov. 2019, github.com/WICG/webpackage/blob/master/explainer.md. Accessed 25 January 2021.

Wills, Craig E., et al. “Using Bundles for Web Content Delivery.” *Computer Networks*, vol. 42, no. 6, 2003, pp. 797–817., doi:10.1016/s1389-1286(03)00221-4, www.sciencedirect.com/science/article/abs/pii/S1389128603002214. Accessed 25 January 2021.

Yasskin, Jeffrey. *Use Cases and Requirements for Web Packages*, 12 Jan. 2021, wicg.github.io/webpackage/draft-yasskin-wpack-use-cases.html#name-offline-installation. Accessed 25 January 2021.