Abusing Web Browsers for Hidden Content Storage and Distribution

Juan David Parra Rodriguez and Joachim Posegga

Institute of IT-Security and Security-Law (ISL), University of Passau
{dp,jp}@sec.uni-passau.de

Technical Report, Number MIP-1603
Department of Informatics and Mathematics
University of Passau, Germany
September 2016

Abstract

An existent gap in the underlying 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 the power of the attack, we 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, browser automation tools, and custom-made software. Moreover, we lay the groundwork towards possible countermeasures and illustrate why thwarting the proposed attack is a difficult research challenge.

1 Introduction

So far, Web security on the client side has been governed by content isolation and by preventing attacks such as script injection, drive-by downloads, and others. For instance, Lekies et. al. described how using local storage for content caching results in script injection, and how to prevent it [19]. 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[1] validation leads to execution of undesired functionality in real life websites [15]. 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-downloads [30].

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

Nonetheless, in spite of the significant efforts invested to secure each API, the undesired consequences arising from client-side API combinations remain uncharted. Herein, 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[2] 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 allow for cross-origin data transfers among browsers.

Moreover, we conclude that the combination of cross-origin channels and the increase of local storage 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.

The presented attack has *three* 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 the browsers and transferring it over direct browser-to-browser links. *Second*, an attacker keeps the website'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 is his/her browser is of sensitive nature, legally or morally. *Third*, the attack is very difficult to defeat because, in principle, every API call required by the attack is not harmful on its own under the current Web security model.

1.1 Contribution and Outline

We describe a novel attack in which the storage and networking capabilities of the browser are abused for the attacker's benefit without requiring any additional installations or vulnerabilities on the client's side. While describing our attack, we also enumerate the security assumptions from postMessage API, the programmatic iframe creation and WebRTC which led to the browser abuse vector.

We have implemented a proof of concept browser network which has long term storage capabilities without making the user aware of its existence. In

[2]Although the website can always request an increase in the quota, this must be approved by the user. This must be avoided by an attacker who wants to hide the resource abuse from the user.

the proposed browser network, data is transferred over peer-to-peer links between browsers regardless of the origin from which they loaded the JavaScript code from, i.e. not covered by the Same Origin Policy. In addition, the persistent storage space allocated for the network inside each browser grows as more origins serve the malicious JavaScript code.

Also, we evaluated the proof of concept through a set of experiments. In each experiment, several real-life browsers were automated to visit a malicious website in a controlled environment; furthermore, several parameters, such as the number of visitors, time between visits, and the visitor return rate, i.e. how many visitors returned to the website in a given period of time, were varied for each execution. Throughout the experiments, network traffic as well as the status of the network was collected for off-line analysis. The results of the analysis show the feasibility of the attack.

From a more constructive perspective, we discuss the challenges faced when attempting several countermeasures and their potential side effects.

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

2 The Attack

We will present the attacker model first, and move to the attack overview and its details afterwards.

2.1 Attacker Model

Throughout the paper, we assume an attacker 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. Neither of these techniques require server side access. Besides, 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 [3]. 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. This can be achieved through a cloud service [29], or by hosting the server. Notably, this server only intervenes during the connection session establishment, but it is not used to transfer data between browsers.

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, 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 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 afterwards.

2.2.1 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 include the Abusive Script in an Intended Site in two ways: delivering the Abusive Script through an advertisement or injecting the code in an Intended Site

To deliver the Abusive Script through an advertisement network, an attacker can purchase browser time distributed across the Web. The effec-

[3] All are separate Origins According to RFC 6454

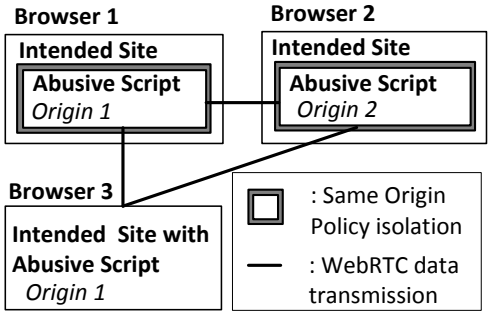

Figure 1: Overview of the Attack

tiveness of delivering JavaScript code to millions of browsers spending a few tens of dollars has been already demonstrated by Grossman et. al. when they created their "million browser botnet" [14]. 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 [38].

To inject an Abusive Script in an Intended Site, the attacker can use known attack techniques such as Man in the Browser [21, 40], Proxy Cache Poisoning [37], or cross-site scripting among others. Cross-site scripting is a particularly promising way to infect websites, given that by 2013 more than 6000 unique vulnerabilities were found across the Alexa top 500 websites (9.6% of the analysed sites) [20]. Further, unlike the case when the Abusive Script is delivered through an advertisement network, the Abusive Script would share the execution context with the Intended Site in this case, as depicted by Browser 3 in Figure 1.

2.2.2 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 in any way.

Besides, data can be sent to browsers who loaded the Abusive Script from any origin, therefore allowing for cross-origin communication not only among different Intended Sites, but also between different Abusive Script origins too. This is possible because according to the proposed security architecture[4] for WebRTC dataChannels [31], enforcing the Same Origin Policy between browser-to-browser channels does not provide any additional security. This design decision was based on two reasons: data channels do not inject code in 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.

2.2.3 Increasing the Local Storage Limit

In this section we will describe how the combination of invisible iframes, local storage, and the postMessage API allows an Abusive Script to use space beyond the intended quota per website, i.e. 5 MB. We start by listing the security assumptions behind each API. Afterwards, we describe the mechanism used to increase the storage quota. To conclude this section, we clarify the issues related to the initial security assumptions leading to the attack.

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. When it comes to spawning an iframe during runtime, no visible problem appears because, unless the iframe and the parent window share the same origin[5], 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. Finally, according to the postMessage specification[24], the assumptions dictate that, as long as developers validate the origin of the messages exchanged and their

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

[5]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.

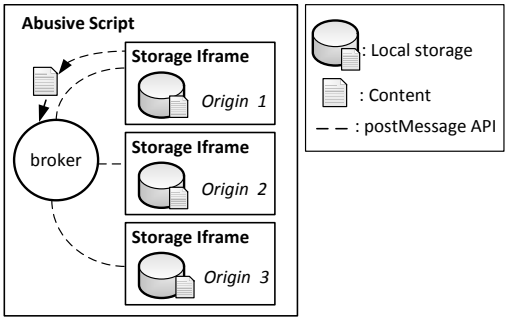

Figure 2: Storage Quota Increase

proper encoding, no vulnerabilities can be exploited. The rationale behind these validations are to prevent websites from acting on commands sent by malicious windows and to avoid script injection attacks respectively.

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. However, through the postMessage API they can communicate with the parent window, i.e. Abusive Script, allowing it to access their local storage. In other words, an attacker can use the postMessage API to create an asynchronous intra-browser messaging system to exchange control commands and data between the Storage Iframe and the Abusive Script, shown as "broker" in Figure 2. In turn, the Abusive Script obtains a quota equivalent to the number of Storage Iframes spawned multiplied by the browser storage quota.

The main problem with the postMessage API security model is its implicit assumption of a benign and a malicious site in the postMessage API interaction. This model falls short when both origins communicating behave maliciously; for example, when they conspire against browser's the visitor. Obviously, without the postMessage API, the attacker could still abuse the browser's storage by spawning frames; however, this would achieve a Denial of Service attack at most, since data would not be reachable when origins differ.

2.2.4 Inter-Browser Cross-Origin Communication

Inter-browser communication is paramount if the attacker wants to instruct browsers to share data among each other. This functionality relies on the

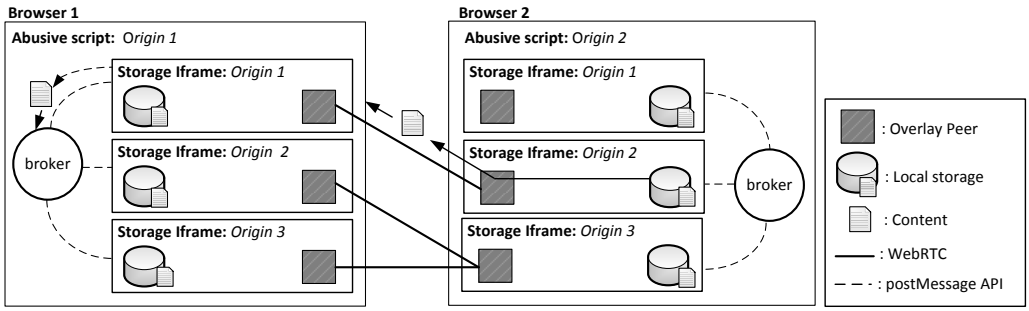

Figure 3: Complete Attack

WebRTC dataChannels [25] which requires an initial negotiation phase. Such initialization phase is solved by the implementation of the *Interactive Connectivity Establishment* protocol (ICE) [17]. In particular cases, when browsers are behind a router with *Network Address Translation* (NAT), a server providing *Session Traversal Utilities for NAT* (STUN) [34] 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) [22] is needed for the communication.

2.2.5 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 Abusive Script presented in Figure 2 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, 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 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. Naturally, 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 [11], we implemented this index in a centralized server because this extension does not strengthen nor weaken our argumentation on the security issues raised by the attack. Likewise, our prototypical implementation does not divide large files into chunks to store them, but requires files to have at most 5 MB when they are encoded in base 64.

We have tested our implementation by building a cross-origin network using Chrome 43.0.2357.81, and Firefox 38.0. In both browsers, the circumvention mechanism to increase the storage quota available for the network worked. Each browser has a 5 MB quota limit enforced per origin; as a result, if there are 2 origins capable of serving the content, each browser can host up to 10 MB, if there are 10 origins, each browser will host 50 MB, etc.

Now, to address the details of our proof of concept implementation, we first present the components required for our implementation. Afterwards, we provide details on the interaction between the different components during runtime and the file replication technique used.

3.1 Components

Our proof of concept requires the following components: an Abusive Script, a signalling server, and a peer and file index.

Abusive Script: We have implemented a JavaScript Abusive Script included in a very simple website. Our "broker" uses the postMessage API to exchange control commands asynchronously between the Abusive Script and the Storage Iframes storing the files. We have adopted a hierarchical approach where the Abusive Script commands each Storage Iframe to exe-

cute actions, e.g. retrieve a file from another peer, and receives callbacks with the status of the task afterwards. This provisions the Abusive Script with an overview of the files that are stored locally. This in turn, enables the network to ensure that files are not replicated more than once per browser. For our proof of concept, we host the Abusive Script and the Storage Iframe in a Web server reachable under n different origins, where n is the number of Storage Iframes created.

Signalling Server: We used a local installation of the PeerJS Server [8]. 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. This facilitates the initial steps required to establish WebRTC channels among them.

Peer and File Index: this is a Python server (using the Tornado framework [39]) 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. Due to the asynchronous nature of WebSockets, clients can notify the Peer and File Index with updates at any time without incurring in the overhead of an HTTP request. Such updates include, for example, notifying the index that a new file is available in the peer. This allows every browser in the network to keep an almost real-time synchronization with the file and peer index. Further, since every browser keeps a WebSocket open with the Peer and File Index server, the server can safely assume that a given browser (and all its Overlay Peers) left the network when the browser connection is closed. Also, for visualization purposes, this server offers a simple HTML page to upload files to a Storage Iframe, retrieve files from other browsers, an query an updated index of peers and files.

3.2 Component Interactions

Before discussing the replication techniques in Section 3.3, we present the general message exchange between the different actors in Figure 5. In this set-up it, is assumed that 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 hosted by 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

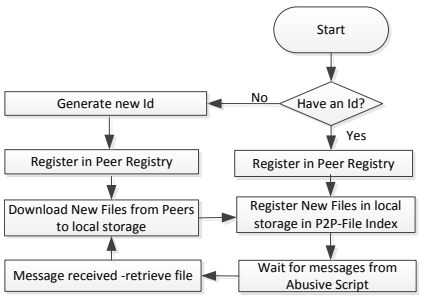

Figure 4: Storage Iframe Execution Flow Diagram (proof of concept)

of origins serving the code. Each Storage Iframe follows the flow diagram in Figure 4. 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 [6]. 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 a new file arrives to Alice's browser, Alice's browser will communicate this to the *Peer and File Index* server.

To keep an updated *Peer and File Index*, the index server removes peers and files hosted by browsers for which the WebSocket connection has been closed. In any case, when any event implying index changes takes place, every browser in the network is notified asynchronously, e.g. Bob and Charlie's browsers, through the WebSocket connection established between the browser and the index server as shown in Figure 5 with the arrows labelled as *Async index updates*.

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

3.3 Content Replication Technique

The replication technique followed by each Abusive Script for the proof of concept is very simple. When there is space in a Storage Iframe, the Abusive Script determines a file which complies with the following conditions, based on the current state of the index: on the one hand, the file should have the least amount replications in the network, and on the other hand, it should neither be stored in the Storage Iframe with available space nor any of its siblings, i.e. it is not replicated in this browser. Afterwards, the Storage Iframe is commanded to download the file determined in the previous step. This guarantees that when nodes leave and files are being less replicated, they are copied to other nodes before they perish. Although this is a very simple replication technique, it is enough to exemplify the implementation of a replication strategy.

We have leveraged the asynchronous features of JavaScript, and the PostMessage API to do a greedy replication from the Abusive Script. First of all, it must be noted that the creation of a Storage Iframe requires a registration process. The registration process includes connecting to the index server, creating the peer in the WebRTC network, and registering the files stored already in its local storage. The greedy replication technique consists in the following: when the Abusive Script starts, it creates every Storage Iframe, and starts distributing tasks to replicate files immediately. However, since the registration process executed inside each Storage Iframe, and the task distribution (done by the Abusive Script) happen concurrently, the Storage Iframe could receive a command to download a file that is already replicated in the browser, but which has not been registered in the index yet. However, in case the Storage Iframe requests a file already replicated in the browser before the index has been fully updated, it will be discarded once it is received. This approach will use more network than strictly necessary in browsers that are coming back, but will ensure that new visitors start replicating immediately. We prioritize fast replication over network usage because files not replicated in timely fashion are lost, while using more network on the browser's side does not bring any cost to the attacker.

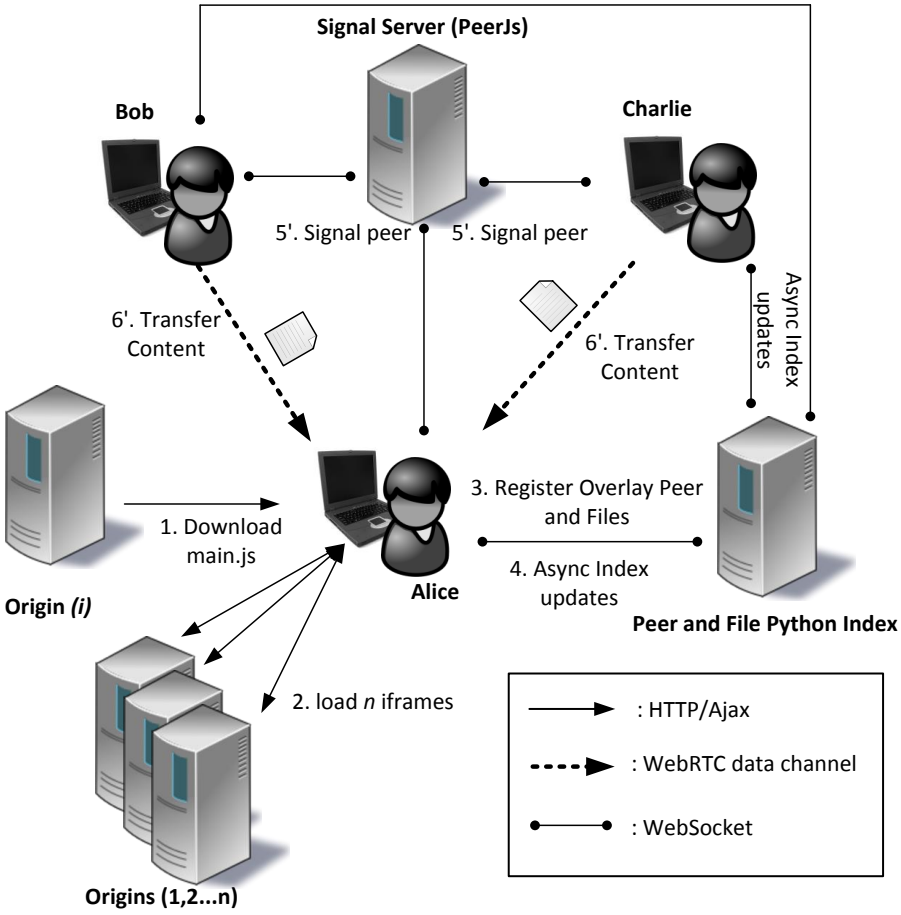

Figure 5: Proof of Concept Diagram

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 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 the 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. Also, every component was restarted between experiments to ensure that sequential runs do not interfere with each other.

4.1 Set-up

Linux containers constitute a resource friendly approach to execute software in an isolated manner. The main difference between containers and virtual machines is that containers reuse the host kernel, and follow a copy-on-write approach. We have used Docker [12] Linux containers to ensure that tests have exactly the same initial state file system. As Figure 6 shows, we have used docker containers to execute the so called *selenium controller*. The selenium 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 then executed on a Chromium browser inside the docker instance through a Selenium [36] driver. To run a headless Chromium browser inside containers, we used Xvfb [41] 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. More to the point, this architecture allowed us to execute the exact same instructions for every browser in both experiments: the attack evaluation, and the simulation of the countermeasure.

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 instance were run in separate containers, in the same host machine. Also, a specific */etc/hosts* file was automatically generated and copied into every container, so that it can address the servers by name regardless of the docker IP. Further, the n domains used for the Storage Frames point to the same apache server in the hosts file.

On the bottom of Figure 6, the *orchestrator* represents a Python program sending actions to every selenium controller used for the experiment. This is a multi-threaded Python application using the Tornado [39] framework to implement 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.

4.2 Data Collection

Figure 7 shows the data sources required for our evaluation in gray-shaded boxes. The data sources were: a network capture including all the traffic during the experiment, and the log files where the peer and index server stores the count of number of replications per file, i.e. a simple array. The former capture is performed by executing tcpdump on the host to capture all the bytes transferred through the *docker0* network interface, which is used by docker to forward all the traffic between containers, and between containers and the host too. The latter is a file, generated by the *Peer and File Index* server, where every change to the file index is logged. The volume where the

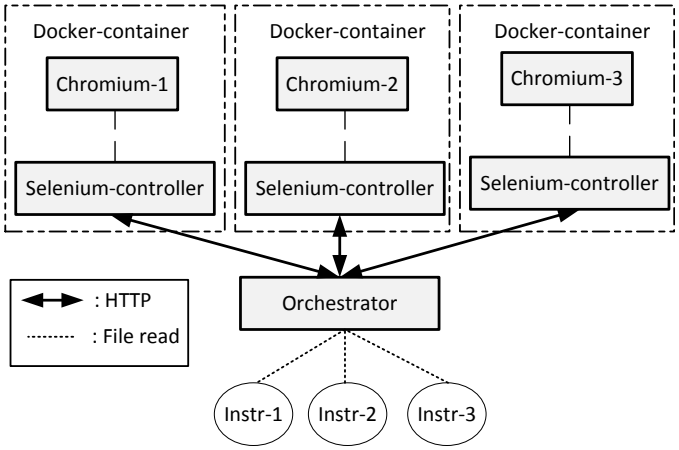

Figure 6: Overall Measurement Set-up for 3 browsers

index log file is stored is actually mounted from a folder in the host system allowing us to recover after the container has finished its execution.

An important property of the browser network is to offload the network stress required to transfer files to other browsers for replication purposes. Thus, we capture the network traffic count to count the bytes transferred over peer-to-peer links using WebRTC and compare it to the number of bytes transferred between browsers and servers during the experiments. In this particular scenario, running docker instances simplifies the recognition of browser-to-browser communication based on the IP addresses. Otherwise, if we would have run every browser as a simple process in the host, filtering the traffic would have been very challenging due to the complexity of the protocols used as part of the WebRTC file transfer, the connection establishment, etc.

4.3 Browsers' Behaviour

A selenium controller has the possibility to do one-time visits, i.e. a non-returning visitor, 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 in 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

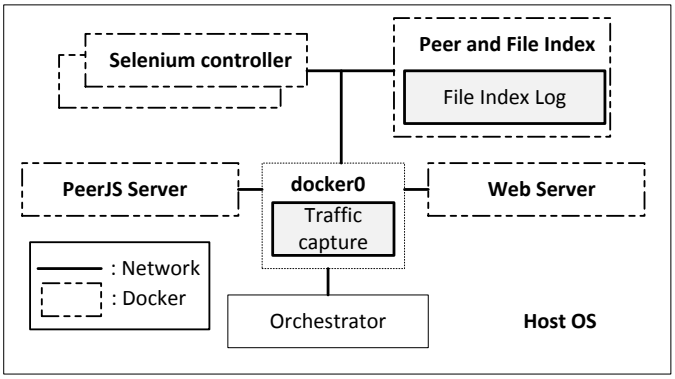

Figure 7: Data Collection

equivalent to a non-returning visitor also returns to the website 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 8. 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 gray-shaded areas for each browser, while time between visits is represented by white sections.

4.4 Measurements

For the experiments, 10 domains have been used in order to use up to 50 MB of data in each browser. 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 once they have been encoded in base 64, which is required to use the local storage API. 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 [26] random gener-

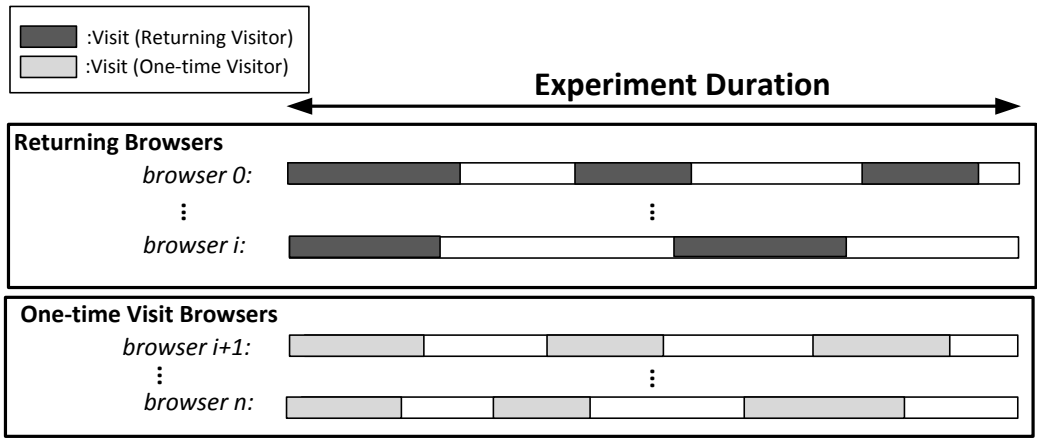

Figure 8: Visits Simulation

ator. 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 [9]. Further, research has reported web sessions to have a mean value of 74 minutes [5]; also, it is known that certain pages such as Facebook, have users with sessions ranging from a few to several tens of minutes [6]. 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 using the ranges [10-40], [110-140] and [210-240] seconds by the random generator of the NumPy library [26]. 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

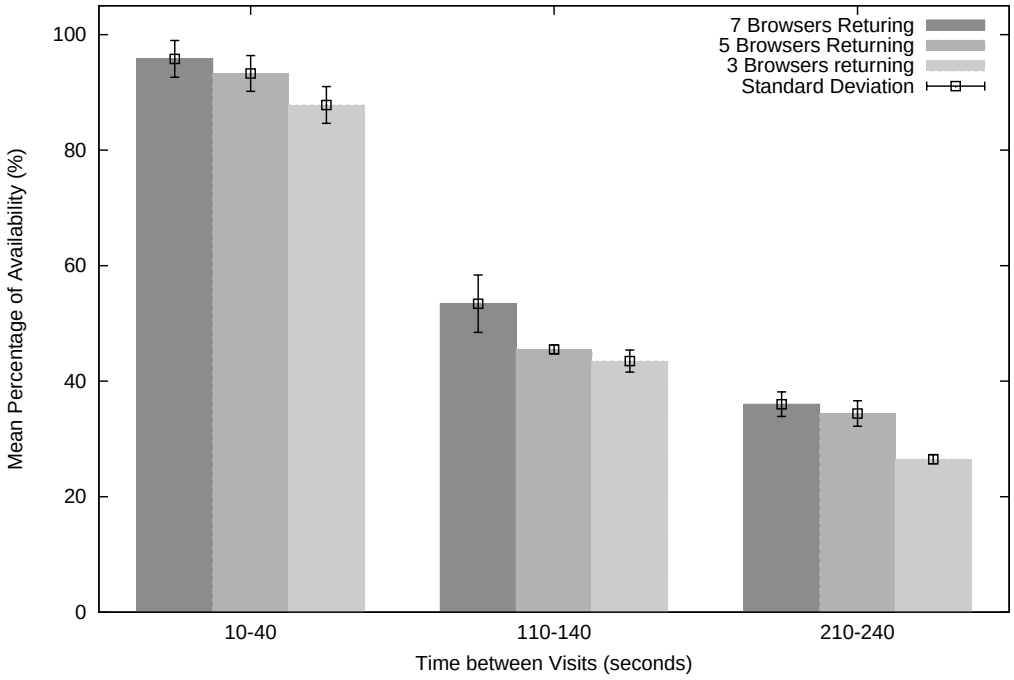

Figure 9: Attack Evaluation with *visit lengths* between 30-50 sec. - with 5 minutes experiments

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.

4.4.1 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. Afterwards, the average value and standard deviation for the array of percentages was calculated using Python NumPy [26].

As shown in Figure 9, 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 websites per month, or per day without any loss of generality. Moreover, considering that a recent marketing report [9] 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 come back 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. Naturally, this could be leveraged by an attacker exploiting several cross-site scripting vulnerabilities on the top Alexa websites as shown by Lekies et .al. [20].

Given that we have already covered the visit length and the visitor return rate, it now boils down to assessing whether the concurrent sessions opened by browsers during our experiment is feasible in real world websites. To this end, we do an approximate estimation of this based on average values. First of all, in equation 1, we calculate the expected number of visits per browser. Thus, we divide the time for the experiment by the average time between one visit and the next, i.e. average time of the visit plus average time of the waiting time between visits. Afterwards, as shown by equation 2, the number of visits per browser times the number of browsers yields the total number of visits in our experiment.

$$Visits_{browser} = \frac{time_{experiment}}{(meantime_{visit} + meantime_{waiting})} \quad (1)$$

$$Visits_{total} = Visits_{browser} * Browsers \quad (2)$$

Assuming a uniform distribution of visits and using the pigeon hole principle this value could be extrapolated to 133.000 visitors per week. This number seems to be acceptable, given that currently the top 500^{th} site according to Alexa's ranking [3, 4] 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 [5].

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 only extrapolate the effectiveness of the presented attack under the following assumptions. 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, visitors of the websites have sessions in the range between 30 and 50 seconds.

4.4.2 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 [13] 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 to represent 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 10 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 colors 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. This effect changes when there are many returning browsers (7), see Appendix for more detailed network capture information. Also, the fact that there is data being transferred between returning browsers happens due to the greedy replication approach mentioned in Section 3.3.

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 less 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 10 shows that non-returning browsers receive 23.39 (18.9 + 4.49) MB from returning and non-returning browsers in 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 (6.97) MB to other non-returning browsers, is a good sign of their contribution to keep files replicated.

5 Towards Countermeasures

We analyse the local storage increase, the cross-origin browser communication, and then the complex attack.

Preventing the Increase of Local Storage constitutes a good start towards a countermeasure against the combined attack (a simulation is presented in Appendix A); however, the attack would still be feasible with

	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

Figure 10: Average data (in MB) transmitted with 5 returning senelinum controllers - time between visits in [10-40] seconds

enough browsers executing the Abusive Script. Be that as it may, the security community has obliged in the past with similar conditions; for instance, a Denial of Service is always possible when enough clients are available to an attacker.

Pragmatic options to limit the quota increase include: limit storage, limit number of frames spawned and throttle the postMessage throughput.

An idea to limit storage is to apply a quota to the parent window, i.e. Abusive Script, covering not only the space used by it but also the space used by frames contained by it. However, this raises two problems. On the one hand, this opens a Denial of Service attack from an iframe included in other sites, in which the use of the quota from the frame would lead to the starvation of other frames and the parent window. On the other hand, sharing the quota can bring privacy issues because a frame could attempt to use as much space available in the storage at different points in time, to learn how the parent window or other frames utilize local storage. This creates a side channel between frames and the parent window.

Limiting the number of frames inside a window is the second option to prevent the local storage explosion; however, it needs to be explored whether a feasible number could be chosen, so that abuse is prevented while keeping today's Web running. This should be determined by observing behaviour patterns on the World Wide Web.

Another option would be to follow similar approaches to the WebSocket implementation in browsers, and throttle the postMessage API throughput, i.e. limit number of messages per unit of time. This approach would not prevent the actual allocation of the space and its availability within each browser; however, this modification is likely to significantly increase the amount of browsers needed to achieve high availability of the data across

the network since they join and leave with high churn. Similarly to the previous approach, its applicability should be assessed by analysing today's programming patterns on the Web.

Preventing Cross-Origin Browser to Browser Communication would prevent an attacker from transferring data among Storage Iframes with different origins (See data transmission between Browser 1 and Browser 2 in Figure 1). However, this still would not change the fact that the Same Origin Policy separation between the Abusive Script and the Intended Site does not prevent the attack. Further, even enforcing the Same Origin Policy between scripts exchanging data through WebRTC disables envisioned use cases. For instance, according to the initial draft on the WebRTC security architecture [31], different origins should be able to use an Identity Provider to authenticate themselves before establishing calls with each other. Further, the implementation of Cross Origin Resource Sharing (CORS) [1] would not help, since the attacker can host his/her own Storage Iframes, and therefore allow requests from other origins.

Lastly, one could explore **the detection of the complex attack**, as described in Section 2.2.5, through dynamic monitoring. To this end, detecting the particular pattern of API calls and data flows observed in our attack is definitely possible; however, this yields a very specific enforcement mechanism covering only this particular attack. Conversely, if generic patterns of browser abuse should be detected, previous work on malicious JavaScript code detection would be a formidable starting point [33, 35, 10]. But, the biggest challenge for such approach is finding a proper dataset, previously labelled, to train the algorithms. This is very difficult since we are describing a new kind of attack.

6 Related Work

Our main contributions are comprised of a novel attack, and a prototype of a hidden content distribution network; therefore, we divide this section accordingly.

6.1 Similar Attacks

From the storage abuse perspective, Feross discovered that a single website could instruct local storage to store data in infinite suborigins. This lead to

abuse the users' disk, filling it until the browser crashes [7, 2]. This relates to our quota bypass mechanism in the sense that both rely on using different origins to increase the quota. However, fixing the bypass mechanism proposed by us has proven to be more difficult, because the malicious behaviour is not executed by one script writing to several suborigins, but instead is executed across sites loaded from several origins and collaborating through an asynchronous message channel, i.e. postMessage API.

Also, there is previous work exploring API combinations for network-related abuse attacks. On the one hand, Parra et. al showed the applicability of forcing the visitor's browser to launch a Denial of Service attack against regular Web servers by spawning WebWorkers that open as many WebSockets as possible [28]. On the other hand, port-scanning from browsers is possible by combining cross-origin requests with WebsSockets [18] and analysing how their states change.

6.2 Content Distribution

PeerCDN [16] is a WebRTC-based Content Distribution Network (CDN) forcing the visitor's browser to share the website's static HTML content with other browsers without his knowledge. Owners of the company claim to achieve a 90% bandwidth reduction for the server hosting the site. After PeerCDN was created, Zhang et. al implemented another browser-based CDN called Maygh [42]. 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. Their architecture includes a coordinator using WebRTC and RTMFP as the directory of clients and content is hosted. 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 recent research work to transmit video streams between browsers using WebRTC [23, 27, 32] to ease the burden imposed on servers hosting the video streams.

Although these three approaches execute JavaScript code without making the visitor fully aware of this, there are three important differences between the previously mentioned approaches and ours. First of all, PeerCDN and the video distributing networks do not use the browsers to store persistent

information, and retrieve it afterwards. The second difference is that the content distributed by peerCDN, Maygh and the video distribution networks is not arbitrary, i.e. it matches the page or the video that is rendered to the visitor of the website. The third and most important difference is that they neither bypass the storage, nor use the cross-origin capabilities as we do.

7 Conclusion

The cross-window communication channels through postMessage API and the direct browser-to-browser data channels trough WebRTC bring more flexibility to the Web developer. However, in both cases it is implicitly assumed that preserving the isolation of the JavaScript context and restricting access to the DOM of scripts engaging in communication is enough to thwart attacks.

Nevertheless, we have shown that an attacker serving malicious code, i.e. Abusive Script, from different origins can use a unified local storage occupying space beyond the intended browser quota. Also, the Abusive Script requires neither access to the DOM nor access to the JavaScript execution context of the compromised website, i.e. Intended Site. Thus, presenting an advertisement to the user is as effective as injecting malicious code on the visited website; as a result, the attack surface is considerably increased in comparison to most existing Web attacks which have to bypass the Same Origin Policy.

Furthermore, circumventing the local storage quota 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 both browsers differ. This allows an attacker to create a browser network for data storage and distribution in a hidden manner.

All in all, we believe there are two revolutionary aspects of our attack. First of all, we present an attack where several origins collude against the user, therefore invalidating several implicit assumptions behind current Web security models. More specifically, assuming that the Same Origin Policy is not needed in cases when data can be relayed through the server, provisions the attacker with a powerful platform to distribute his data across browsers through WebRTC. Likewise, we show that preventing code injection and ensuring proper origin authentication in postMesage API interactions is not enough when two or more origins collude to increase the total storage used by

a particular site. The second key aspect of our attack is that we are extending a young research field exploring browser abuse through combination of client-side APIs and assessing their feasibility in a real-life deployment. Further, we hope our discussion of countermeasures constitutes the foundation towards the prevention of the attack while preserving today's Web functionality.

Lastly, we expect to raise awareness on the need to expand the Web security model to include scenarios in which several origins collude against a visitor to obtain or misuse the browser's resources, as presented in Section 2.

References

[1] Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/, jan 2014.

[2] F. Aboukhadijeh. The Joys of HTML5: Introducing the new HTML5 Hard Disk Filler API, 2013.

[3] Alexa. http://www.alexa.com/, 2015.

[4] Alexa Traffic Ranking and visitor statistics for 7 years. http://www.rank2traffic.com/, 2015.

[5] S. Antonatos, P. Akritidis, V. T. Lam, and K. G. Anagnostakis. Puppetnets: Misusing Web Browsers As a Distributed Attack Infrastructure. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.

[6] E. Athanasopoulos, A. Makridakis, S. Antonatos, D. Antoniades, S. Ioannidis, K. G. Anagnostakis, and E. P. Markatos. Antisocial Networks: Turning a Social Network into a Botnet. In *Proceedings of the 11th International Conference on Information Security*, ISC '08, pages 146–160, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Web Code Weakness allows Data Dump on PCs. http://www.bbc.com/news/technology-21628622, 2008.

[8] M. Bu and E. Zhang. The PeerJS library. http://peerjs.com/, 2012.

[9] Clicktale: Web-Aanalytics Benchmark Q2 2013. http://blog.clicktale.com/wp-content/uploads/2013/10/ClickTale-2013-Web-Analytics-Benchmarks.pdf, 2013.

[10] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 281–290, Raileigh, North Carolina, USA, 2010. ACM.

[11] D. Dias. WebRTC Explorer. https://github.com/diasdavid/webrtc-explorer, 2015.

[12] Docker. https://www.docker.com/, 2016.

[13] Dpkt package. https://pypi.python.org/pypi/dpkt, 2016.

[14] J. Grossman and M. Johansen. Million Browser Botnet. https://www.blackhat.com/us-13/briefings.html, 2013.

[15] S. Hanna, E. C. R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song. 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.

[16] J. Hiesey, F. Aboukhadijeh, and A. Rajah. PeerCDN. https://peercdn.com/, 2013.

[17] R. J. Rfc5245: Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. RFC 5245, April 2010.

[18] L. Kuppan and M. Saindane. JS Recon. http://www.andlabs.org/tools/jsrecon/jsrecon.html, 2010.

[19] S. Lekies and M. Johns. Lightweight Integrity Protection for Web Storage-driven Content Caching. In *Workshop on Web 2.0 Security and Privacy (W2SP)*, 2012.

[20] S. Lekies, B. Stock, and M. Johns. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1193–1204, New York, NY, USA, 2013. ACM.

[21] R. Linn and S. Ocepek. Hookin' Ain't Easy: BeEF Injection with MITM. Technical report, 70 W. Madison Street, Suite 1050 Chicago, IL 60602, 2012.

[22] R. Mahy and M. P. Rfc5766: Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat(stun). RFC 5766, April 2010.

[23] A. J. R. Meyn, J. K. Nurminen, and C. W. Probst. Browser to Browser Media Streaming with HTML5. Master's thesis, Aalto University, 2012.

[24] Mozilla Developer Network(MDN) - Window.postMessage(). https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage, April 2015.

[25] A. Narayanan, C. Jennings, A. Bergkvist, and Burnett. WebRTC 1.0: Real-time Communication Between Browsers. W3C working draft, W3C, Sept. 2013. http://www.w3.org/TR/2013/WD-webrtc-20130910/.

[26] NumPy. http://www.numpy.org/, 2016.

[27] J. Nurminen, A. Meyn, E. Jalonen, Y. Raivio, and R. Garcia Marrero. P2p media streaming with html5 and webrtc. In *Computer Communications Workshops (INFOCOM WKSHPS) 2013 IEEE Conference on*, pages 63–64, April 2013.

[28] Parra Rodriguez, Juan D. and Posegga, Joachim. *Security and Privacy in Communication Networks: 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*, chapter Why Web Servers Should Fear Their Clients, pages 401–417. Springer International Publishing, Cham, 2015.

[29] PeerServer Cloud service. http://peerjs.com/peerserver, 2016.

[30] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[31] E. Rescorla. ietf-draft: WebRTC Security Architecture. https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-11, March 2015.

[32] 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*, pages 1–6, Jan 2014.

[33] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient Detection and Prevention of Drive-by-download Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, Austin, Texas, USA, 2010. ACM.

[34] J. Rosenberg, R. Mahy, P. Matthews, and W. D. Rfc5389: Session traversal utilities for nat (stun). RFC 5389, RFC Editor, October 2008.

[35] K. Schütt, M. Kloft, A. Bikadorov, and K. Rieck. Early Detection of Malicious Behavior in JavaScript Code. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, AISec '12, pages 15–24, Raileigh, North Carolina, USA, 2012. ACM.

[36] SeleniumHQ: Browser Automation. http://www.seleniumhq.org/, 2016.

[37] L. shung Huang, E. Y. Chen, A. Barth, E. Rescorla, and C. Jackson. Talking to yourself for fun and profit. In *Web 2.0 Security and Privacy (WSP 2011)*, 2011.

[38] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[39] Tornado Web Server. https://github.com/tornadoweb/tornado, 2009.

[40] S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 525–530, New York, NY, USA, 2014. ACM.

[41] Xvfb: virtual framebuffer X server for X Version 11. http://www.x.org/archive/X11R7.6/, 2016.

[42] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram. Maygh: Building a cdn from client web browsers. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 281–294, New York, NY, USA, 2013. ACM.

A Countermeasure Simulation

We have simulated an enforcement mechanism preventing an Abusive Script from increasing its local storage quota. To do this, we run the same experiments presented in Section 4.4, i.e. the exact same instructions to every browser, but with only one domain instead of 10 domains hosting the malicious site. This is equivalent to enforcing a 5 MB quota, instead of 50 MB quota per tab.

The mean percentage of the time in which a file is available in the network and its standard deviation is depicted in Figure 11. A significant reduction in the availability of files in the network, and a high standard deviation due to the effectiveness of the countermeasure can be observed. When examining the index log files, we observed only few files being replicated at the same time. This happens due to the lack of space in the browser to host more files. Also, several files have a 0% availability while others have higher values. This leads to high standard deviation, i.e. even greater than the mean value.

As mentioned in Section 5, one can overcome this countermeasure with enough visitors. However, this would increase the complexity of the attack severely making it less attractive.

Although Chromium is open source, we decided to simulate the countermeasure instead of modifying the browser's source because, as mentioned in Section 5, it is still unclear how this countermeasure can be implemented without compromising functionality, privacy, and without opening vectors for Denial of Service.

B Network Measurements

This Appendix includes the network captures on a peer-to-peer basis, discriminating by browser, type of server, etc. For readability purposes, Figure 12, 13 and 14 are divided in regions according to the number of returning browsers.

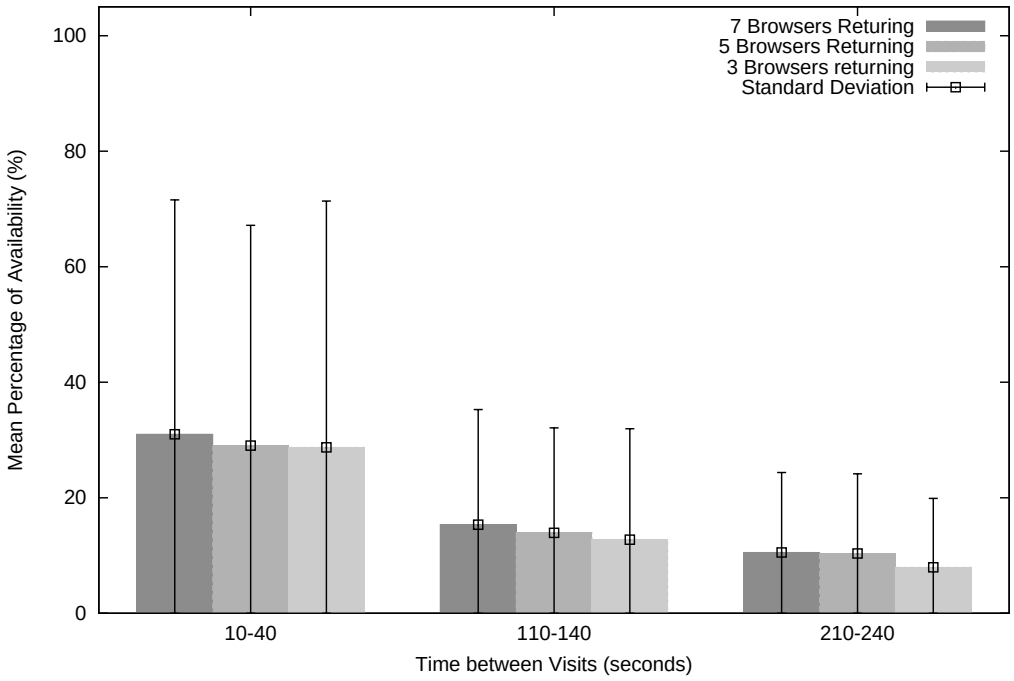

Figure 11: Execution of experiments with a 5 MB quota on the client side

	browser0	browser1	browser2	browser3	browser4	browser5	browser6	browser7	browser8	browser9	index	peerjs	webserver
browser0	-	2.59	27.85	34.35	10.40	12.17	49.17	1.41	12.71	28.09	0.03	0.32	0.03
browser1	60.34	-	42.50	8.77	31.35	5.28	22.86	4.03	46.06	27.06	0.04	0.39	0.03
browser2	32.52	40.69	-	3.73	59.95	18.89	13.86	11.51	56.36	20.30	0.04	0.46	0.04
browser3	1.47	1.04	3.54	-	0.46	-	0.07	1.86	4.03	8.51	0.03	0.30	0.40
browser4	0.44	1.50	2.67	10.02	-	3.73	10.52	25.95	0.15	3.49	0.03	0.37	0.41
browser5	0.51	0.20	2.45	-	0.15	-	-	7.31	-	-	0.03	0.18	0.32
browser6	2.23	6.62	4.27	1.87	0.43	-	-	7.33	2.22	6.89	0.03	0.31	0.46
browser7	0.08	0.20	2.39	0.09	4.77	0.32	0.32	-	26.69	0.44	0.03	0.24	0.33
browser8	0.48	2.28	2.51	7.37	3.70	-	6.49	1.16	-	0.00	0.03	0.38	0.35
browser9	1.32	1.13	6.22	0.32	0.15	-	0.28	10.01	0.02	-	0.03	0.40	0.40
index	1.05	1.06	1.00	0.84	0.85	0.87	0.91	1.00	0.83	0.71	-	-	-
peerjs	0.46	0.48	0.70	0.26	0.35	0.19	0.30	0.26	0.34	0.30	-	-	-
webserver	0.03	0.03	0.03	4.13	4.16	3.47	4.13	3.46	3.47	4.16	-	-	-

Figure 12: Network Capture(MB) time between returning visits in [10-40] sec. 3 returning browsers

	browser0	browser1	browser2	browser3	browser4	browser5	browser6	browser7	browser8	browser9	index	peerjs	webserver
browser0	-	21.69	46.00	18.12	23.76	23.98	5.63	21.64	29.30	-	0.04	0.31	0.03
browser1	0.92	-	4.24	0.45	25.06	0.15	0.03	0.08	18.52	10.83	0.03	0.27	0.04
browser2	5.75	21.55	-	12.62	6.67	21.82	1.59	6.84	39.19	25.96	0.04	0.34	0.04
browser3	38.99	9.73	31.03	-	8.26	39.90	45.29	35.71	16.44	28.52	0.04	0.36	0.04
browser4	8.89	1.25	12.21	14.17	-	8.90	36.45	13.43	13.91	28.36	0.03	0.35	0.04
browser5	1.04	3.59	0.95	1.79	6.47	-	0.20	0.11	1.77	-	0.03	0.24	0.37
browser6	7.13	0.01	0.08	1.99	3.33	3.52	-	15.99	0.11	14.43	0.04	0.30	0.34
browser7	0.88	1.76	5.46	1.54	2.39	2.43	0.72	-	0.95	1.20	0.03	0.25	0.33
browser8	1.24	0.74	1.71	0.76	1.89	0.07	1.26	0.05	-	3.59	0.03	0.29	0.34
browser9	-	4.00	2.93	9.18	1.14	-	16.15	27.02	0.15	-	0.04	0.29	0.33
index	1.32	1.12	1.29	1.43	0.98	1.15	1.26	1.07	1.10	1.21	-	-	-
peerjs	0.43	0.26	0.35	0.58	0.38	0.25	0.33	0.27	0.27	0.36	-	-	-
webserver	0.03	0.03	0.04	0.03	0.03	3.45	3.47	3.47	3.47	3.47	-	-	-

Figure 13: Network Capture(MB) time between returning visits in [10-40] sec. 5 returning browsers

	browser0	browser1	browser2	browser3	browser4	browser5	browser6	browser7	browser8	browser9	index	peerjs	webserver
browser0	-	11.74	1.11	25.19	28.57	12.91	7.73	9.10	34.21	11.35	0.04	0.33	0.04
browser1	18.25	-	24.27	28.58	38.57	12.38	13.59	21.83	20.93	13.67	0.04	0.40	0.05
browser2	8.93	11.10	-	15.34	11.32	9.54	15.30	36.06	3.04	24.46	0.04	0.42	0.05
browser3	7.94	6.00	37.60	-	7.98	9.40	1.89	32.76	-	3.76	0.03	0.34	0.04
browser4	10.17	1.67	8.65	17.51	-	12.74	-	13.79	-	8.79	0.03	0.28	0.04
browser5	0.61	9.59	10.97	19.53	5.46	-	0.21	13.66	28.12	14.21	0.04	0.28	0.05
browser6	3.91	0.59	0.70	0.08	-	4.92	-	28.58	9.88	7.99	0.04	0.26	0.03
browser7	0.40	6.05	1.61	3.05	1.00	6.56	4.70	-	5.21	0.01	0.04	0.33	0.34
browser8	1.49	5.96	12.38	-	-	1.23	0.44	0.22	-	-	0.03	0.29	0.35
browser9	0.43	0.63	1.15	0.22	0.37	2.41	0.40	0.02	-	-	0.03	0.22	0.33
index	1.58	1.52	1.35	1.18	1.18	1.39	1.44	1.30	1.25	1.24	-	-	-
peerjs	0.40	0.43	0.54	0.41	0.33	0.30	0.26	0.33	0.27	0.24	-	-	-
webserver	0.03	0.03	0.03	0.03	0.03	0.03	0.02	3.46	3.47	3.47	-	-	-

Figure 14: Network Capture(MB) time between returning visits in [10-40] sec. 7 returning browsers