Local Storage on Steroids: Abusing Web Browsers for Hidden Content Storage and Distribution

Juan D. Parra Rodriguez and Joachim Posegga

University of Passau, Innstraße 43, Passau, Germany
{dp,jp}@sec.uni-passau.de

Abstract. Analysing security assumptions taken for the WebRTC and postMessage APIs led us to find a novel attack abusing the browsers' persistent storage capabilities. The presented attack can be executed without the website's visitor knowledge, and it requires neither browser vulnerabilities nor additional software on the browser's side. To exemplify this, we study how can an attacker use browsers to create a network for persistent storage and distribution of arbitrary data.
In our proof of concept, the total storage of the network, and therefore the space used within each browser, grows linearly with the number of origins delivering the malicious JavaScript code. Further, data transfers between browsers are not restricted by the Same Origin Policy, which allows for a unified cross-origin browser network, regardless of the origin from which the script executing the functionality is loaded from.
In the course of our work, we assess the feasibility of a real-life deployment of the network by running experiments using Linux containers and browser automation tools. Moreover, we show how security mechanisms against third-party tracking, cross-site scripting and click-jacking can diminish the attack's impact, or even prevent it.[1]

Keywords: Web Security · WebRTC · postMessage · Browser Security· Content Security Policy

1 Introduction

So far, the Web security community has invested significant efforts to research the impact of single API calls introduced by HTML5 standards on the client side. For instance, Lekies et. al. described how using local storage for content caching results in script injection, and how to prevent it [25]. Also, in the case of the postMessage API, which allows two windows to have cross-origin communication within the browser, Hanna et. al. illustrated how the lack of origin[2] validation

[1] This is an author-prepared version of the paper published here: `https://doi.org/10.1007/978-3-030-01704-0_19`

[2] two JavaScript execution contexts have the same origin only if they have the same IP or fully qualified hostname, and if they use the same protocol and port.

leads to execution of undesired functionality in real life Web sites [20]. Last but not least, Provos et. al. detected that the dynamic creation of zero pixel frames through scripts is a common attack vector used for drive-by downloads [35].

In spite of the significant efforts invested to secure each API, undesired consequences arising from client-side API combinations remain uncharted. So, we explore two particular aspects of browser APIs. On the one hand, we show that using the postMessage API, local storage, and the dynamic creation of Iframes leads to a transparent[3] increase of the total storage available for a website in the visitor's browser, i.e. beyond the storage quota. On the other hand, we show how WebRTC data channels aggravate the situation by allowing cross-origin data transfers among browsers. Thus, the combination of both factors comprises a novel attack vector in which the visitor's browser is coerced, not only to store data permanently but also to transmit such data directly to other browsers without the user's knowledge. This kind of attack could be catalogued as a browser resource abuse problem, which is orthogonal to more known Web attacks, e.g. cross-site scripting, since it does not pertain to the user's data or session.

The presented attack has *two* interesting properties. *First*, the attack relieves the server from the responsibility (and performance overhead) associated with hosting and distributing the content. This is a direct consequence of storing the content in browsers and transferring it over direct browser-to-browser links. *Second*, an attacker keeps the site's visitor oblivious to the malicious behaviour, i.e. storage and distribution of unknown content, since no warnings or messages are presented to the user. This lack of awareness on the user's side is particularly concerning when data stored in his/her browser is used for illegal purposes. Another concerning aspect is the use of computational resources in detriment of the user, e.g higher electricity costs and decreasing lifespan of a computing system, without his consent. This kind of abuse has lead to a court settlement between the state of New Jersey and a company doing Bitcoin mining on browsers to monetize Web sites [22].

Our **contributions** can be summarized as follows: *1)* we describe a novel attack whereby the persistent storage and networking capabilities of the browser are abused for the attacker's benefit, yet without requiring any additional software or vulnerability exploitation on the client's browser or operating system. *2)* we enumerate the security assumptions from the browser APIs (postMessage, Iframe creation and WebRTC) which led to the browser abuse vector. *3)* We implement a proof of concept browser network which has long-term storage capabilities, and transfers data over peer-to-peer links between browsers, and bypasses the Same Origin Policy, without making the user aware of its existence. *4)* We evaluate the proof of concept through a set of experiments by automating real-life browsers in a controlled environment while modifying the number of visitors, the time between visits, and the visitor return rate [4]. *5)* From a more constructive perspective, we discuss how existing security measures, taken by the browser's user or Web developers, can prevent the attack.

[3] the mechanism described here does not require the user's consent

[4] number of visitors who returned to the website in a given period of time

This paper is organized as follows. We describe our attack in Section 2. Section 3 and 4 describe the proof of concept implementation and its evaluation. Afterwards, we present a discussion of countermeasures in Section 5 followed by related work in Section 6. Lastly, we present our conclusions in Section 7.

2 The Attack

This section clarifies the attacker model, the benefits for the attacker, as well as the technical details exploited; however, this section is written under the assumption that users have not deployed any security mechanisms in the browser or on their sites, e.g. CSP policies. Throughout this paper, we will refer to the attacker model presented here. Later on, Section 5 presents countermeasures available today.

2.1 Attacker Model

We assume an attacker slightly less powerful than the Web attacker formalized by Akhawe et al.[5]. A Web attacker can execute JavaScript code in the victim's browser according to the browser's policies. Also, the attacker can host malicious servers which do not need to comply with Web standards. Moreover, a Web attacker can obtain valid domain names and certificates for his servers.

In our attacker model, the attacker is capable of executing a *script abusing the browser's storage, i.e.* **Abusive Script**, when a *website is intentionally opened by a visitor, i.e.* **Intended Site**. This can be achieved through an advertisement network, or script injection techniques. Further, the JavaScript context where the Abusive Script is executed, as well as its origin, are totally irrelevant for the attack. To increase the browser's storage without the user's knowledge, the attacker needs to host an Abusive Script in several origins. This can be easily achieved by using free domains; also, if the attacker owns a domain already, he could generate many sub-domains or use several ports in one domain to deliver the script [5]. The final storage space available for the attacker will be the number of origins hosting his script multiplied by the storage quota imposed by the browser. Nonetheless, unlike the Intended Site, the Abusive Script does not need to be intentionally opened by the user.

To communicate data between browsers, the attacker needs access to a server to negotiate browser-to-browser connections. Notably, this server only intervenes during the connection session establishment, but it is not used to transfer data between browsers. The attacker creating the network is slightly weaker than a Web attacker because all servers comply with Web standards, and the script context where the attacker's code is loaded is irrelevant for the attack.

2.2 Attack Details

For the sake of clarity, Figure 1 depicts the attack where three different browsers opened Intended Sites including Abusive Scripts in different ways. First of all,

[5] All are separate Origins According to RFC 6454

the figure shows Intended Sites including Abusive Scripts from two different origins, i.e. Origin1 and Origin2. Further, cross-site scripting injection (Browser 3) would allow the Abusive Script to access the JavaScript execution context of the Intended Site. On the contrary, Intended Sites are shown in Browser 1 and 2 load the Abusive Script in a different context, e.g. inside an Iframe. The latter occurs when the Abusive Script is present in an advertisement and is therefore isolated from the Intended Site context due to the Same Origin Policy. Now, we mention how to achieve the Abusive Script's execution, the irrelevance of the Same Origin Policy for the attack, how to increase the browser quota, the browser-to-browser channels, and summarize the complete attack.

Abusive Script Execution Although script injection through additional software is possible, we analyse techniques without requiring browser plug-ins, vulnerability exploitation or additional software on the client's side.

An attacker can deliver the Abusive Script through an advertisement network. This has been demonstrated by Grossman et. al. [19] and has been used to do crypto-mining without the visitors' knowledge [40]. Although in this case, the Abusive Script would be included inside an Iframe in the Intended Site, as seen in Browser 1 and 2 in Figure 1, this does not interfere with the attack. Further, it does not matter whether the advertising executing the Abusive Script is delivered through legitimate advertising networks or advertising injectors [41].

Scripts can also be included in sites by either leveraging forgotten inclusions, or by modifying popular libraries or CMS Widgets. Nikiforakis et al. [32] showed a number ways allowing to execute malicious code, e.g. using stale IPs or domains that are still included but forgotten. Particularly, the authors found out that 56 domains used in the 47 top Alexa Web sites were available for registration at the time. Also, thousands of sites were affected after by two Content Management System plugins performing crypto-mining without the user's knowledge [12, 28].

Last but not least, cross-site scripting is a particularly promising way to infect Web sites, given that by 2013 more than 6000 unique vulnerabilities were found across the Alexa top 500 Web sites (9.6% of the analysed sites) [26].

Irrelevance of the Same Origin Policy The attacker's goal is to execute the Abusive Script and abuse the local storage space and networking capabilities of the browser; hence, accessing the DOM or the JavaScript context of the Intended Site is not a prerequisite for the attacker. Thus, as it can be seen in Figure 1, the Same Origin Policy isolation between the Intended Site and the Abusive Script is not hindering the attacker.

Data can be sent to browsers which loaded the Abusive Script from any origin. Thus, cross-origin communication is allowed, not only among different Intended Sites but also between different Abusive Script origins too. This is possible because according to the security architecture proposed[6] for WebRTC dataChannels [36], enforcing the Same Origin Policy between browser-to-browser

[6] This is a IETF-draft which means this is still work in progress.

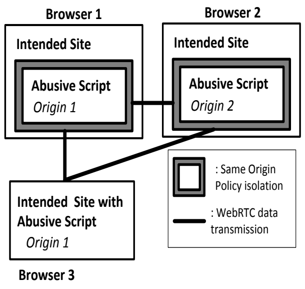

Fig. 1. Attack's Overview

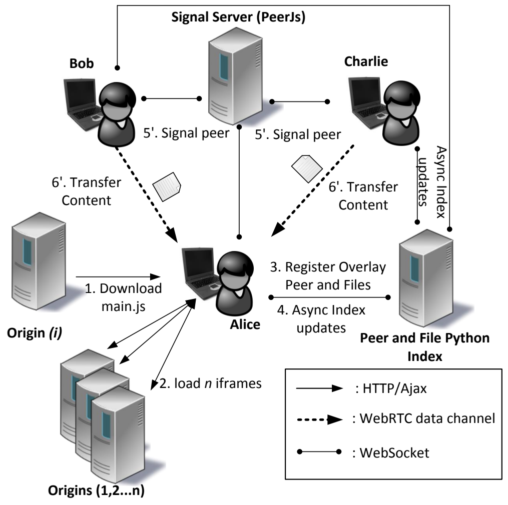

Fig. 2. Proof of Concept Diagram

channels does not provide any additional security. This design decision was based on two reasons: data channels do not inject code into other origins, and data can always be forwarded through the severs. Although these two statements are true, enabling cross-origin communication over peer-to-peer links is problematic because the direct channel empowers the developer to move data from one browser to another without the user's knowledge regardless of the origin from which the code was loaded from. What is worse, this happens without burdening the server with the data transfer. The latter is of utmost importance for the scalability of the attack since, although data could be relayed through a server, this would impose a heavy toll on the performance of the server, therefore making the proposed attack less attractive.

Increasing the Local Storage Limit From the local storage perspective, a 5 MB quota is enforced per origin, unless the user opts-in to increase it for a particular origin. The quota prevents a single origin from abusing the browser's local storage. From this point of view, letting a script create an Iframe is not problematic because, unless the Iframe and the parent window share the same origin[7], data loaded inside the Iframe (and its JavaScript execution context) is out of reach of the script creating it due to the Same Origin Policy. Although this separation of script contexts is helpful for data isolation, it can be misused to increase the local storage used on the browser.

[7] windows can also set their origin to be a super origin, i.e. mysite.company.com can set its origin as company.com to share the same origin with other pages.

The technique used to bypass the quota enforcement for a particular website uses *Iframes with different origins to store data in their local storage, i.e.* **Storage Iframes**. Given that each Storage Iframe has a different origin, each one of them has 5 MB of local storage. A Similar approach has been used to show how information can be placed within the user's browser by Feross [9, 1]. So far, this technique allows an attacker to store information in several Iframes, but how to access such information as one centralized database is not yet solved.

An attacker can solve this problem by using the PostMessage API to communicate data from several Storage Iframes controlled by him. According to the postMessage specification[30], the assumptions dictate that, as long as developers validate the origin of the messages exchanged and their proper encoding, no vulnerabilities can be exploited. The rationale behind these validations is to prevent Web sites from acting on commands sent by malicious windows and to avoid script injection. Unfortunately, as this fails to consider two origins colluding against the browser, the Abusive Script obtains a quota equivalent to the number of Storage Iframes spawned by it multiplied by the browser storage quota. In other words, postMessages are used as an asynchronous intra-browser messaging system to exchange control commands and data between the Storage Iframe and the Abusive Script, shown as "broker" in each browser in Figure 3.

Inter-Browser Cross-Origin Communication Inter-browser communication is paramount if the attacker wants to instruct browsers to share data with each other. This functionality relies on the WebRTC dataChannels [31] which requires an initial negotiation phase. Such initialization phase is solved by the implementation of the *Interactive Connectivity Establishment* protocol (ICE) [23]. In particular cases, when browsers are behind a router with *Network Address Translation* (NAT), a server providing *Session Traversal Utilities for NAT* (STUN) [38] allows them to discover their public IP address and port. In most cases a short intervention of a STUN server is enough to enable browsers to communicate with each other directly. The previous protocols are covered by a server accessible by the attacker, as mentioned in Section 2.1. Nevertheless, in some cases, it may be impossible to establish a direct connection between two peers who are behind two different NAT routers. Then, an additional relay server implementing the *Traversal Using Relay NAT* protocol (TURN) [27] is needed for the communication.

Putting it All Together In order to put together an attack in which data stored in a browser is available across the whole cross-origin browser network, the attacker needs to extend the increase of the Local Storage use with browser-to-browser connectivity. As a result, each Storage Iframe hosts an **overlay peer**, i.e. *a WebRTC enabled frame*. Also, the Storage IFrame needs to receive control commands, through postMessage API, not only to share data from Local Storage but also to connect to other peers, retrieve and send data from them, etc. Figure 3 reflects an example in which two browsers visit one origin each,

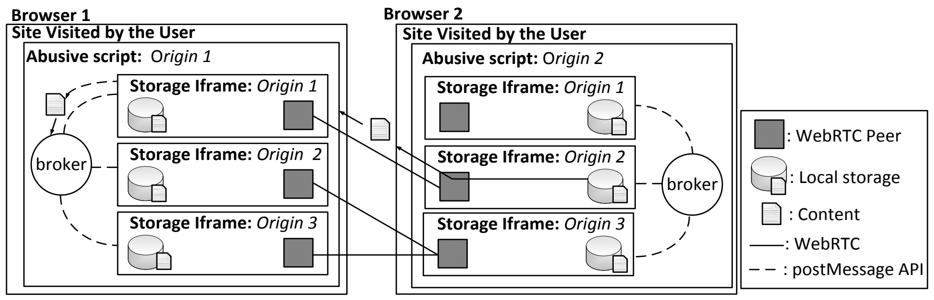

Fig. 3. Attack Schematics

where the Abusive Script is hosted. Further, this figure shows the Storage Iframe hosted on three different origins, i.e. Origin1, Origin2, Origin3.

3 Proof of Concept

We have built a proof of concept where every browser opening a website containing an Abusive Script replicates files present in a unified browser network. In our implementation there is no central server hosting the files; instead, every browser can register files in the network and they will be automatically replicated by other browsers. Further, every browser spawns several storage frames, i.e. 10 Origins equivalent to 50 MB in our case, and attempts to replicate as many files as possible. The replication process stops when every file in the network is replicated locally, or when there is no space left in any Storage Iframe. Also, content transfers happen over browser-to-browser WebRTC connections.

Although the mapping between peers and files in the network could have been distributed across the browser network, e.g. using a Distributed Hash Table [14], this neither strengthens nor weakens our argumentation on the security issues raised by the attack. Likewise, our prototypical implementation requires files to have at most 5 MB when they are encoded in base 64. Also, we have tested our implementation with Chrome 43.0.2357.81, and Firefox 38.0.

Based on Figure 2, the remainder of this section describes the components and the message exchange in our proof of concept. The **components** are:

Abusive Script: Our "broker" uses the postMessage API adopting a hierarchical approach where the Abusive Script commands each Storage Iframe to execute actions, e.g. retrieve a file from another peer, and receives callbacks with the status of the task. This provisions the Abusive Script with an overview of files stored locally, and ensures files are replicated at most once per browser.

Signalling Server: We used a local installation of the PeerJS Server. This open-source server, in combination with the Peer client library, provides a high-level API allowing to send signals to peers in the network and to establish WebRTC channels among them.

Peer and File Index: this is a Python server used to track which files are stored in which peer as well as which peers are currently in the overlay network.

Whenever a browser joins the network a WebSocket is opened to this server. We used WebSockets to receive and broadcast notifications from and to the peers when the index has changed, etc. Also, when a WebSocket connection is closed the server can safely assume that a given browser (and all its Overlay Peers) left the network. Also, for visualization purposes, this server offers a simple HTML page to upload files to a Storage Iframe, retrieve files from other browsers, and query an updated index of peers and files.

The **message exchange** between the different components mentioned before is depicted by Figure 2. In this set-up, we show how Bob and Charlie have already joined the network; thus, they are already registered in the *Peer and File Index*, and they already stored locally some files required by the peers in Alice's browser.

The first step corresponds to when Alice opens an Intended Site containing an Abusive Script. The second step takes place when the Abusive Script generates n different invisible Storage Iframes, where n is the number of origins serving the code. The registration of the Storage Iframe as a peer in the network is shown in the third step. At this point, in the fourth step, the Abusive Script will command each Storage Iframe to download files from several peers. Once a Storage Iframe, inside Alice's browser, receives a command from the Abusive Script requiring the acquisition of a file from a specific peer, it will start the transfer between browsers. This process starts when Alice's browser uses the PeerJS implementation to negotiate the connection details to establish the WebRTC channel with the specific Iframe in Bob's and Charlie's browsers [8]. Once the signalling process succeeds, a direct connection between Alice and Bob, and another one between Alice and Charlie can be established, so the content can be transferred directly. Once Alice receives a new file, her browser will communicate this to the *Peer and File Index* server. Also, arrows labelled as *Async index updates* in Figure 2 show the WebSockets asynchronous full-duplex updates between peers and the index server.

Each Abusive Script follows a simple replication approach. When there is space in a Storage Iframe, the Abusive Script instructs a Storage Iframe to replicate the file with the least amount replications in the network that has not been stored in the browser. This guarantees that when nodes leave and files are being less replicated, they are copied to other nodes before they perish.

4 Evaluation

We do not pretend to cover an extensive performance evaluation of the proof of concept. Instead, we merely want to establish a set of conditions under which the attack works and argue for its plausibility in a real-world deployment. Thus, there are two concerns that we need to address. First of all, the browser network should keep files available in spite of the high churn produced by browsers joining and leaving the Intended Site. Also, network overhead imposed on servers, e.g. the signalling server, should be negligible compared to the network use on the

[8] Steps 5 and 6 are denoted with an apostrophe to represent that they are executed in parallel

browser's side. This would guarantee that the network can scale without requiring high computational resources from the attacker. To this end, we collect log files and network traffic from several experiments. The main goal is to calculate how long is a file available in the network during an experiment run and also to assess the network load on the servers and browsers forming the browser network. Moreover, every component was restarted between experiments to ensure that sequential runs do not interfere with each other.

4.1 Set-up

We have used Docker [15] Linux containers to ensure that tests have exactly the same initial state (docker image). As shown by Figure 4, we used docker containers to execute the so-called *selenium controller*. The controller is a custom-made multi-threaded Java application providing a REST API. This application receives commands, including actions such as open a website, close the window, or wait a certain time before the next instruction, through HTTP. These actions are executed on a Chromium browser inside the docker instance through a Selenium driver [39]. To run a headless Chromium browser, we used Xvfb as an X server to simulate a terminal without using hardware for it.

Having a generic selenium client proved to be very useful to execute several tests without re-building the containers for every test case. In addition to the containers for the selenium controller, an apache2 (hosting the Intended Site, the Abusive Script, and the Storage Frames), a Peer and Index server, as well as a PeerJS server were run in separate containers, in the same host machine.

On the bottom of Figure 4, the *orchestrator* represents a Python program sending actions to every selenium controller used for the experiment. This is a multi-threaded Python application implementing an HTTP server to receive callbacks from the selenium controllers, once they have finished a task. The Orchestrator implements the waiting times between browser visits and specifies which Chromium profile should be used for the browser session to be opened from the selenium controller. Specifying a certain profile empowers the Orchestrator to ensure that elements stored in the local storage for the given profile are available in the browser session executed by Selenium. For example, if the orchestrator wants to simulate a visitor that comes for the first time to a website, a clean profile without any cookies, local storage items, or any other previous information is used. Conversely, loading a Selenium session with a specific profile, which has already been used by a browser session which visited the network's site, would contain all the stored files in local storage and is therefore used to represent a returning visitor. The profiles are represented as folders in the case of Chromium and Chrome. Moreover, the host machine used was a Lenovo T430S with 16 GB RAM memory and an Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz processor with Ubuntu 12.04 LTS.

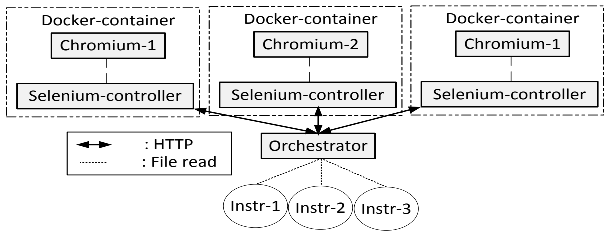

Fig. 4. Overall Measurement Set-up for 3 browsers

4.2 Browsers' Behaviour

A selenium controller has the possibility to do one-time visits, or a returning visit depending on the profile used, see Section 4.1. Therefore, we generate instructions to simulate returning and non-returning visits. We divide the set of browsers into two sets accordingly. In this way, a returning controller will always return with its previous state during the whole experiment. On the contrary, a selenium controller doing visits equivalent to a one-time visit also returns to the Web site following the same pattern, but it loads a fresh profile every time. Since the latter kind of selenium controller represents a one-time, or "non-returning" visitor, it is also called non-returning selenium controller (or browser) from now on. For each returning or non-returning selenium controller, the process to generate the ***visit length***, i.e. time in which the browser keeps the Intended Site open, and the ***time between visits***, i.e. time until the browser comes back, is generated using a random number generator, see Figure 5. Thus, the time of the experiment is filled with sequences of visits followed by waiting times between visits. The visit length is depicted in the grey-shaded areas for each browser, while the time between visits is represented by white sections.

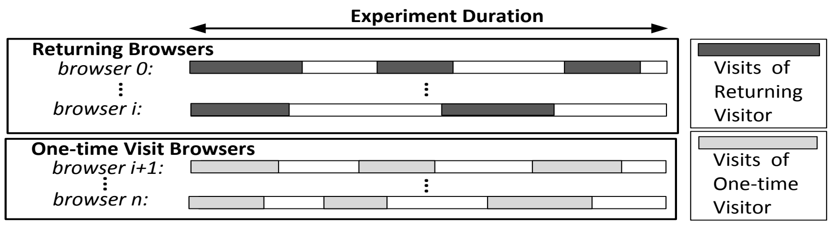

Fig. 5. Visits Simulation

4.3 Measurements

Figure 6 shows the data sources required for our evaluation in grey-shaded boxes. The data sources were: a network (tcpdump) capture including all the traffic during the experiment, and the log files where the peer and index server counts the number of replications per file, i.e. a simple array.

The content hosted by the network is comprised of 33 pictures with an average size of 1 MB each, i.e. a total of 33 MB. This size ensures that 33 MB can be stored in one browser (using up to 50MB of Local Storage) once they have been encoded in base 64. Although exploring how the network reacts when not all files can be stored in one browser would be interesting, we omit this analysis because the performance of the browser network is not our primary goal.

The ***visit length*** for every visit in the experiments has been randomly generated in a range from 30 to 50 seconds using NumPy [33] random generator. We consider this number to be conservative since there are marketing reports showing average sessions across countries higher than 50 seconds for every kind of website category [13]. Further, research has reported Web sessions to have a mean value of 74 minutes [7]; also, it is known that certain pages such as Facebook, have users with sessions ranging from a few to several tens of minutes [8]. The duration of every experiment is 5 minutes.

As mentioned in Section 4.1, returning visits are achieved by instructing a selenium controller to load a Chromium profile containing information from a previous visit. Moreover, to have files in the browser network, each selenium controller acting as a returning client has a profile containing its initial state. Therein lie all the files to be replicated in the browser network. This profile is copied to the docker instance at the beginning of every experiment in order to keep a consistent initial state across the different runs of the tests. Browsers acting as first visitors don't use these profiles and have no information in local storage, cookies, or browsing cache.

We vary two parameters during our experiments, namely the time between visits, and the number of selenium controllers returning to the website, i.e. using a Chrome profile containing data from their previous visits. Further, the ***time between visits*** is generated randomly within the ranges [10-40], [110-140] and [210-240] seconds. Note that *even though we use returning browsers with relatively short periods of time, a single browser return can represent a different user but with the same local storage state; or in other words, there is no one-to-one mapping between real users and browsers.* The number of returning selenium controllers has also been modified to be 3, 5 and 7 out of 10 browsers for each set of experiments, which yields a 30, 50 and 70% visitor return rate.

In the upcoming sections, we focus on the two critical aspects under evaluation: the file availability of the network, and the network load imposed on the browsers and servers.

File Availability The analysis of the index file, generated by the *Peer and File Index* server consisted on verifying the timestamps and state of the index to calculate the percentage of the time for the experiment run in which each file was available. Then, the average value and standard deviation for the array of percentages was calculated using Python NumPy [33].

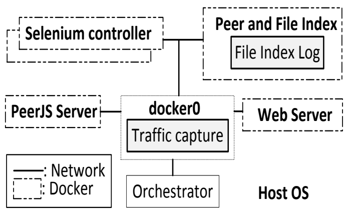

Fig. 6. Data Collection Set-Up

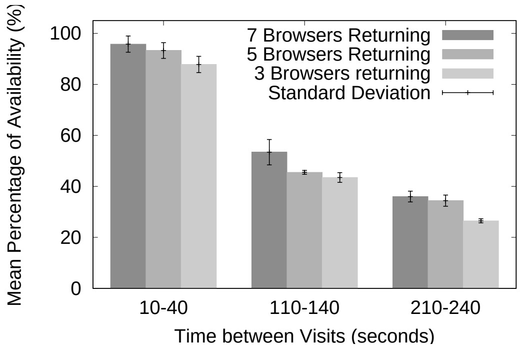

Fig. 7. Attack Evaluation with *visit lengths* between 30-50 sec (5 min. experiments)

As shown in Figure 7, the availability is strongly influenced by the time between visits; on the contrary, it is noticeable that the percentage of returning visits impacts to a lesser extent. With the shortest time between visits (10-40 seconds), the mean availability for the files is 95.7%, 93.2%, and 87.8% for 70%, 50% and 30% of return rate respectively; furthermore, in all the cases the standard deviation lies between 3.0% and 3.1%.

We can safely conclude that when 3 out of 10 browsers are controlled by the returning selenium controller, there is a 30% visitor return rate. This can be directly extrapolated to visitor return rate calculated for Web sites per month, or per day without any loss of generality. Moreover, considering that a recent marketing report [13] states that return visitor rates commonly lie between 25 and 52%, achieving a visitor rate of 30% for an Intended Site is realistic from the returning visitor perspective.

Further, regarding the comeback rate our browser network has two advantages. The first advantage in favour of the attacker is that he does not need to ensure a high return rate for every Origin used by the network, e.g. Origins used to store the Storage Iframes. As long as an Intended Site is visited, the Abusive Script will spawn invisible frames which can point to any domain without the user's knowledge. The second advantage is that, although a 30% return visitor ratio is feasible to achieve, the requirements for the browser network are less restrictive. The attacker could place the Abusive Script in several Intended Sites, such that whenever they are visited, they spawn n Storage Frames owned by the attacker. Since the Same Origin Policy is not affecting our network, the browser will always join the same network, i.e. returning to it, in spite of visiting a different Intended Site, or even when the Abusive Script is from a different Origin. Therefore, the return rate required for the attack is not that of a single Intended Site, but rather the return rate of all the Intended Sites serving the Abusive Scripts combined.

Given that we have already covered the visit length and the visitor return rate, it is key to assess whether the concurrent sessions opened by browsers during our experiment is feasible in real-world Web sites. To this end, we do an approximate estimation of this based on average values. First of all, we calculate the average number of visits per browser as the duration of the experiment divided by the sum of the average time of a visit and the average waiting time between visits. This yields a total of 8.75 visits per browser with the shortest wait between visits (10-40 seconds). Thus, it follows that for 10 browsers, we have 87.5 visits every 5 minutes (the length of the experiment). Assuming a uniform distribution of visits and using the pigeonhole principle this value could be extrapolated to 176.400 visitors per week. This number seems to be acceptable, given that currently the top 500$^{\text{th}}$ site according to Alexa's ranking [6] has 78 Million visits per month, and research has shown that even several years ago more than 20% of typical commercial sites had more than 10.000 browser clients concurrently connected, and from 4 to 10% of randomly selected sites would be able to host more than 1000 concurrent nodes [7].

Like with the previous observation, placing the Abusive Script in several origins allows the attacker to increase the number of visitors to the browser network since it is not covered by the Same Origin Policy. This increases the chances of the applicability of the attack.

To summarize, we can extrapolate the effectiveness of the presented attack when the following assumptions are met. First of all, every file can be stored in one browser, i.e. the attacker has deployed JavaScript code in sufficient domains. Second, the attacker is capable of placing Abusive Script in at least one domain achieving a return rate of at least 30% for all domains combined. Third, Web sites' visitors have sessions in the range between 30 and 50 seconds.

Network Analysis Raw network traffic has been collected from every experiment. The raw capture file, containing all the bytes exchanged between entities of the browser network, was processed after the experiment has finished by a Python script using the dpkt [16] package to count the bytes aggregated by source and destination IP. We use this information to analyse properties of the browser network. For readability reasons, the information is not shown on a per-entity basis, but instead we focus on interaction between three groups of entities: the group of returning browsers, the group of browsers executing the one-time visits, and the group of servers including the index and peer server, the Web server, and the PeerJs server. The nature of the network analysis requires representing the network traffic for each experiment run individually. Due to the similarity between network captures, we chose one experiment to analyse the traffic, i.e. time between visits in [10-40] with 5 selenium controllers returning. In Figure 8 we depict the average amount of data (in MB) transmitted between the group represented by the row of the matrix to the group represented by the column of the matrix; also, darker colours represent less amount of data. Based on this, it is observed that browsers executed by selenium clients send a very small amount of data to servers. It is also clear that browsers exchange the highest amount of data in the browser network, as expected. Another interesting fact is that returning browsers send more data to non-returning browsers than returning browsers, this happens because non-returning browsers have a clean local storage every time they join, and therefore attempt to replicate files constantly. Due to HTTP Headers, static content must not be retrieved again (when it has not changed). This is clearly observable because returning browsers send and receive fewer data to/from servers in comparison to browsers controlled by non-returning selenium controllers. Last but not least, returning browsers send a considerable amount of bytes to non-returning browsers, which is not reciprocal. Figure 8 shows that non-returning browsers receive 23.39 (18.9 + 4.49) MB from returning and non-returning browsers on average. Moreover, non-returning browsers deliver 6.97 (2.48 + 4.49) MB to returning and non-returning browsers in average. Nonetheless, the fact that they deliver almost 5 (out of 6.97) MB to other non-returning browsers, is a sign of their contribution towards keeping files replicated.

	returning selenium	**non-returning selenium**	**servers**
returning selenium	15.57	18.90	0.13
non-returning selenium	2.48	4.49	0.22
servers	0.55	1.17	0.00

Fig. 8. Average data (in MB) transmitted with 5 returning selenium controllers - time between visits in [10-40] seconds

5 Countermeasures

In this section, we cover how security-aware Web developers and browser's users can employ third-party tracking protection and Content Security Policy (CSP) directives available today to thwart the attack. Also, the countermeasure discussion is continued by analysing relevant proposals for CSP that would help against the attack but have not been adopted yet.

Third-Party Tracking: Previous research has shown that Internet users are constantly under surveillance when sites include third-party functionality on the Web [17]. Thus, browser vendors let users prevent third-party sites from tracking them [18, 11], i.e. use cookies or any other permanent storage mechanism. This implies that users can prevent the Abusive Script in Figure 1 to use their local storage because it is a third-party site included by the Intended Site.

CSP: The Content Security Policy (CSP) specification is a tool for developers and Web masters to restrict functionality and limit privileges of resources loaded from their sites, through headers in the HTTP response. Restrictions include, but are not limited to whitelisting sources from which content or scripts can be loaded, which resources can execute scripts, or whether their environment should be sandboxed. It must be noted, that CSP is not meant to supersede proper output encoding and input validation, but it offers a second line of defense implemented by browsers when a Web application has been compromised.

CSP contains a `sandbox` directive offering the same functionality offered by the HTML5 attribute under the same name for Iframes [43]. Both mechanisms would ensure that an Iframe cannot execute JavaScript unless the `allow-scripts` used. And even if the `allow-scripts` keyword is used, sandboxed Iframes are assigned to a random origin making all same-origin checks fail, which in turn does not allow them to use Local Storage or cookies.

To prevent click-jacking, a developer can use the CSP `frame-ancestors` keyword to ensure that a particular site can only be embedded in resources loaded from a list of origins. If a security-aware Web master specifies a restrictive list of frame ancestors for his site, this would prevent an attacker who has compromised the site from including this particular site as a Storage Frame in the Abusive Script. In more practical terms, this means that an attacker injecting the Storage Frame code in *Origin1* depicted in Figure 1, cannot include *Origin1* in his Abusive Frame due to the frame ancestors list. However, if an attacker would host the Storage Frame on his own server, the attack would still work.

The `script-src` CSP directive specifies which scripts can be executed from a particular site. Thus, with a restrictive policy allowing to include only secure scripts, which cannot be compromised by the attacker, it becomes impossible for the attacker to execute his Abusive Script or the Storage Frame functionality. In practice, this mechanism has faced several challenges, i.e. it has been already shown that 94% of all policies deployed with CSP can be bypassed due to unsafe exceptions [44]; however, the authors also proposed a new keyword, i.e. `strict-dynamic` which is part of the current CSP draft, to ease the definition of CSP script source policies.

Pending CSP Proposals: Now we cover CSP extensions limiting the studied attack which have been proposed but are either not implemented, or have been discussed but are not included in CSP yet.

Hanna et. al have shown that developers tend to forget place proper origin validations when there are scripts collaborating and exchanging messages over the PostMessage API [20]. In 2011, one of the authors proposed to address this issue by providing a declarative way to specify which sites can interact with other origins (whitelist) as part of CSP, and this has been discussed over the Web security standardization mailing list already [3]. Recently (5 years after the initial discussion), a new issue has been created to decide where and how enforcement on PostMessages would be meaningful for existent Web applications [4]. Although this discussion revolves around CSP3, PostMessage API enforcement has not been included yet. If a Web master or developer would be able to specify with which origins can a Web application interact with using PostMessages, the mechanisms to increase the Local Storage limit could be hindered from distributing and serving all the content over the broker shown in Figure 3.

Early warnings pointing out that WebRTC can be used for data exfiltration are visible as an issue for CSP created in 2014 [42, 46]. Later, certain sites started abusing the WebRTC API to transfer data without the user's knowledge or control. Thus, there is a new thread for discussion on the latest CSP specification [24], still open, but created 2 years after the initial issue. If users would be able to restrict with which origins can a site communicate using WebRTC data channels, the cross-origin feature provided by the invisible DataStore would be removed from the attack. However, this feature is not part of the CSP3 [45] draft.

6 Related Work

Using Local Storage to store information on the client without the user's knowledge has been introduced by Bogaard et. al [10]. Their work focused on placing a single file on a Web server and distributing pieces of this file to several browsers. Then, the Web developer would deploy a different application to retrieve the content to the server again. The attack studied shares the motivation to keep the user uninformed, but it neither builds a browser network nor circumvents Local Storage quotas through PostMessages. From the storage abuse perspective, Feross discovered that a single website could instruct Local Storage to store data in many subdomains. This lead to abuse the users' disk, filling it until the browser crashes or the whole disk is occupied [9, 1]. This relates to our quota bypass mechanism as both rely on using different origins to increase the quota. However, we have enhanced this approach to make the data accessible to the Abusive Script, by implementing letting several origins collaborate through an asynchronous message channel, the broker shown in Figure 3 implemented through the postMessage API. There have been previous browser networks using WebRTC to deliver static content. For example, PeerCDN [21] is a WebRTC-based Content Distribution Network (CDN) using the visitor's browser to share the website's static HTML content with other browsers. Owners of the company claim to achieve a 90% bandwidth reduction for the server hosting the site. Zhang et. al implemented another browser-based CDN called Maygh [47]. Maygh relies not only on WebRTC, but also on Real Time Media Flow Protocol (RTMFP), i.e. a closed source protocol accessible from Flash plug-ins. The authors examined the performance and the applicability of the CDN network by conducting experiments where simulated browsers would visit the website using the CDN. They conclude a reduction of 75% on bandwidth use on the operator of the website's side. Further, to avoid abusing the clients, the CDN network ensures that users do not upload more than 10 MB to the CDN. From a slightly different perspective, there is research work to transmit video streams between browsers using WebRTC [29, 34, 37] to ease the burden imposed on servers hosting the video streams. And there is a tool designed to implement a similar protocol to Torrent within browsers called WebTorrent [2].

Although these three approaches execute JavaScript code to distribute content, there are important differences between the previously mentioned approaches and ours. First of all, content and the video distribution networks do not use the browsers as a storage system to put and retrieve information unrelated to what they are consuming. Instead, these approaches replicate the content matching what is being rendered to the visitor of the website. Also, these content distribution networks, do not collude against the user bypassing the storage restriction as the attack described here does. Also, they do not leverage data channels across different origins, which is part of the attack presented.

7 Conclusion

Cross-window and browser-to-browser communication channels provided by the postMessage API WebRTC, respectively, bring more flexibility to Web developers; however, adding new communication channels to an already highly complex security model is problematic. Specifically, we show that despite extensive research on new APIs added to the browser [20, 25], there are combinations of browser APIs posing threats to browsers. An attacker serving malicious code, e.g. through an advertisement network, can access persistent storage mechanisms in browsers beyond the intended quota per site. Furthermore, circumventing the local storage enforcement can be combined with coercing the visitor's browser to communicate stored data through browser-to-browser links, even when the site's origins of sites loaded by browsers differ. Thus, an attacker can create a browser network for data storage and distribution in a hidden manner. As discussed in Section 2.1, the attacker requires neither access to the DOM nor access to the JavaScript execution context of the compromised website, i.e. Intended Site.

The attack we presented has several key differences with respect to "common" Web attack scenarios. On the one hand, the attacker abuses the resources available to the browser instead of targeting a Web application, e.g. to compromise the user's credentials. On the other hand, the attack presented here goes beyond a single misbehaving script. Instead, several colluding scripts are loaded by the initial Abusive script. This goes against some of the typical assumptions of the current Web security model and is visible in three dimensions: Iframe isolation, cross-window communication, and cross-browser communication. The issue with Iframes pertains to the local storage origin-based isolation, which is useful for data access control but enables the storage quota explosion. For cross-window communication, the PostMessage assumes that a window should protect itself against other rogue windows sending malicious messages. This fails to consider two malicious windows cooperating to abuse the browser (instead of attacking the window receiving messages). A similar principle applies to cross-browser communication. Although it seems that browser-to-browser channels do not pose a threat to the user as they cannot steal information from other JavaScript contexts, they can be used to create an overlay network of browsers to host potentially malicious information. Aside from saving computation resources, an attacker can force browsers to store information used for criminal activities, while avoiding the risk associated with hosting and distributing the information himself. In other words, an attacker can complicate forensic analysis greatly by distributing his malicious information across browsers, yet being able to retrieve it when needed.

Although resource abuse cases have not been included in the security model, we also show how mechanisms initially intended against click-jacking, third-party tracking and cross-site scripting can be used to prevent the attack. With this paper, we expect to raise awareness about resource abuse through browsers to ensure that existing countermeasures stay in place.

Acknowledgements

This research has been supported by the EU under the H2020 AGILE (Adaptive Gateways for dIverse muLtiple Environments), grant agreement number H2020-688088.

References

1. Aboukhadijeh, F.: The Joys of HTML5: Introducing the new HTML5 Hard Disk Filler API. `www.filldisk.com/`, accessed: 2018-04-15
2. Aboukhadijeh, F.: Webtorrent. `https://github.com/feross/webtorrent` (2014), accessed: 2018-04-15
3. Akhawe, D.: CSP and PostMessage. `https://lists.w3.org/Archives/Public/public-web-security/2011Dec/0020.html`, accessed: 2018-04-15
4. Akhawe, D.: do we want a directive to control postMessage explicit channels outbound? . `https://lists.w3.org/Archives/Public/public-web-security/2011Dec/0020.html`, accessed: 2018-04-15
5. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a Formal Foundation of Web Security. In: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium. pp. 290–304. CSF '10, IEEE Computer Society, Washington, DC, USA (2010). https://doi.org/10.1109/CSF.2010.27
6. Alexa Traffic Ranking and visitor statistics for 7 years. `http://www.rank2traffic.com/`, accessed: 2018-04-15
7. Antonatos, S., Akritidis, P., Lam, V.T., Anagnostakis, K.G.: Puppetnets: Misusing Web Browsers As a Distributed Attack Infrastructure. ACM Trans. Inf. Syst. Secur. **12**(2) (2008)
8. Athanasopoulos, E., Makridakis, A., Antonatos, S., Antoniades, D., Ioannidis, S., Anagnostakis, K.G., Markatos, E.P.: Antisocial Networks: Turning a Social Network into a Botnet. In: Proceedings of the 11th International Conference on Information Security. pp. 146–160. ISC '08, Springer-Verlag, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85886-7_10
9. Web Code Weakness allows Data Dump on PCs. `http://www.bbc.com/news/technology-21628622` (2008), accessed: 2018-04-15
10. Bogaard, D., Johnson, D., Parody, R.: Browser web storage vulnerability investigation html5 localstorage object. In: Proceedings of the International Conference on Security and Management. pp. 1–7 (07 2012)
11. Clear, enable, and manage cookies in Chrome. `https://support.google.com/chrome/answer/95647`, accessed: 2018-04-15
12. Cimpanu, C.: Cryptojacking Script Found in Live Help Widget, Impacts Around 1,500 Sites. `https://www.bleepingcomputer.com/news/security/cryptojacking-script-found-in-live-help-widget-impacts-around-1-500-sites/`, accessed: 2017-11-25
13. Clicktale: Web-Aanalytics Benchmark Q2 2013. `https://research.clicktale.com/web_analytics_benchmarks.html`, accessed: 2018-04-15

14. Dias, D.: WebRTC Explorer. https://github.com/diasdavid/webrtc-explorer, accessed: 2018-04-15
15. Docker. https://www.docker.com/, accessed: 2018-04-15
16. Dpkt package. https://pypi.python.org/pypi/dpkt, accessed: 2018-04-15
17. Englehardt, S., Reisman, D., Eubank, C., Zimmerman, P., Mayer, J., Narayanan, A., Felten, E.W.: Cookies that give you away: The surveillance implications of web tracking. In: Proceedings of the 24th International Conference on World Wide Web. pp. 289–299. WWW '15, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland (2015). https://doi.org/10.1145/2736277.2741679
18. Disable third-party cookies in Firefox to stop some types of tracking by advertisers. https://support.mozilla.org/en-US/kb/disable-third-party-cookies, accessed: 2018-04-15
19. Grossman, J., Johansen, M.: Million Browser Botnet. https://www.blackhat.com/us-13/briefings.html, accessed: 2018-01-15
20. Hanna, S., Shin, E.C.R., Akhawe, D., Boehm, A., Saxena, P., Song, D.: The emperor's new APIs: On the (in) secure usage of new client-side primitives. In: Workshop on Web 2.0 Security and Privacy (W2SP) (2010)
21. Hiesey, J., Aboukhadijeh, F., Rajah, A.: PeerCDN. https://peercdn.com/ (2013), accessed: 2018-04-15
22. Hoffman, J.J.: New Jersey Division of Consumer Affairs Obtains Settlement with Developer of Bitcoin-Mining Software Found to Have Accessed New Jersey Computers Without Users' Knowledge or Consent. http://www.njconsumeraffairs.gov/News/Pages/05262015.aspx, accessed: 2018-04-15
23. J., R.: Rfc5245: Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. RFC 5245 (April 2010), https://tools.ietf.org/html/rfc5245, accessed: 2018-04-15
24. Kesteren, A.v.: WebRTC RTCDataChannel can be used for exfiltration. https://github.com/w3c/webappsec-csp/issues/92, accessed: 2018-04-15
25. Lekies, S., Johns, M.: Lightweight Integrity Protection for Web Storage-driven Content Caching. In: Workshop on Web 2.0 Security and Privacy (W2SP) (2012)
26. Lekies, S., Stock, B., Johns, M.: 25 million flows later: Large-scale detection of dom-based xss. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. pp. 1193–1204. CCS '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2508859.2516703
27. Mahy, R., P., M.: Rfc5766: Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat(stun). RFC 5766, IETF (April 2010), https://tools.ietf.org/html/rfc5766
28. Maunder, M.: Wordpress plugin banned for crypto mining. https://www.wordfence.com/blog/2017/11/wordpress-plugin-banned-crypto-mining/, accessed: 2018-01-15
29. Meyn, A.J.R., Nurminen, J.K., Probst, C.W.: Browser to Browser Media Streaming with HTML5. Master's thesis, Aalto University (2012), https://aaltodoc.aalto.fi/handle/123456789/6094
30. Mozilla Developer Network(MDN) - Window.postMessage(). https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage (April 2015), accessed: 2018-04-15
31. Narayanan, A., Jennings, C., Bergkvist, A., Burnett: WebRTC 1.0: Real-time Communication Between Browsers. W3C working draft, W3C (Sep 2013), http://www.w3.org/TR/2013/WD-webrtc-20130910/

32. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You are what you include: Large-scale evaluation of remote javascript inclusions. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 736–747. CCS '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2382196.2382274
33. NumPy. `http://www.numpy.org/`, accessed: 2018-04-15
34. Nurminen, J., Meyn, A., Jalonen, E., Raivio, Y., Garcia Marrero, R.: P2p media streaming with html5 and webrtc. In: Computer Communications Workshops (INFOCOM WKSHPS) 2013 IEEE Conference on. pp. 63–64 (April 2013). https://doi.org/10.1109/INFCOMW.2013.6970739
35. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All your iframes point to us. In: Proceedings of the 17th Conference on Security Symposium. pp. 1–15. SS'08, USENIX Association, Berkeley, CA, USA (2008)
36. Rescorla, E.: ietf-draft: WebRTC Security Architecture. `https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-11` (March 2015), accessed: 2018-04-15
37. Rhinow, F. and Veloso, P.P. and Puyelo, C. and Barrett, S. and Nuallain, E.O.: P2P live video streaming in WebRTC. In: Computer Applications and Information Systems (WCCAIS), 2014 World Congress on. pp. 1–6 (Jan 2014). https://doi.org/10.1109/WCCAIS.2014.6916588
38. Rosenberg, J., Mahy, R., Matthews, P., D., W.: Rfc5389: Session traversal utilities for nat (stun). RFC 5389, RFC Editor (October 2008), `https://tools.ietf.org/html/rfc5389`
39. SeleniumHQ: Browser Automation. `http://www.seleniumhq.org/`, accessed: 2018-04-15
40. Telegraph, T.: YouTube shuts down hidden cryptojacking adverts . `http://www.telegraph.co.uk/technology/2018/01/29/youtube-shuts-hidden-crypto-jacking-adverts/`, accessed: 2018-01-15
41. Thomas, K., Bursztein, E., Grier, C., Ho, G., Jagpal, N., Kapravelos, A., Mccoy, D., Nappa, A., Paxson, V., Pearce, P., Provos, N., Rajab, M.A.: Ad injection at scale: Assessing deceptive advertisement modifications. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy. pp. 151–167. SP '15, IEEE Computer Society, Washington, DC, USA (2015). https://doi.org/10.1109/SP.2015.17
42. Thomson, M.: CSP for WebRTC. `https://lists.w3.org/Archives/Public/public-webappsec/2014Aug/0162.html`, accessed: 2018-04-15
43. W3CScools: HTML Iframe sandbox Attribute. `https://www.w3schools.com/tags/att_iframe_sandbox.asp`, accessed: 2018-04-15
44. Weichselbaum, L., Spagnuolo, M., Lekies, S., Janc, A.: CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1376–1387. CCS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978363
45. West, M.: Content Security Policy Level 3. `https://www.w3.org/TR/2016/WD-CSP3-20160913/`, accessed: 2018-04-15
46. West, M.: WebRTC via 'connect-src'? `https://www.w3.org/2011/webappsec/track/issues/67`, accessed: 2018-04-15
47. Zhang, L., Zhou, F., Mislove, A., Sundaram, R.: Maygh: Building a cdn from client web browsers. In: Proceedings of the 8th ACM European Conference on Computer Systems. pp. 281–294. EuroSys '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2465351.2465379