Why Web Servers Should Fear their Clients
Abusing Websockets in Browsers for DoS

Juan D. Parra Rodriguez and Joachim Posegga

Institute of IT-Security and Security Law
University of Passau, Innstraße 43, Passau, Germany
{dp,jp}@sec.uni-passau.de

Abstract

This paper considers exploiting browsers for attacking Web servers. We demonstrate the generation of HTTP traffic to third-party domains without the user's knowledge, that can be used e.g. for Denial of Service attacks.

Our attack is primarily possible since Cross Origin Resource Sharing does not restrict WebSocket communications. We show an HTTP-based DoS attack with a proof of concept implementation, analyse its impact against Apache and Nginx, and compare the effectiveness of our attack to two common attack tools.

In the course of our work we identified two new vulnerabilities in Chrome and Safari, i.e. two thirds of all browsers in use, that turn these browsers into attack tools comparable to known DoS applications like LOIC.

1 Introduction

The last two decades of Web technology were governed by making Web browsers more powerful, thus increasing the computational power at the "edges" of the Internet. In particular, the power of JavaScript within the browser has grown immensely since its inception. Although this is indeed very profitable for Web developers, the potential for abuse of clients has also grown: at the moment, there are browser compatible JavaScript libraries for controversial purposes, such as bitcoin mining [7, 19], cracking of cryptographic hashes [20, 2], port-scanning [16], TOR network bridging [9], and even an attack that can use the users' disk space beyond reasonable limits [6, 1]. These are some examples where functionality is pushed beyond its "intended" behaviour.

In spite of how worrisome the aforementioned examples may be, this paper has a narrower scope and is limited to analysing a particular attack misusing browsers, namely, Denial of Service (DoS) against regular Web servers. This attack has an increased surface in comparison to existent WebSocket-based DoS attacks, which targeted only servers implementing the WebSocket protocol[26, 15]. The main contributions of this paper can be summarized as follows:

- Assess protection mechanisms from browsers against malicious code using WebSockets to perform DoS attacks against third parties. Our analysis led

us to discover two previously unknown vulnerabilities affecting two famous browsers.
- Evaluate and compare the impact of our attack executed in different browsers by measuring network traffic and connections on the server side. The results are further compared with two native tools doing HTTP and SYN flood DoS attacks: LOIC [18], and Syn-GUI [10].

At the time of writing, browser statistics [28] indicate that 64.9% of users utilize Chrome (or Chromium), while Safari is used by 3,8% of the market. These are the browsers most affected by our findings, so roughly two thirds of the browsers' population can be employed to execute the DoS attack as presented in this paper. Furthermore, every browser supporting WebSockets can already contribute to the DoS attack although to a lesser extent.

The rest of the paper is organized as follows. First, we explain the details of our attack in Section 2. Next, we describe our attack in Section 3. Section 4 shows the physical set up, measurements acquired and the discussion of the results. Then, related work is covered in Section 5. Finally, we draw conclusions from our research in Section 6.

2 Attack Details

DoS attacks from browsers scale very well since malicious content delivered to the client can be rather small, and it is delivered only once. As shown in Figure 1, as soon as the content is conveyed, the Web browser will open as many connections as possible to the third-party domain (i.e. victim) without the end-user noticing any of it.

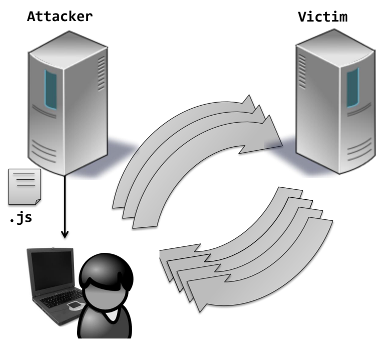

Fig. 1. Attack

Browsers can spawn threads to execute heavy computation in the background called WebWorkers [12]. We spawn 4 WebWorkers in the attack. Inside each WebWorker, a function opening the socket is called 500 times, although it could also be called endlessly if desired. In our particular case, the number of threads and the number of sockets were selected after testing which configuration performed best across different browsers. The code for each WebWorker should resemble the following snippet:

```
var j = 0;
for(j=0; j<500; j++)
{
  socket();
}
```

Due to the asynchronous nature of JavaScript, an infinite recursion is required to keep a connection open at all times. To implement this, a callback is provided to the socket's close event handler so the socket creation function is called again recursively. This is an example attacking `victim.domain`.

```
function socket(){
        var wsUri = "ws://victim.domain";
        var websocket = new WebSocket(wsUri);
        websocket.onclose = function(evt) {
                    socket();
     };
}
```

Although the WebSocket connection request is not really a “typical” GET request, it is enough to make the server reply back with the content of the Web site; therefore, using this code allows to successfully open HTTP connections with third-parties, i.e victim servers.

Aside from Chrome (also Chromium), and Safari, this attack also affects a more modest browser called Rekonq; nonetheless, measurements for this browser were omitted due to its low current participation on the browser's market.

3 Attack Prerequisites and Synergies

Analysing the aspects leading to the attack, and the facets intensifying its impact is of paramount importance. This allows to extract lessons learnt which can be capitalized in the future. In this section we discuss the two triggering factors for the attack, i.e. CORS and the lack of backward protocol compatibility. Afterwards, a synergy increasing the power of the attack is presented.

3.1 Cross Origin Resource Sharing

Cross Origin Resource Sharing (CORS) provides a mechanism to enable client-side sharing of cross-origin requests [14]. It is an opt-in mechanism empowering hosts to allow other domains to request their content through the browser. In general, when the browser loads a Web site (Origin1.com) as depicted in step (1) of Figure 2, and the loaded site (Origin1.com) contains JavaScript code attempting to acquire content from an external domain (External-domain.com), the browser must determine, by means of a preflight request, whether the third-party domain wants to serve HTTP requests commenced by Origin1.com.

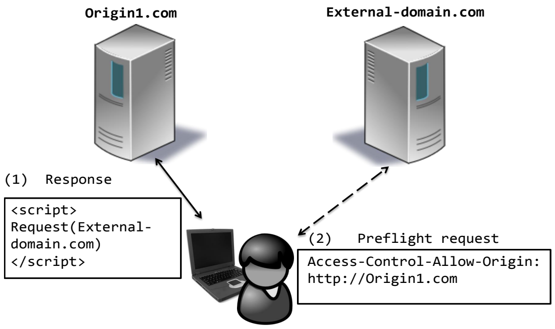

Fig. 2. Pre-flight request CORS

As shown in step (2) of Figure 2, the preflight request to the third-party domain (External-domain.com) gives information to the browser regarding the HTTP methods and origins allowed by the third-party server (External-domain.com). Subsequently, this information is used by the Web browser to decide whether HTTP requests should be sent to the third-party domain or not.

In theory, CORS would block DoS attacks against third-party domains, since it would require the victim to include the attacker's domain in their preflight request: a highly unlikely scenario. However, in practice, certain requests such as image requests, iframe content requests, or creation of WebSockets are not forbidden to avoid hindering functionality in Web sites. This yields the possibility to generate certain requests to third-party domains making not only DoS, but also other attacks such as port scanning possible.

3.2 Lack of Backward Protocol Compatibility

Huang et al. [13] have shown how the WebSocket handshake definition, as a sequence of HTTP messages, brings security problems. Although not related to DoS attacks, Huang et al. discovered that an HTTP header misinterpretation allowed to poison the cache of transparent proxies in the network. This attack forced every proxy client to load malicious content delivered by the attacker. We exploited a different aspect of the WebSocket handshake, yet related to their work since it benefits from an HTTP header omission. The issue pertains to the first message of the WebSocket handshake, which is an HTTP GET request with additional headers (e.g. Upgrade, Connection, Sec-WebSocket-Key) [21].

Whereas the WebSocket specification contemplates how a WebSocket server deals with additional headers in the handshake, it disregards how a regular Web server would reply to such request. Currently, when a WebSocket handshake is sent to a plain Web server, the server will interpret the request as a regular GET request, ignoring additional headers, and sending the content of the main page back. In this way, an attacker can generate requests for content to third-party Web servers just by attempting to open a WebSocket with the host, even though they are not WebSocket servers.

3.3 Browser Vulnerabilities

Antonatos et al. [4] have used several data sources, ranging from Alexa ranking [3] to an instrumentation of their institution's Web site to study users' behaviour, in order to estimate from a theoretical perspective the impact of a DoS attack abusing Web browsers. Their work revealed that the mean time that users keep pages on their browsers open is around 74 minutes. They also conclude that abusing browsers is a real threat; especially, because according to their results, more than 20 percent of typical commercial sites could abuse 10.000 clients, and 4 to 10 percent of randomly selected sites can use more than 1.000 browsers.

We have discovered two new vulnerabilities, increasing the aggressiveness of a DoS attack, after testing the most known browsers supporting WebSockets: Chrome, Chromium, Firefox, Safari, and Opera. One vulnerability affects Chromium and Chrome, and another one affects Safari. In this context, even though browser vulnerabilities are not a prerequisite, they create a powerful synergy with the DoS attack. In the presence of a browser vulnerability, less popular sites would still constitute a powerful ally for an attacker because clients would generate more traffic than intended.

The Chrome (and Chromium) networking stack follows an asynchronous design philosophy for performance reasons. Chrome was designed to restrict the amount of connections generated by WebSocket handshakes. However, this verification was implemented synchronously. As a result, when the user left the malicious (or compromised) Web site, all the queued connections to third-party domains were opened. We reported this behaviour through the Chromium bug tracking site, i.e. the Open Source project in which Google Chromium is based upon. This issue has been fixed by Chromium developers in the Chromium master branch [22]. Also, we found, and reported through the apple's developers bug reporting site, that Safari does not cap the amount of connections generated by WebSocket handshakes per tab. For both vulnerabilities, using WebWorkers exacerbated the effect of the attack by opening several WebSocket connections concurrently.

4 Testing and Analysis

In order to assess whether our attack is feasible in the real world, its impact must be measured and compared to applications designed for DoS. In this section we describe the physical and virtual set-up used to confirm the feasibility of our attack, as well as the measurements obtained and the discussion around them.

4.1 Set up

As illustrated on Figure 3, one router (MikroTik RouterBoard RB750 Series) and four physical machines were used: three Dell Inspiron 15 with 6 GB RAM memory and with an Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz processor (in gray), and one Lenovo T430S with 16 GB RAM memory and with an Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz processor (in white). All the network connections used Ethernet cables to ensure a fast, and reliable physical connection between the physical machines. On every one of the Dell machines, a fresh installation of Kubuntu 13.10 saucy was made; also, an Ubuntu 12.04 TLS was installed in the Lenovo T430S hosting the attacker's Web site.

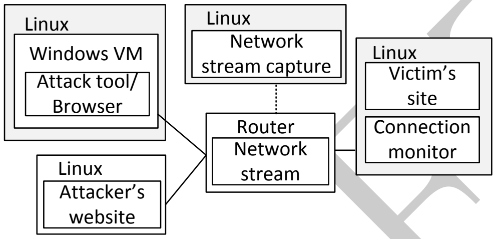

Fig. 3. Testing environment

One of the Dell machines running Kubuntu had a VirtualBox Version 4.3.18 r96516 with a virtual machine running Windows 7 Enterprise Service Pack1, where all the attacks were executed from. This machine was configured with a Bridged VirtualBox network adapter, 4 GB of RAM memory, and 1 CPU core without capping the execution. Inside this Virtual machine the following programs were used for benchmarking the DoS attack: LOIC version 1.0.7.42, and Syn-GUI version 2.0. The browsers compared to the DoS tools were executed on the same virtual machine using Chrome version 39.0.2171.95 Official Build, and Safari Windows version 5.1.7. The choice for the SYN-flood tool and Safari's version was stirred by our need to execute every attack on the same platform, i.e. the Windows virtual machine. Nevertheless, the same behaviour observed in Safari version 5.1.7, was also observed in newer versions running in Mac OSX systems.

Due to the popularity of Apache and Nginx [25] they were used to run the victim's Web site on one of Dell machines natively in Kubuntu. The Apache version was 2.4.6, and the Nginx version was 1.4.1. Both servers were used in their default configuration, delivering only static (HTML) content.

On the network side, to hinder the interference of network capturing with the measurements as much as possible, the router forwarded the traffic between the victim site and the host running the attack to an external host (above, connected with dashed line) for further offline analysis. The external host received the traffic encapsulated with TaZmen Sniffer Protocol (TZSP), a protocol designed to encapsulate network traffic over the wire. Afterwards, the external host stored the traffic at the end of the experiment. This configuration delivered better results than capturing and storing traffic inside the router.

On the victim's server, the total amount of TCP connections to port 80 (used by the Web servers) listed by the Operating System was stored every 100 milliseconds (ms) approximately. This connection count included not only the fully established connections (i.e. ESTABLISHED state), but also those which were in an intermediate step or momentarily unused, such as those with state SYN_RECV, TIME_WAIT, or FIN_WAIT. This approach was preferred because it was observed that depending on server implementation aspects, connections may be left inactive for longer or shorter periods depending on how the socket and thread pools are handled. Also, the efficiency of the server can affect the time required to fully establish or close the socket from the moment when a SYN, or FIN packet is received respectively.

4.2 Measurements

For each server, a series of measurements were conducted while they were attacked by two DoS tools and the two most affected browsers. Before each measurement, the server was restarted in order to avoid affecting server performance due to previous execution of attacks.

For Safari, Chrome, as well as for every DoS tool (i.e. LOIC and Syn-GUI), a 20 second attack was performed. In the case of Chrome, a massive number of connection requests was generated when the user left the Web site. As a result, the measurements for Chrome were performed by opening the malicious Web site, waiting for 20 seconds in the Web site, and then closing the tab. Naturally, in the case of Chrome, the aggressiveness of the attack is proportional to the amount of time spent on the malicious page.

From Figure 4 to Figure 9, the left Y axis represents the network statistics obtained by counting the number of packets in a 100 milliseconds timespan that matched certain criteria. The values plotted therein contain the following measurements: the amount of SYN packets sent by the client (black continuous line), the number of acknowledged connections from the server (red dots), and the amount of HTTP requests answered by the server successfully (gray impulses filling the curve). On the right hand Y axis, the number of connections on the server side is shown with a dashed blue line. Unfortunately, unlike the left Y axis, due to the divergence of the number of connections, the right Y axis scale has to be adapted accordingly from graph to graph.

From now on, each subsection will show the results for a given attack tool, such as Chrome, Safari or LOIC, for both target servers, i.e. Apache and Nginx. t the end, we dedicate a subsection to the results obtained when attempting the attack after a security patch has been applied on Chromium.

Chrome In Figure 4 and 5, when the user leaves the tab roughly at the 20th second of the measurement, there is a peak of SYN packets sent by the browser, i.e 100-200 packets/100 ms for Apache, and 100-150 packets/100 ms for Nginx. At this point, the Web server stops replying with content. This is evidenced by the lack of gray filling under the curve. Additionally, the server fails to acknowledge all the connections requested, which is visible because the values for the number of acknowledgements, represented as red dots, appear below the continuous black line depicting the number of connection requests, i.e. SYN packets.

Also, whereas the network traffic is similar in both captures for Chrome, i.e. Apache and Nginx, the amount of server side connections for Nginx is more than twice of the values observed for Apache, reflecting that Nginx opens more connections on the operating system level than Apache.

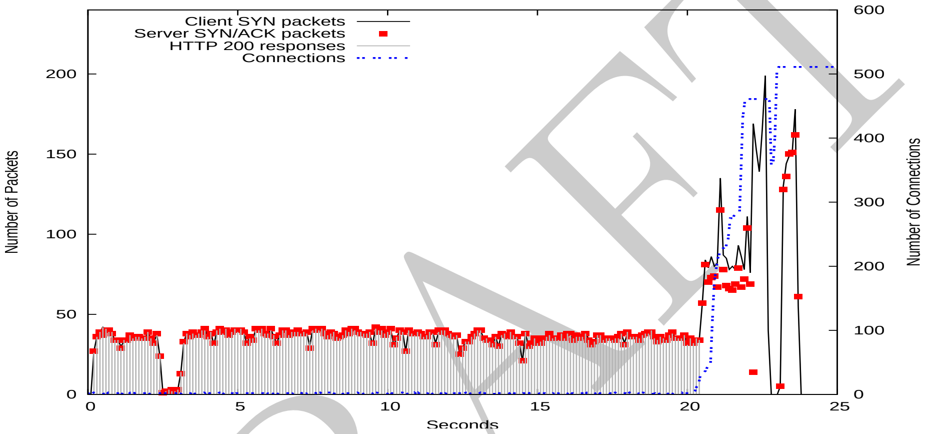

Fig. 4. Chrome-Apache

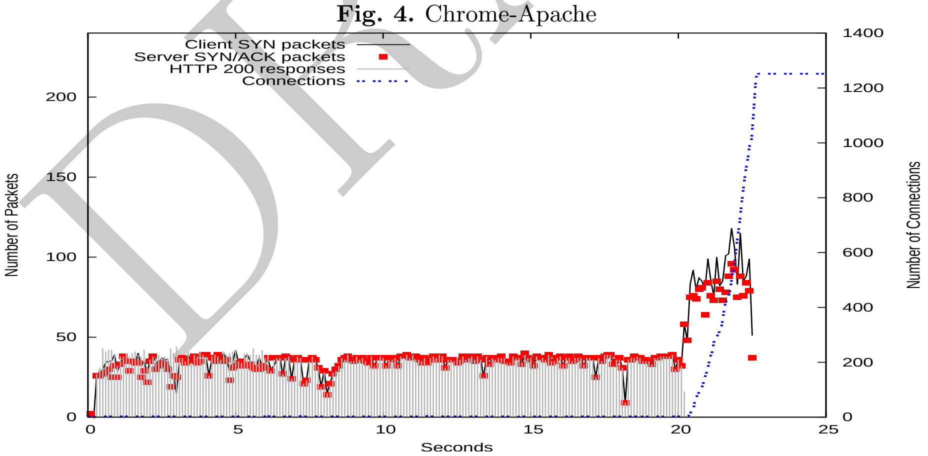

Fig. 5. Chrome-Nginx

Safari Safari can inhibit Apache from replying content from the beginning of the attack. It can be seen in Figure 6 that, due to the peak of SYN packets sent by Safari, the connections are not even acknowledged by the server, and the HTTP responses are scarce and severely delayed. Further, when comparing the two browsers for the Apache measurements, The number of SYN packets is slightly higher in the case of Safari, i.e. 150-250 instead of 100-200 packets/100 ms in the case of Chrome.

In the case of Safari attacking Nginx, peaks of SYN packets having 50-120 packets/100 ms force the server to stop replying to the HTTP requests temporarily, however, Nginx starts to provide HTTP responses afterwards. Eventually, it almost replies for every connection request. As expected, Nginx opens a higher amount of connections than Apache on the Operating System level, just like in the previous case.

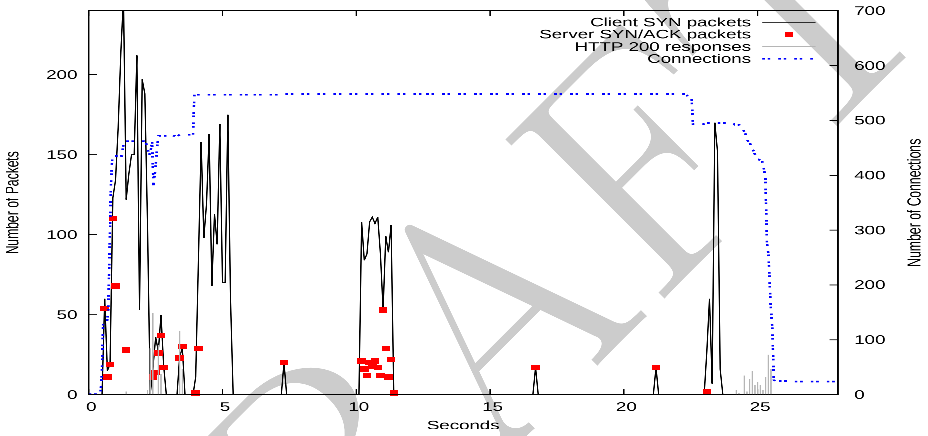

Fig. 6. Safari-Apache

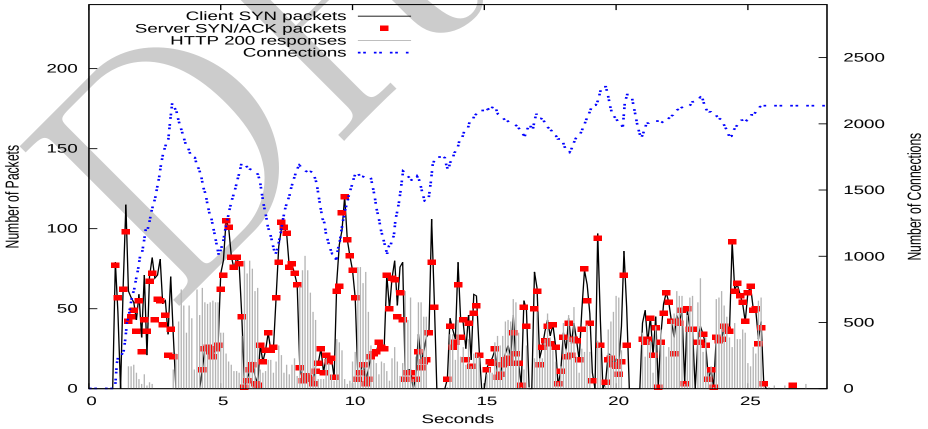

Fig. 7. Safari-Nginx

LOIC LOIC is a Windows application commonly used to perform DoS attacks. This tool can be compared to the browser attack because it attempts to flood the Web server with HTTP messages using a parametrizable amount of threads.

At first, we tested LOIC with 4 threads to have equivalent conditions to our attack from the browsers; however, the impact was significantly lower than using Safari or Chrome. As a result, this was modified to use the default configurations, i.e. 10 threads and wait for server response, while our attack had 4 WebWorkers, i.e. browser threads.

There are several differences in the network traffic when Chrome and Safari are compared to LOIC. For example, when the user leaves the tab in Chrome or when the attack is performed in Safari, the amount of SYN packets, and network traffic produced in general, is considerably higher than the traffic produced by LOIC. This situation allows Chrome and Safari to coerce Apache and Nginx to stop replying, which is not achieved by LOIC.

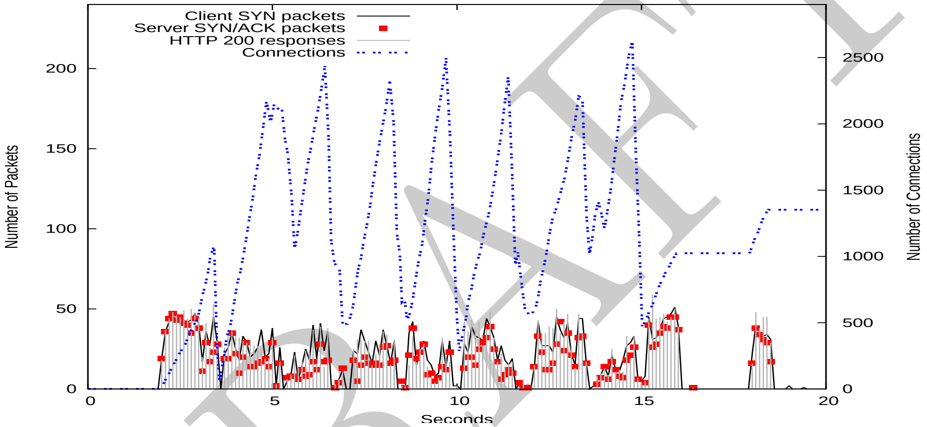

Fig. 8. LOIC-Apache

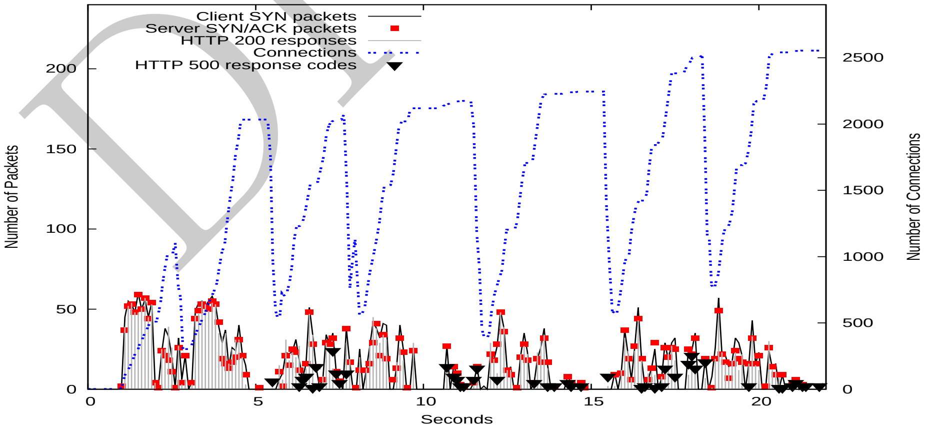

Fig. 9. LOIC-Nginx

However, LOIC has higher impact than the browsers on the amount of connections opened on the server side by Nginx and Apache. Moreover, in the particular case of LOIC's attack against Nginx, a special behaviour was observed. LOIC induces a concurrency problem for Nginx due to the amount of opened connections on the server side.

For Linux systems each socket is treated as a file descriptor [17], and in the particular case of LOIC against Nginx, the number of connections is consistently around 2500 connections for periods of up to 5 seconds. As a result, Nginx replies with an HTTP 500 error code, see black triangles in Figure 9, due to too many open file descriptors, which is evidenced in the log files.

Syn-GUI Syn-GUI implements a SYN flood attack by sending SYN packets massively to the server without implementing the HTTP protocol in the application layer, as previously mentioned attacks do. Comparing the browser attack to a SYN flood attack is relevant, since the strongest property observed for browsers was the power to generate a high amount of SYN packets during the initial hand shake of the TCP connection. As a result, the analysis presented hereafter focuses only on this aspect; also, because other measurements, such as the number of HTTP responses cannot apply to a SYN flood attack.

As it can be seen in Figure 10 and Figure 11, the peaks of SYN packets sent by Syn-GUI are still smaller compared to the peaks observed for Chrome and Safari. However, the amount of SYN packets is higher that the packets sent by LOIC.

Withal, it must be mentioned that certain countermeasures are effective against SYN flood attacks, but they fail to thwart LOIC or the browser attack. For example, SYN cookies avoid the allocation of resources on the server side until the TCP connection is actually opened. Although this would be effective against a SYN flood, it will not work neither against LOIC, nor against the browsers because the they actually open the TCP connections and send HTTP request to the server subsequently.

Fixed Browser We executed the trunk raw build of Chromium, including the security patch, to test the fix of the previously reported vulnerability. For this graph, both of the Y axes have been adjusted to a narrower scale in order to show the behaviour of the packets in the network and the connections opened. In Figure 12 and Figure 13, the amount of connections is controlled when the user leaves the tab. Further, the number of requests, and SYN packets in the network is not increasing beyond 8 packets/100 ms: A pretty low value, considering that it was around 40 packets/100 ms in the unpatched version, even before the user left the tab.

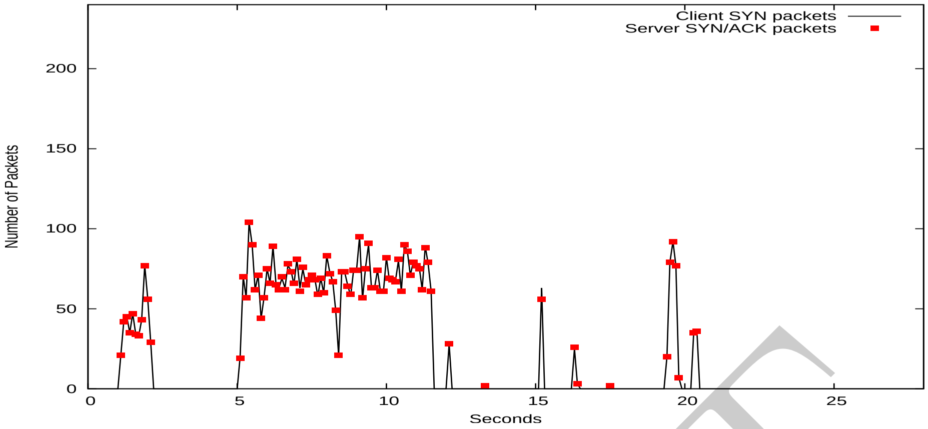

Fig. 10. SynGUI-Apache

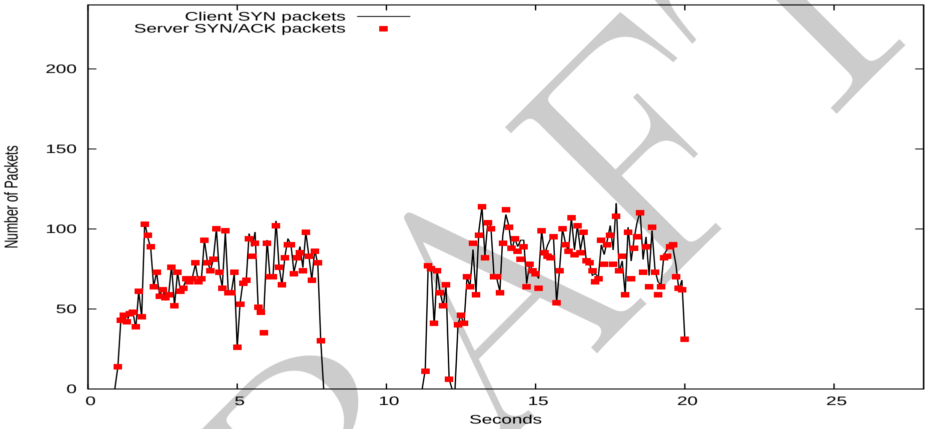

Fig. 11. SynGUI-Nginx

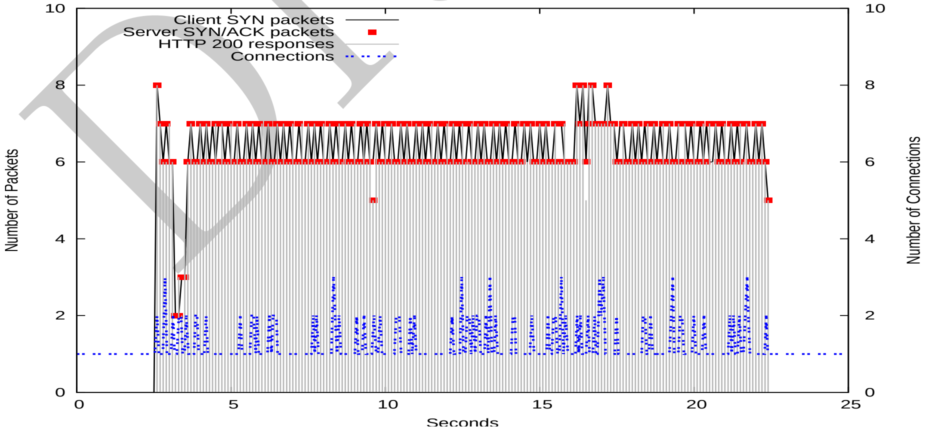

Fig. 12. Chromium Fixed-Apache

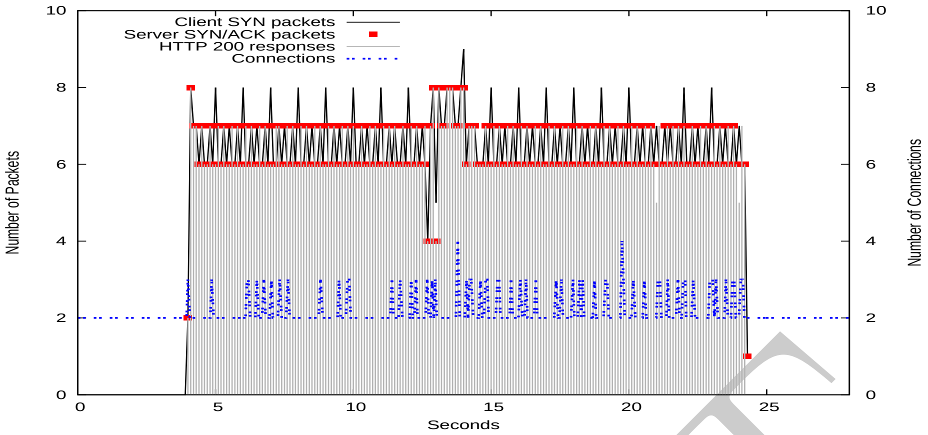

Fig. 13. Chromium Fixed-Nginx

4.3 Discussion

The comparison between browsers and the SYN flood tool demonstrated that the amount of connection requests generated from the browser is comparable to the number of requests generated by the SYN flood application. Nevertheless, abusing browsers has an additional advantage in respect to SYN flood: the browser implements the whole TCP, and HTTP network stack. As a result, countermeasures against browsers need to be more complex; for example, SYN cookies would thwart the SYN flood attack, yet they will fail against a Web browser attack.

Moreover, Safari and Chrome hinder the server from replying requests right after the attack is performed due to the sudden increase of TCP connection requests. To mitigate the impact from an attack exploiting browsers on the server side, configuring firewall rules to drop SYN packets after a certain amount of connections has been reached with given IP seems reasonable. However, these rules would have to be generated dynamically; especially, because different browsers execute the malicious JavaScript code while they visit the malicious Web site. This makes the set of browsers to be banned a constantly moving target, due to the high churn of visitors that a Web site has. Furthermore, if the browser attack is programmed in a less aggressive manner, recognition on the server side could be avoided, yet achieving the goal of the attack as long as the proper amount of browsers are available. This is the case of existent attacks [5, 4] which are described in more detail in Section 5.

LOIC excels in a different aspect. Although it does not coerce the server to stop replying like Safari or Chrome, is capable to force Nginx to reply with an internal server error as it opens more file descriptors than the limit allowed by the underlying operating system. This could be tackled by increasing the maximum limit for concurrent opened file descriptors in the server Operating

System; however, this only would cure the symptom. To solve this problem on the server side, specific business logic on how sockets and file descriptors are handled by Nginx would have to be modified.

So far, possible server-side countermeasures have been discussed. Now, we focus on client-side modifications which could help to address the problem. It has been previously shown, in Section 4.2, that the fixed Chromium version behaves mercifully with the server. Withal, It must be noted that although this diminishes the power of one single browser against a server, the problem of detecting when several browsers are colluding against a domain remains unsolved. We consider that employing machine learning techniques for malicious JavaScript code detection can help to detect properties of the DoS attacks [23, 8]. Further, a technique favouring early detection malicious code, such as the work presented Schütt et al. [24], may be the best match in order to stop the attack as early as possible. However, the main challenge is the identification of the proper features to process in the algorithms, as well as finding proper data sets, so learning algorithms can be trained and validated afterwards.

Last but not least, from the measuring perspective, collecting network traffic during a DoS attack characterizes it better than only counting TCP connections on the server side; however, this method still yields results specific to the server platform. This is unavoidable because there are a number of possible resources that could be exhausted, such as open ports, files opened by the server, CPU time, etc. However, common patterns observed allow us to conclude which are the strong and weak points of each attack and how they perform.

5 Related Work

Puppetnets was a term coined by Antonatos et al. for a botnet of browsers executing port-scanning, DoS, and worm propagation [4]. The authors described a simple DoS attack which did not exploit any implementation aspects of the browser. Still, they spent significant efforts on estimating the impact of an eventual attack. Their findings indicate that around 20 percent of commercial sites could be used to steer around 10.000 browsers, while the top-500 popular sites could leverage up to 100.000 browsers. They enumerate several possible countermeasures against the three aforementioned attacks, but they concluded that none of the presented options were completely satisfying. Athanasopoulos et al. performed measurements on the network level focused on the impact of RTT times in the generation of HTTP traffic from browsers, yet leaving any practical DoS evaluation of the scope of the paper. Instead, we focus on the validation of the attack by demonstrating its impact on real world server implementations, and the generation of traffic from one browser.

Grossman et al. [11] presented that it is possible to use browsers to conduct a DoS attack using Ad Networks to deliver the malicious JavaScript code, yet spending a small amount of money. Furthermore, although the DoS attack is already feasible due to the wide reach of an Add Network, its power can be further increased when there is a vulnerability on the browser's side. Grossman et al. discovered a vulnerability in Firefox. The exploitation consisted of using the browser's JavaScript API to attempt to load an image, but changing the HTTP scheme to FTP in the url. This allowed an attacker to create a higher amount of requests. Instead, we use the WebSocket API combined with the spawning of WebWorkers to generate requests from the browser, and we found a vulnerability affecting the handling of WebSocket handshakes for Chrome, and Chromium, and another one for Safari. Grossman et al. show the effectiveness of the attack by measuring the amount of HTTP connections on the server side for one server implementation, i.e. Apache. Instead, we monitored not only the established connections on the server side, but also the network traffic on the router level for two different server implementations, i.e. Apache and Nginx. This allows us to analyse more deeply the impact of each attack. Moreover, the comparison between common DoS tools and the proposed browser attack is also missing in the work presented by Grossman et al.

According to a recent technical report [27, 5], an anti-censorship project in China called GreatFire.org suffered a large scale DoS attack. The attack was executed from browsers of innocent Web site visitors located all around the world. The report includes the JavaScript code suspected of launching the attack. The presented code lacks of the mixture of WebWorkers and WebSockets presented in our work. In contrast to our approach, the discovered attack sends GET requests using the AJAX get function provided by jQuery. This shows that it neither employs WebWorkers, nor exploits cross-protocol or browser implementation problems to increase its power, as we do.

There is also existing work on executing DoS attacks using WebSockets from browsers [26, 15]; however, this kind of attack only targets WebSocket-enabled servers, and the existent work does not include detailed measurements. More to the point, our attack has an increased surface in comparison to the existent WebSocket-based DoS attacks, since it can be directed against any Web server, even if it does not implement the Websocket protocol.

6 Conclusion and Future Work

We showed in our paper how to turn modern Web browsers into attack tools by exploiting certain features of WebSockets. We measured the effectiveness of our attack, against two different Web server implementations, by combining server side measurements of the number of TCP connections with a network layer capture analysis. This allows us to confirm that the impact of the DoS attack from browsers is comparable, or in some cases more effective than using DoS tools.

The massive DoS attack discovered recently [27, 5] shows the significance of studying and understanding DoS attacks steering innocent browsers against victim servers, such as the attack discovered in our research. Also, Google Chromium's team decision to implement a security patch based on our vulnerability report reassures the relevance of the proposed attack, and it also enables our research to achieve real world impact. However, from a wider perspective, we consider that the criteria for moving functionality to the client side should be further researched, so future attacks can be prevented. For instance, if infinite trivial recursions including network operations such as the one presented would be forbidden, the attack would have been less powerful. Besides, preventing WebWorkers to open WebSockets, like Firefox does, would have also been a good countermeasure to limit the power of the attack. This proves, once again, that taking the least-privilege path when enabling functionality will always be the most secure approach.

Also, since the first the inception of DoS attacks from the browser [4], its detection has proven to be a difficult due to the number of possibilities to obfuscate, or dynamically modify and execute JavaScript code. We consider that future research efforts should study the feasibility of applying advanced algorithms, e.g. machine learning detection methods [23, 8, 24], for code property detection to assess their usefulness in this realm.

Finally, the risk of malicious code being executed transparently to the user could be mitigated by developing specific mechanisms within the browser. For instance, users could be empowered to control or monitor browser resource usage such as number of connections, number of spawned threads, etc.

7 Acknowledgments

The research leading to these results has received funding from the European Union's FP7 project COMPOSE, under grant agreement 317862. Further, the authors would like to thank Oussama Mahjoub for providing valuable insights during the collection of network captures.

References

[1] Introducing the new HTML5 Hard Disk Filler API. http://feross.org/fill-disk/, 2013.

[2] MD5-Password-Cracker. https://github.com/feross/md5-password-cracker.js/, 2013.

[3] The top 500 sites on the web. http://www.alexa.com/topsites, 2015.

[4] 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.

[5] Using Baidu to steer millions of computers to launch denial of service attacks. https://drive.google.com/file/d/0ByrxblDXR_yqeUNZYU5WcjFCbXM/view, 2015.

[6] Web code weakness allows data dump on PCs. http://www.bbc.co.uk/news/technology-21628622, 2008.

[7] Bitcoin Miner for Websites. http://www.bitcoinplus.com/miner/embeddable, 2011.

[8] 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, New York, NY, USA, 2010. ACM.

[9] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. Porras. Evading Censorship with Browser-based Proxies. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, PETS '12, pages 239–258, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] SynGUI. http://download.cnet.com/SynGUI/3000-18510_4-10915777.html, 2014.

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

[12] I. Hickson. Web workers. Candidate recommendation, W3C, may 2012. http://www.w3.org/TR/2012/CR-workers-20120501/.

[13] L.-s. Huang, E. Y. Chen, A. Barth, E. Rescorla, and C. Jackson. Talking to Yourself for Fun and Profit. *Proceedings of W2SP*, pages 1–11, 2011.

[14] A. v. Kesteren. Cross-Origin Resource Sharing. W3C recommendation, W3C, jan 2014. http://www.w3.org/TR/2014/REC-cors-20140116/.

[15] A. Kulshrestha. An Empirical study of HTML5 Websockets and their Cross Browser behavior for Mixed Content and Untrusted Certificates. *International Journal of Computer Applications*, 82(6):13–18, Nov. 2013.

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

[17] Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/select.2.html, 2015.

[18] A Network Stress Testing Application. https://sourceforge.net/projects/loic/, 2014.

[19] N. Matthews. jsMiner. https://github.com/jwhitehorn/jsMiner, 2011.

[20] N. Matthews. Ravan: JavaScript Distributed Computing System (BETA). http://www.andlabs.org/tools/ravan.html, 2012.

[21] A. Melnikov. The websocket protocol. RFC 6455, RFC Editor, dec 2011.

[22] A. Rice. Chromium Code Reviews Issue 835623003: Add a delay when unlocking WebSocket endpoints. (Closed). https://codereview.chromium.org/835623003, 2015.

[23] 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.

[24] 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.

[25] Web Server Usage Statistics . http://trends.builtwith.com/web-server, 2015.

[26] M. Shema, S. Shekyan, and V. Toukharia. Hacking with WebSockets. http://media.blackhat.com/bh-us-12/Briefings/Shekyan/BH_US_12_Shekyan_Toukharian_Hacking_Websocket_Slides.pdf, 2012.

[27] Internet activists blame China for cyber-attack that brought down GitHub. http://www.theguardian.com/technology/2015/mar/30/china-github-internet-activists-cyber-attack, 2015.

[28] Browser Statistics. http://www.w3schools.com/browsers/browsers_stats.asp, 2014.