Cashing out the Great Cannon? On Browser-Based DDoS Attacks and Economics

Giancarlo Pellegrino
Saarland University
gpellegrino@mmci.uni-saarland.de

Christian Rossow
Saarland University
crossow@mmci.uni-saarland.de

Fabrice J. Ryba
Freie Universität Berlin
fabrice.ryba@fu-berlin.de

Thomas C. Schmidt
HAW Hamburg
t.schmidt@haw-hamburg.de

Matthias Wählisch
Freie Universität Berlin
m.waehlisch@fu-berlin.de

ABSTRACT

The *Great Cannon* DDoS attack has shown that HTML/JavaScript can be used to launch HTTP-based DoS attacks. In this paper, we identify options that could allow the implementation of the general idea of browser-based DDoS botnets and review ways how attackers can acquire bots (e.g., typosquatting and malicious ads). We then assess the DoS impact of browser features and show that at least three JavaScript-based techniques can orchestrate clients to send thousands of HTTP requests per second. Seeing the vats potential, we evaluate the economics of browser-based botnets and show that their cost are about as high as traditional DDoS botnets—while giving far less flexibility in terms of attack features and control over the bots. Finally, we discuss victim- and browser-side countermeasures.

1. INTRODUCTION

Distributed Denial-of-Service (DDoS) attacks continue to be a severe problem to the Internet. In April 2015, researchers observed a new type of DDoS attack, coined the *Great Cannon* [1]. Here, a powerful attacker injected malicious JavaScript code into HTTP traffic. The malicious code turned browsers into DoS clients by aggressively requesting web resources from victims. The Great Cannon thus acted as man-in-the-middle and was reported to manipulate web communication at Chinese ISPs to attack GitHub.

We envision that even less powerful adversaries can launch similar *browser-based* DDoS attacks. The Great Cannon incident has illustrated that web clients can serve as DoS bots—even without being compromised by malware. Attackers have abused the feature-rich communication API of browsers to launch DoS attacks, similar to how traditional DDoS botnet would operate. Actually, Grossmann and Johansen already presented this kind of threat at Black Hat USA 2013 [2], showing that attackers can inject malicious ads to launch DDoS attacks. Kuppan mentioned HTML5 even at Black Hat 2010 as a potential vector for DDoS attacks [3]. However, while the general attack principle is already well-known [4], there is no systematic understanding of which browser features may be (ab)used in such attacks. Right now, only anecdotal reports show that certain browser APIs can be abused, but without giving sufficient detail. Moreover, browser-based DDoS attacks seem promising also to less powerful actors, such as cyber criminals with economic incentives. However, little is known about the usefulness (and costs) of browser-based DDoS attacks for "typical" cyber criminals.

In this paper, we aim to close this gap. We start by exploring ways how cyber criminals may actually attract DoS clients, similar to their need to establish a traditional botnet (§ 2). For example, we review typosquatting and injection of malicious ads as potential methods to acquire new browser-based bots. We then systematically review features of modern browsers that could be used for attack, such as JavaScript APIs (e.g., XMLHttpRequest, WebSocket). Next, we measure how dangerous these features actually are in a DoS attacks. That is, in a local experiment, we measure the potential request rates of these attacks and discuss the evasion flexibility they offer (e.g., manipulation of HTTP headers).

Second, we provide an economic comparison between traditional DDoS botnets and advertisement-driven browser-based botnets (§ 3). We aim to answer the following research questions. What are the costs for operating the two types of botnets? How long do bots stay online, once acquired? And is it likely that profit-maximizing cyber criminals will jump on the wagon of attacks like the Great Cannon? We approximate the costs for both malware-driven and browser-based DDoS botnets. Our results show that costs are comparable, ranging between \$0.0006 and \$0.02 per day and attack source.

We conclude with a discussion on defenses against browser-based botnets (§ 4). We discuss defenses from the victim's point of view (such as header-based filtering), and present ideas on how current browsers can be adapted to mitigate such kind of attacks.

Our summarized contributions are as follows:

1. We systematically review browser features that may support browser-based DDoS attacks and measure their impact.
2. We approximate and compare the costs for malware- and browser-based DDoS botnets.
3. We discuss potential client- and victim-side defenses against browser-based DDoS attacks.

Our preliminary results show that there are still several open questions, which should be tackled by the research community. In this paper, we strongly advocate for continuing the systematic analysis of this threat landscape to increase trust in the web ecosystem.

2. BROWSER-BASED DDOS BOTNETS

The Great Cannon attack has revealed the vast potential of abusing normal web clients as weapon in DDoS attacks. We will revisit such browser-based DoS attacks from a different attacker model, in that we assume an attacker with economical incentives. According to most analyses, the Great Cannon attack was only possible because ISPs actively manipulated web traffic. The attacker(s) injected JavaScript code to normal websites that would launch DDoS attacks towards certain targets. In contrast, we assume an attacker with significantly less power, with the goal to assess if browser-based DDoS attacks may even attract other types of attackers. In Section 2.1, we thus discuss how an attacker can rent or invest in clients—instead of just hijacking them via traffic manipulation.

Furthermore, in Section 2.2, we will investigate various methods how an adversary can leverage HTML or JavaScript code to perform DoS attacks. Then, we will compare these attacks with features of traditional malware-based DoS bots.

2.1 Acquisition of Browser Bots

In order to launch a browser-based DDoS attack, the first step an attacker has to perform is acquiring an army of bots. In our context, *bot* refers to web clients that can be instructed using common HTML/JavaScript code. Note that we do *not* require to compromise the host, i.e., the browser-based bot is different from the malware-infected host. Still, how can an adversary find bots to launch a DDoS attack? We imagine a few methods an attacker might use, as presented in the following:

Typosquatting—First, an attacker could leverage typosquatting [5, 6] to obtain new clients that mistype a domain they wanted to visit, i.e., registering domain names that are similar to well-known sites. Once the clients visit the web site, the attacker would try to increase the time the user stays on this website. For example, the adversary may show the correct content via HTML `<iframe>` tags, hiding the fact from the user that she actually is on a fake website.

Popular typosquatting domains are already pre-registered by the owners of the correct domains or by attackers. However, the web domain space is large and an attacker may fish in the long tail. Domain tasting (i.e., temporarily registering domains) is not prohibited in general. For country code top-level domains, ICANN charges a surcharge of $0.20 per domain. For a domain at registrars such as Dynadot, this leads to overall costs of $1.00-$5.00. Assuming $2.5 on average and 10 visitors per domain/day, a quick back-of-the-envelope analysis shows that an attacker could easily instruct 4,000 clients with a budget of $1,000. Our preliminary measurements indicate that 10 visitors per day are reasonable for typosquatting domains that have been selected in a plain, non-sophisticated way—we expect significantly more visitors with carefully selected domains.

However, this field needs further research in several directions: How many users can be gathered in parallel and how long do the users stay on such sites? Which domains should be registered? We leave these questions to future work.

Instrumenting Machine-Generated Visits—Another way to attract clients is making websites popular (e.g., using SEO) and then try to catch machine-generated visitors, such as crawlers. As soon as a webpage is registered with a search engine or linked to another site, this page is likely to be scanned by crawlers or even attackers that scan for known vulnerabilities or backdoors. On arbitrary requests, the botnet controller can deliver pages including external references.

Hijacking Popular Websites—Since many years, attackers have hijacked well-known websites to exploit their visitors via drive-by downloads. Instead of infecting the clients, attackers could just inject some JavaScript code to the HTML structure of the website, which renders the attacking code.

Instrumenting Ad Networks—Finally, we discuss how to use ad networks to obtain browser-based bots—a technique that we will analyze in more detail in this paper. Online ads are important building block in the business chain of the web ecosystem. Ad networks provide a convenient way to distribute advertisements to a large crowd of users. Beneficial for the potential attacker is that ad services such as the Google Display Network charge users only per-click and not per-view basis. Pay-per-click ads are especially appealing to an attacker, as this allows to inject malicious code that is only *viewed* by clients and thus does not introduce additional costs.

2.2 Browser as a Bot

Next, we are going to compare browser-based DDoS attacks with attacks that DDoS botnets can launch.

DDoS botnets span malware-infected hosts that are specialized in the execution of distributed denial of service attacks. Typically, the bot master dispatches the target coordinates and the type of attack to each bot. Bots can perform a variety of attacks, such as SYN floods or requesting web resources via HTTP. Additionally, bots can vary each attack to evade detection and increase its effectiveness.

In contrast to malware, browser-based DoS attacks have limited flexibility. Here, we assume that the attacker does not exploit the browser, but just uses some active code (such as JavaScript) to launch an attack. In the following, we revisit communication features of modern web browsers and discuss to what extent they can be leveraged in DoS attacks. We aim to reproduce HTTP-based DoS attacks that are also common in modern botnets, such as Yoddos or DirtJumper [7, 8]. That is, we will discuss how we can generate DoS-like request behaviors purely by using a non-compromised browser. We chose to focus on JavaScript-based attack code due to its popularity and wide availability.

2.2.1 *DoS-Enabling JavaScript Features*

JavaScript programs can issue HTTP requests via APIs *designed* for the network communication such as XMLHttpRequest and WebSocket. Furthermore, JS programs may use *other* APIs that implicitly result in sending HTTP requests. For example, the JS program can modify the content of the `src` attribute of the `<img>` tag, and as a result, the browser will issue an HTTP request to fetch the content of `src`. In the following, we will describe four APIs that can be abused to launch HTTP(S)-based DoS attacks: XMLHttpRequest (XHR) API [9], the WebSocket (WS) API [10], and Server-Sent Event (SSE) [11], and the Image API [12].

XMLHttpRequest—This API is used to send asynchronous requests to the server side of a web application [9], i.e., XHR requests:

```
var target = "http://$target/";
var xmlhttp=new XMLHttpRequest();
xmlhttp.open("GET", target);
xmlhttp.send();
```

As opposed to the other three JS features, the XMLHttpRequest API allows a JS program to control some headers and the request method of the HTTP request. For example, the JavaScript program can set an arbitrary HTTP body and content type, and, it allows to set a few HTTP request headers, e.g., the request content type [9].

WebSocket API—WebSocket allows the client and the server side of a web application to establish a full-duplex stream-oriented communication channel [10], i.e., a web socket. WebSocket is an extension of the HTTP protocol in which first the parties perform a handshake and then can stream data over the web socket.

As the WebSocket handshake involves HTTP requests, an attacker may use XHR requests to initiate a web socket. To avoid that, the handshake implements a protection mechanism which leverages on additional headers that cannot be modified by client-side programs [10]. Unfortunately, the WebSocket protocol specifications do not describe any mechanism to protect non-WebSocket servers from malicious WebSocket clients. Malicious JS code may misuse the handshake by requesting resources hosted by a non-WebSocket server. Such server may ignore the characteristic web socket HTTP headers, and thus it can accept WebSocket handshake HTTP requests as normal HTTP requests. A malicious JS program can misuse the WebSocket protocol as follows:

```
var target = "ws://$target/";
var websocket = new WebSocket(target);
```

The variable `target` contains the URL of the target. WebSocket URLs use the HTTP scheme `ws://` or `wss://`. The scheme `ws://` resolves the default TCP port 80 and refers to a WebSocket server that does not use any secure transportation layers. Instead, the scheme `wss://` resolves the default TCP port 443 and relies on SSL/TLS. Our example code instantiates a WebSocket client and passes the variable `target`. As a result, the web browser will start the WebSocket handshake with the target. Although the web server is a non-WebSocket server, it will process the HTTP request as a valid request.

Server-Sent Event API—SSE is a communication API which allows a JavaScript program to receive a stream of events from the server side [11]. The channel is established similarly as seen before for the WebSocket. First, the browser sends and HTTP request to the server, which dispatches server events. Then, the server responds with an HTTP response and a stream of events. SSE may also be abused by a malicious JS program in an attack:

```
var target = "http://$target/";
var source = new EventSource(target);
```

Image API—The Image API is a JavaScript-based interface to the `<img>` HTML tag [12]. A malicious JavaScript program can abuse the Image API. For example, instead of providing the URL of an image, it can provide the URL of the target to attack as follows:

```
var img = new Image();
img.src = "http://$target/";
```

In this fragment of code, the JavaScript code initializes a new Image object. Then, it sets the `src` property of the Image class with the absolute or relative URL of the image—the target. When the browser interprets this fragment of code, it will issue an HTTP request for the resource / from $target.

2.2.2 *API Aggressiveness*

To launch successful DoS attacks, an attacker needs to instrument a client to send many HTTP requests. While we found that four JS APIs allow us to send HTTP requests in principle, we need to measure their "aggressiveness". In this section, we will describe an experiment to measure the request rates that the four APIs offer.

Testbed—In our experiments, we used a server (the target) and a client (the HTTP bot) connected to a Gigabit LAN. The server is an Intel Xeon dual-core 2.50GHz with 8GB of RAM. The client is a quad-core Intel i7. We set up Lighttpd [13] on the server listening to four TCP ports. We then configured the firewall to change the states of the four TCP ports to the following: open port (accept connections), ignore incoming packets (drop), respond with ICMP host unavailable, and finally, with a TCP reset packet.

The client was set up with Mozilla Firefox 37.0.2 and Google Chrome 42.0.2311.135. Then, we instrumented each browser with a JavaScript code that was constantly establishing requests for a run-time of 60 seconds. First, we wrote the malicious script to continuously invoke API calls. However, this approach causes the browser to stall or even crash. We modified this approach by setting a maximum number of API calls per second that the malicious scripts intends to send. We considered four different API call frequencies: 1000, 2000, 3000, and 4000 calls per second. In our tests, we also used web workers in order to parallelize the API calls. We used 0, 2, and 3 web workers. For each test, we captured the network traffic for further analysis.

Results—The result of the analysis is shown in Tables 1 and 2. Table 1a shows the aggregated results. Table 1b details the attacks via a single-threaded JavaScript program (no web workers). Table 2 shows the results with a varying number of web workers.

Table 1a shows the results of our tests per browser and per TCP port states. The columns [Reqs/s] is the average number of HTTP requests per second whereas [SYN/s] is the average frequency of TCP SYN packets sent to the server.

When the TCP port is open, the XMLHttpRequest can generate 1,000 and 2,100 HTTP requests per second with Chrome and Firefox, respectively. The WebSocket API turns out to generate less than 35 req/s with Chrome and almost zero request with Firefox. This is caused by the behavior of Firefox upon failure in setting up a WebSocket. In our experiments, we use the WebSocket handshake to request resources to a non-Websocket server. This causes the WebSocket handshake to fail. When this happens, Firefox introduces delays between consecutive attempts. This delay reduces drastically the number of requests per second that the browser issues. Then, the Server-Sent Event API can produce 210 and 250 requests per second for Chrome and Firefox, respectively. Last, the Image API can generate about 80 and 750 requests per second. Table 1 shows also that the number of SYN packets rate is about the same for HTTP requests. This behavior is caused by the fact that the server does not use persistent TCP connections with the browser. This means that the server terminates the TCP connection after sending the HTTP response. However, if the server supports persistent connections, the number of SYN packets per second reduces drastically[1]. In our experiments, we observed that the number of SYN packets reduces of about x13 for the XHR, SSE, and Image API. However, in the case of WS the number of SYN packets per second remains the same. This is caused by the errors during the creation of a web socket. As mandated by the WebSocket protocol, the party detecting the failure terminates the TCP connection. As opposed to the frequency of SYN packets, the number of sent requests slightly increases. For example, in about 70% of our experiments the number of requests per second increases of a multiplicative factor between x1.01 and x2.

When the TCP port is either in a packet drop or reject state, the requests and SYN rates are negligible. However, when the kernel sends TCP RST packets for closed TCP ports, our the JavaScript APIs almost consistently exhibit a very aggressive behavior. Even as compared to the "Open" state, the SYN packets frequency is significantly higher. This is caused by the fact that browsers typically limit the number of parallel connections, whereas the connections (due to the RST) early leave this stage—and new connection attempts are established. It is worth to point out that SYN floods are not really a security issue if the port is in a packet drop, reject, or reset state.

Finally, Table 1a shows that Firefox, in general, performs better than Chrome with speeds that are 2x and 9x faster than Chrome.

Table 1b shows the results of our test when no workers are not used. The column Avg. [Reqs/s] is the average HTTP request per second when calling 1000, 2000, 3000, and 4000 times per second the API functions. The column Max [Reqs/s] is for the maximum values. With exception of the WS API, Firefox achieved the highest request per second rate of our experiments. Firefox can sent requests per second with a rate of about 2,800 Reqs/s with the XHR API, 1,900 Reqs/s with SSE API, and 1,900 Reqs/s with Image API. These values are in average 7x faster than the speed of Chrome. By comparing the SSE results of Firefox between Table 1a and

[1]These results are not shown in Tables 1 and 2

		TCP port states				
		Open		Drop	Reject	Reset
		[Reqs/s]	[SYN/s]	[SYN/s]	[SYN/s]	[SYN/s]
XHR	Chrome	1,005.30	1,012.47	0.60	2.76	2,102.14
	Firefox	2,165.76	2,166.43	0.60	4.42	4,821.30
WS	Chrome	34.65	34.65	0.09	1.45	37.45
	Firefox	0.04	0.04	0.19	0.04	0.04
SSE	Chrome	210.69	211.12	0.60	2.82	529.27
	Firefox	258.69	259.60	0.20	0.91	912.09
Image	Chrome	84.60	84.65	0.63	2.73	161.40
	Firefox	751.15	751.21	0.60	5.43	2,237.81

(a) Results grouped by TCP port state. Values are the average values with 0, 2, and 3 web workers (except for the Image API), and with 1000, 2000, 3000, and 4000 API calls per second.

		TCP port state	
		Open	
		Avg. [Reqs/s]	Max [Reqs/s]
XHR	Chrome	1,359.59	1,886.33
	Firefox	1,456.74	2,892.03
WS	Chrome	58.31	73.47
	Firefox	0.12	0.13
SSE	Chrome	399.97	941.58
	Firefox	776.07	1,907.48
Image	Chrome	84.60	109.38
	Firefox	751.15	1,916.28

(b) Excerpt of results only for TCP port open and no workers.

Table 1: Aggressiveness of JavaScript APIs as implemented by Chrome and Firefox.

Table 1b, it emerges that the average number of requests per second in Table 1b is considerably greater than Table 1b. This difference is caused by the fact that in Firefox the Server-Sent Event API is not available within Web Workers.

Table 2 shows the results with different web workers. With the XMLHttpRequest API, Chrome and Firefox present an opposite behavior. While with the increase of workers Chrome decreases the number of requests per second, Firefox slightly increases the packet rate. Chrome also exhibits a similar behavior with the WebSocket and SSE API. As said before, Firefox has negligible request rates with WebSocket, whereas with SSE can generate about 800 Reqs/s without workers. As explained before, Firefox does not allow the access to the SSE API within web workers. As a result, the number of requests per second is zero. Finally, the results with the Image API are the one showed in Table 1b with no workers. According to the Web Worker specifications, web workers have no access to the DOM which includes the Image interface. As a consequence, we did not perform tests with web workers.

2.2.3 *Further Attack Features and Filter Evasion*

DDoS attacks are more effective if adversaries can hide their malicious traffic within benign traffic. Whereas a traditional DDoS bot has all the flexibility to generate HTTP(S) traffic, this is not the case for browsers. In this section, we will discuss if (and how) evasion techniques could be implemented.

Arbitrary `Referer` and `Host` headers—One of the DDoS commands of traditional bots is issuing HTTP requests with custom-chosen `Referer` and `Host` header. JavaScript programs can modify HTTP request headers only with the XMLHttpRequest API. However, the JavaScript program cannot modify all the HTTP request headers. There is a blacklist of headers that *cannot* be modified, including `Referer` and `Host`. This may leave defenders are valuable angle to characterize malicious communication.

Requests with no Response—Second, some DDoS bots feature an attack type that requests resources via HTTP, but does not wait for the responses. The motivation behind this CPU or memory exhaustion attack is that the server has to fetch the requested resources (which may be large), and the client does not need to receive it. We thus inspected if JavaScript code can interrupt the TCP socket before the HTTP response is entirely received. A bot can interrupt a TCP socket in different ways (e.g., by sending an RST or by dropping incoming TCP packets without acknowledging their reception) and in different moments (e.g., before the reception of the HTTP response or after the reception of the first packet of the response). In this section, we details these two aspects.

		Workers	[Reqs/s]	[SYN/s]
XHR	Chrome	0	1,359.59	1,370.11
		2	966.69	973.51
		3	689.63	693.80
	Firefox	0	1,456.74	1,456.66
		2	2,424.13	2,425.50
		3	2,616.40	2,617.14
WS	Chrome	0	58.31	58.31
		2	29.30	29.30
		3	16.33	16.33
	Firefox	0	0.12	0.12
		2	0.00	0.00
		3	0.00	0.00
SSE	Chrome	0	399.97	400.92
		2	155.05	155.27
		3	77.05	77.19
	Firefox	0	776.07	778.81
		2	0.00	0.00
		3	0.00	0.00
Image	Chrome	0	84.60	84.65
	Firefox	0	751.15	751.21

Table 2: Aggressiveness correlated with the number of workers. The average number of requests is calculated between the values when invoking 1000, 2000, 3000, and 4000 times per second the API calls.

While a traditional bot has direct control of TCP connections and it can terminate them in many ways (e.g., TCP RST, TCP FIN), a client-side JavaScript program cannot directly setup TCP connections[2] and it relies on high-level communication APIs which

[2]The W3C is working on a draft to standardize TCP and UDP sockets [14]. Browsers supports TCP and UDP sockets however, their access is limited to extensions or to privileged external applications [15, 16].

abstract away the details of the underlying TCP connection. These APIs provide primitives to abort a request or to close the connection. For example, the XMLHttpRequest API allows to abort an XHR request via the `abort` function. Similarly, the WebSocket API and the Server-Sent Event API have a `close` function. Protocol and API specifications do not mandate the specific technique to terminate a TCP connection upon the call of these functions. However, in our experiments we observed an uniformity of behavior between Chrome and Firefox—both browsers terminates the TCP connection with a RST packet.

After clarifying *how* to close connections via JavaScript, we now elaborate *when* this can be done. A traditional bot can terminate the TCP connection at any point in time, e.g., right after sending the last TCP packet of the HTTP request, or right after receiving the first TCP packet with the HTTP response. In contrast, the JavaScript communication APIs do not allow a direct way to control in which point the connection can be closed. However, an attacker can control disconnects by scheduling timeouts (e.g., via `setTimeout`). While a short timeout can cause to reset the connection right after the TCP handshake, a longer timeout can cause the browser to receive the entire HTTP response. In order to send the RST packet in the right moment, an attacker may need to estimate the timeout by monitoring the response time of the target.

IP Spoofing—IP source address spoofing is frequently used by attackers to hide their identity or to launch amplification attacks [17]. While potentially possible for a traditional DDoS bot, it is not possible to send IP-spoofed traffic via JavaScript, though.

3. DDOS BOTNET ECONOMICS

In this section, we will measure the costs of running browser-based as compared to the costs of traditional malware-infected botnets. This will help to understand if the attackers may have an economical incentive to resort to browser-based DDoS attacks (as opposed to buying malware installations).

3.1 Costs for Browser-Based Bots

For our preliminary measurements, we deployed four advertisements in the Google Display Network from May 10-17, 2015. We explicitly followed a conservative model in the sense of simple advertisements and a non-sophisticated attacker strategy. Using this approach we gain insights into the ad network without assuming experienced attackers, which is in line with our perspective of attacks for the mass.

Each advertisement includes HTML or JavaScript code to request resources from an external monitor server, in detail:

Ad 1 requests a URL in the static structure of the HTML page;

Ad 2 requests a URL via the JavaScript interface of HTML tags;

Ad 3 requests every five seconds a resource as in Ad 2;

Ad 4 sends a single content request using API designed for communication, i.e., the XMLHttpRequest API.

The different access mechanisms allow us analyze both the local configuration of the users as well as protection mechanisms of the ad network. Ad 3 enables us to measure the session time, i.e., how long a user stays on the site that shows the advertisement.

Deployment Experiences—Advertisement are verified by Google before they are officially published. Our advertisements have been accepted within 30-40 minutes. Deploying a malicious ad campaign is thus possible on short notice.

To evaluate the complexity for an attacker to inject a malicious advertisement, we tried to derive a basic understanding of the verification process, i.e., if the verification is handled manually or automatically. For this we uploaded complementary advertisements that are copies of Ad 1-4 but refer to a different landing page in case a user clicks the ad. This landing page reflects the content of the original landing page but makes the content invisible. These incorrect advertisements have also been accepted. We suppose that the verification process is performed by rather simple processes such as pattern matching rules. Any advanced check, in particular a verification by a human, could easily reveal our trap.

Client Statistics—Over one week of measurements, the ad network generated 32,932 requests to our external server. Those requests result from only presenting the embedded advertisement on a customer page of the ad network. In addition, we measured 174 requests that result from clicks. It is worth noting two observations. First, Google Display follows a pay-per-click (ppc) model, leading to very low costs of $\approx$\$1.23 in our case, i.e., four advertisements initiated overall 33k requests to an external server. Second, we could easily increase the number of requests by changing our HTML/JavaScript code. An increased number of requests will increase the attack potential but not affect the costs as requests initiated by our ads are independent of clicks.

Surprisingly, the number of requests varies significantly per day and advertisement type (HTML, JavaScript, and XHR). Content is not loaded via XHR requests (Ad 4), and content requests using plain HTML (Ad 1) is more evenly distributed (cf., Table 3).

In the next step, we focus on dedicated users by analyzing client IP addresses in more detail. Figure 1 shows how long a client was viewing an advertisement on average, based on the data Ad 3 creates. The box plot visualizes the mean (square), median (line), and the 25- and 75-percentiles of the gathered data. Note that we cut the y-axis for visibility reasons. The maximum value for May 10 is 785 minutes.

Overall, a significant distribution among the clients is visible, which is not surprising for two reasons. First, users behave quite differently when viewing web content. Second, when a user changes a web site depends also on the presented content (e.g., news site versus search website). However, in our current setup we cannot control on which website the advertisement is embedded.

We compared the number of sessions per client with the number of impressions provided by Google and found that Google indicates much higher number of visits. It is rather unlikely that this is due to ad blockers because those tools use black lists and thus do not prefetch code. For Ad 2-4 this might be due to disabled JavaScript at the client-side, which then leads to less external requests at our monitoring server. For Ad 1, which is using plain HTML, we would expect less deviation. Using the Google impression statistics for the estimation of the attack impact leads to overestimated results. This observation nicely illustrates that the design of our methodology (i.e., relying on an external monitor) was crucial.

More surprisingly is that the number of unique clients heavily depends on the day (cf., Table 3). For an attacker, this complicates predictions about the size of the botnet.

Finally, we analyze the distribution of the geographic location of the clients using the MaxMind IP to country mapping. Around 80% of the IP addresses viewing our ads are assigned to Russia. Among the remaining top ten countries are also Germany, Switzerland, UK, and France. All of these countries provide good Internet connection, which will allow the attacker to initiate even large volume content access.

Our current results can be considered as the minimum attack potential, which is already high. Only less than 1% of the users of

	External Requests [# GET and HEAD]					Clients [# Unique IP Addresses]				Budget [$]				
Day	Ad 1	Ad 2	Ad 3	Ad 4	Sum	Ad 1	Ad 2	Ad 3	Ad 4	Ad 1	Ad 2	Ad 3	Ad 4	Sum
05/10	243	2	24,076	0	24,321	122	2	69	0	0.05	0	0	0	0.05
05/11	232	0	182	0	415	116	0	2	0	0	0	0	0	0
05/12	262	1	3129	0	3,399	169	1	3	0	0.23	0	0	0	0.23
05/13	2,170	8	80	0	2,252	774	3	5	0	0.59	0	0	0	0.59
05/14	1,112	2	459	0	1,573	759	2	2	0	0.07	0	0	0	0.07
05/15	515	0	0	0	515	384	0	0	0	0.05	0.03	0	0	0.08
05/16	412	2	0	0	414	318	2	0	0	0.06	0	0.02	0	0.08
05/17	43	0	0	0	43	40	0	0	0	0.11	0	0.02	0	0.13

Table 3: Overview of attack potential per advertisement and day measured at our external monitor.

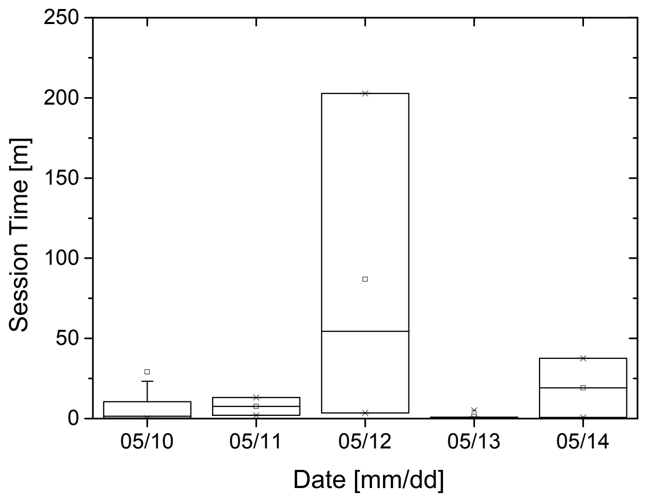

Figure 1: Statistical overview: Duration in minutes a client sees Ad 3

Ad 3 click the advertisement, where each click costs ≈1 cent. The attacker is able to attract up to 69 users per day. In the best case a web client was under the control of the attacker for up to 13 hours. These preliminary results indicate that an attacker may achieve much higher impact with more sophisticated malicious advertisements. On the client recruiting side, an attacker could try to create less attractive ads to reduce the probablity of clicks (i.e., costs) or try to control ad replacement with respect to more frequently visited pages. On the ad programming side, the attacker could simply increase the number of initiated requests. By comparing Ad 1 with Ad 2 and 3 we already found that the amount of clients differs. Implementing a much more aggressive request scheme in Ad 1 could lead to higher attack potential but we do not have experiences how the ad network react on this. We will focus on a more complete anatomy of ad networks in future work.

To summarize, we found varying viewing behaviors for the ads we injected. When computing the costs for brower-based botnets, we focus on the results of Ad 3, as this advertisement allowed us to track the viewing time of the instrumented clients. For the accumulated online time of all clients (2,327 minutes), this specific ad cost $0.04. On average, an attacker has to pay a risk budget of $0.02 per day and source.

3.2 Costs for DDoS Malware

Previous studies found out the price for malware installations in the underground range between $6 and $140 per 1000 installations [18]. Still, it is unclear for how long an attacker can abuse these bots once he bought them. We thus measured how long a traditional malware-infected bot would stay online. To this end, we leverage our data set from our infiltration of the Zeus P2P botnet with sensors in October 2013. This data set constitutes one of the very few sources to measure the uptime of malware-infected hosts, in particular since Zeus-infected host have a unique identifier that allows us to track individual bots. Albeit Zeus P2P has not been used for DDoS attacks frequently—in fact it had the capability to perform such attacks—we assume that populations of other botnets behave similar. On average, a bot stays online for 11.9 hours per day, i.e., about half a day. In addition, we observed that 63.5% of the bots are still infected after 1 week. This is along the lines of our previous observation that the Zeus P2P botnet population fluctuates about 5% [19] per day. We thus estimate that—with a single infection—a bot remains operative for about 20 days, resulting in about 10 days of an online bot. Combining both observations, we conclude that traditional malware costs between $0.0006 and $0.014 per day and source—assuming of full utilization of the bot whenever it is online, and considering infection costs of $0.006 and $0.14 per bot.

3.3 Economics Analysis

We now compare the costs for the deployment of malicious ads with the deployment of traditional malware by a brief back-of-the-envelope calculation. In fact, we found that the costs for both botnets are comparable—between $0.006 and $0.014 per day and attack source. Browser-based botnets are cheaper than infections in high-cost countries (like the US), but are more expensive than botnets in countries for which pay-per-install (PPI) installation cost less [18]. However, our economic analyis is clearly limited. First, we only compared the prices of one PPI network with one ad network. Second, attacks may actually build up DDoS botnets for free (e.g., by infecting embedded devices with default logins). Last, we did not try to improve our ad to make it less attractive to being clicked on in order to reduce the pay-per-view price. Still, we show that the costs are largely similar.

However, the *functionality* of browser-based bots is limited compared to traditional bots, as the common web API exposes less functionality. For example, malware may monetize in more ways than just DDoS attacks (e.g., ID theft or spamming), whereas browser-based botnets are most suitable for DDoS. Then again, considering that new web technologies, such as WebRTC, offload system functionality into the web browser, we can expect a rich set of interfaces in the near future. Having a feature set comparable to malware within a browser will increase the revenue of browser-based botnets.

Finally, the *level of control* for browser-based DDoS botnets is limited. Most importantly, ad campaigns introduce a delay between issuing and viewing the ad, whereas an attack using a DDoS botnet can be started immediately via C&C commands. Another drawback of ad networks is that they are less predictable how many bot clients are recruited, and the number of simultaneously-running bots is low. Our preliminary results showed that there is room for optimization, e.g., by making the ad more attractive to be displayed by investing higher ad costs or spreading the ad among multiple ad networks.

4. CONCLUSION AND OUTLOOK

We have discussed browser-based DDoS botnets, a serious threat to the Internet. We have shown that the attack does not introduce higher costs at the side of the adversary. Instead, the attacker model is in line with requirements (expertise, money, etc.) of our threat model. However, we have also shown certain limitations to browser-based botnets, both regrading the attack flexibility and the way the bots can be controlled. In the following, we will nevertheless discuss ideas to mitigate some of the problems of DDoS-based botnets. We will finally conclude this paper with an outlook to future work.

4.1 Attack Mitigation

Rate Limiting—We analyzed for two common browsers (Chrome, Firefox) how many media items are allowed to be loaded in parallel (e.g., `img src=""`). All of them had a limit of six but we also found that loading via JavaScript is less limited. Such a limit needs always to be considered with respect to the quality of experiences for a user. It is very likely that this limit will be increased in the future, in particular with an increased deployment of multipath transport.

Partial Cross-Origin Resource Sharing—Disabling cross-origin resource sharing (CORS) prevents a client from loading resources located under a different origin than the origin of the webpage. All modern web browser allow for cross-origin resource sharing by default, at least for non-AJAX content. However, due to Content Delivery Networks (CDNs), and due to the tendency to of external resources (e.g., CSS), the web heavily requires support of CORS—disabling CORS would be too restrictive.

A compromise might be partial CORS. The current CORS mechanism requires interaction between client and server, where the server signals legitimate cross domains and the browser might prevent content rendering. This mechanism implies the drawback that the client still sends a request to the server. Here we propose the idea of a local decision by the client. Instead of allowing requests for arbitrary origins, one could allow only requests to domains under the same administrative control. For example, a client would request content from `youtube.com` embedded into a page in the origin `google.com`, as both domains are managed by the same operator.

The verification if two domains belong to the same operator can be implemented by the client using DNSSEC. Having such a name-based attestation infrastructure in place, a client can check if two different names have been signed by the same private key, which belongs to the operator. Note that it is common practice among large DNS operators to use the same zone/key signing keys for different zones.[3] Local DNS caching will help to reduce overhead.

Server-Side Filters—Finally, given the limited flexibility in changing the HTTP requests, browser-based DDoS attacks can be identified as such by filters. For example, the `Origin` header was present both in the attack traffic by Great Cannon as in our test attack traffic. This header reveals the server that instructed the client to issue the HTTP request towards the victim, and is thus descriptive—especially in case the attacking code is loaded from a single server only (e.g., a single typosquatting domain, or a single ad network). To the best of our knowledge, the attacking code cannot overwrite the `Origin` header with an arbitrary URL. Still, such filters may be too coarse-grained (e.g., blocking requests from entire ad networks) and may also block benign clients. In addition, if ad networks allow ads to be loaded from any external hosts, then the `Origin` can be chosen by the attacker—while still requiring multiple hosts or domains to vary the value.

Furthermore, servers can deploy rate limiting based on the HTTP `Referer` header values. This header is inserted by the web browser and frequently used. However, this might change in the future, as the `Referer` conflicts with privacy concerns. Second, the browser must not send a `Referer` field if the previous page was accessed via HTTPS [20]. Third, with HTML 5, a website may include an attribute that instructs the browser not to send the `Referer` field.

[3]For detailed discussion among operators about this topic, we refer to `http://lists.dnssec-deployment.org/pipermail/dnssec-deployment/2010-March/003704.html`.

4.2 Future Work

In the future, we aim to improve our measurement on the attack economics. We will add experiments of other ways to acquire browser-based bots, such as typosquatting. In addition, to improve the statistical significance of our cost estimations, we will expand our measurements to multiple PPI and ad networks. Second, we intend to analyze the potential of a peer-to-peer control layer between web browser based on WebRTC. Third, we will investigate the solution space to the browser abuses. So far, our ideas are just hypothetical, and we plan to design more detailed schemes and thoroughly evaluate them.

5. REFERENCES

[1] B. Marczak, N. Weaver, J. Dalek, R. Ensafi, D. Fifield, S. McKune, A. Rey, J. Scott-Railton, R. Deibert, and V. Paxson, "China's Great Cannon," Citizen Lab, University of Toronto, Technical Report, April 2015. [Online]. Available: https://citizenlab.org/2015/04/chinas-great-cannon/

[2] J. Grossman and M. Johansen, "Million Browser Botnet," in *Presentation at Black Hat USA 2013*, 2013.

[3] L. Kuppan, "Attacking with HTML5," in *Presentation at Black Hat 2010*, 2010.

[4] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis, "Puppetnets: Misusing Web Browsers As a Distributed Attack Infrastructure," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06, 2006.

[5] J. Szurdi, B. Kocso, G. Cseh, J. Spring, M. Félegyházi, and C. Kanich, "The long "taile" of typosquatting domain names," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* Berkeley, CA, USA: USENIX Assoc., 2014, pp. 191–206.

[6] P. Agten, W. Joosen, F. Piessens, and N. Nikiforakis, "Seven months' worth of mistakes: A longitudinal study of typosquatting abuse," in *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015).* Internet Society, February 2015. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/471369

[7] A. Welzel, C. Rossow, and H. Bos, "On measuring the impact of ddos botnets," in *Proceedings of the Seventh European Workshop on System Security*, ser. EuroSec '14. New York, NY, USA: ACM, 2014, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/2592791.2592794

[8] A. Büscher and T. Holz, "Tracking ddos attacks: Insights into the business of disrupting the web," in *Proceedings of the 5th USENIX Conference on Large-Scale Exploits and Emergent Threats*, ser. LEET'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228340.2228351

[9] A. van Kesteren, J. Aubourg, J. Song, and H. R. M. Steen, "XMLHttpRequest Level 1," http://www.w3.org/TR/XMLHttpRequest/, 2014.

[10] I. Fette and A. Melnikov, "The WebSocket Protocol," https://tools.ietf.org/html/rfc6455, 2011.

[11] I. Hickson, “Server-Sent Events,” http://www.w3.org/TR/2009/WD-eventsource-20091029/, 2009.

[12] I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O’Connor, and S. Pfeiffer, “A vocabulary and associated APIs for HTML and XHTML,” http://www.w3.org/html/wg/drafts/html/CR/embedded%2Dcontent%2D0.html#dom%2Dimage, 2014.

[13] L. Developers, “Lighttpd,” http://www.lighttpd.net/, 2015.

[14] C. Nilsson, “TCP and UDP Socket API,” http://www.w3.org/2012/sysapps/tcp-udp-sockets/, 2015.

[15] Mozilla Developer Community, “TCPSocket,” https://developer.mozilla.org/en-US/docs/Web/API/TCPSocket, 2015.

[16] Google Inc., “Network Communications,” https://developer.chrome.com/apps/app_network, 2015.

[17] C. Rossow, “Amplification Hell: Revisiting Network Protocols for DDoS Abuse,” in *Proc. of NDSS*. Internet Society, 2014.

[18] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, “Measuring pay-per-install: The commoditization of malware distribution.” in *Proc. of USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2011.

[19] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich, and H. Bos, “P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets ,” in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)* , San Francisco, CA, May 2013.

[20] R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” IETF, RFC 7231, June 2014.