Mitigating Browser-based DDoS Attacks using CORP

by

Akash Agrawall, Krishna Chaitanya, Arnav Kumar Agrawal, Venkatesh Choppella

in

Innovations in Software Engineering Conference, ISEC

Report No: IIIT/TR/2017/-1

Centre for Software Engineering Research Lab
International Institute of Information Technology
Hyderabad - 500 032, INDIA
February 2017
Mitigating Browser-based DDoS Attacks using CORP

Akash Agrawall
IIIT Hyderabad

Krishna Chaitanya
Telikicherla
Microsoft India

Arnav Kumar Agrawal
Carnegie Mellon University

Venkatesh Choppella
IIIT Hyderabad

ABSTRACT
On March 27, 2015, Github witnessed a massive DDoS attack, the largest in Github’s history till date. In this incident, browsers and users were used as vectors to launch the attack. In this paper, we analyse such browser-based DDoS attacks and simulate them in a lab environment. Existing browser security policies like Same Origin Policy (SOP), Content Security Policy (CSP) do not mitigate these attacks by design. In this paper we observe that CORP (Cross Origin Request Policy), a browser security policy, can be used to mitigate these attacks. CORP enables a server to control cross-origin interactions initiated by a browser. The browser intercepts the cross-origin requests and blocks unwanted requests by the server. This takes the load off the server to mitigate the attack.

CCS Concepts

• Security and privacy → Denial-of-service attacks;

Web application security;

Keywords

DDoS, Browser-based DDoS, Browser, Javascript, Cross-origin requests, MITM (Man in the middle)

1. INTRODUCTION

Since September 1996, several websites have witnessed various attacks on their availability [10, 12, 19, 29, 26]. They include DoS (Denial of Service) attacks, DDoS (Distributed Denial of Service) attacks and application layer DDoS attacks.

In a DoS attack, an attacker uses a single Internet connection to flood a server with fake requests. This overloads the server, rendering it incapable of serving requests to other users. There have been many cases of DoS attacks in the past [10, 20]. The common mitigation technique for these attacks is to blacklist the IP address of the device that generates the traffic.

A DDoS attack occurs when multiple devices that are distributed across the Internet flood the bandwidth of a targeted system. This is more dangerous than a DoS attack since the volume of traffic generated is multifold. Also, it is difficult to mitigate DDoS attacks by blacklisting IP addresses since the requests originate from a wide range of IP addresses. DDoS attacks can be very expensive for the victims. 49% of these attacks last between 6-24 hours [13]. There have been many cases of DDoS attacks in the past [11, 12, 19, 21]. One of the largest DDoS attacks launched in the history was against the website KrebsOnSecurity, a security news site, at a rate of around 620 Gbps on September 20, 2016 [21]. There was some indication that the botnet composed of “Internet of Things” (IoT) devices, such as routers, IP cameras.

An application layer DDoS attack is a type of DDoS attack in which the attackers use the application layer, layer 7 of OSI model, to launch the attack. In this paper we discuss a specific type of application layer DDoS attack, in which web browsers are used as vectors to launch the attack. We label these attacks as browser-based DDoS attacks.

By the design of web browsers, a request to a website could cascade a set of requests to multiple other sites. Such cross-origin (or cross-site) HTTP requests triggered without explicit user interaction are the root cause of several browser-based attacks. We have described the model of HTTP transaction and cross-origin request in detail in Section 2.1. Telikicherla et al. [34] classified the attacks possible due to cross-origin requests into two categories: exfiltration and web infiltration attacks. Figure 1 explains the difference about these categories of attacks. As shown in the figure a genuine website, say G.com, could now be compromised by an attacker who injects malicious content like an image tag pointing to attacker’s site, say A.com. A victim requesting the infected page could end up unwittingly participating in exfiltration, i.e., the leakage of private data to A.com. In a web infiltration attack, a request is initiated from an evil page to a genuine but unsuspecting server. Telikicherla et al. classified CSRF [1], clickjacking [31] and cross-site timing [22] as web infiltration attacks. On closer analysis it can be observed that a browser-based DDoS attack can be classified as a type of a web infiltration attack. Web infiltration is complementary to exfiltration.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM, ISBN 978-1-4503-4856-0/17/02... $15.00
DOI: http://dx.doi.org/10.1145/3021460.3021477

1 Internet Protocol (IP) address is a label assigned to a device in computer network.

ISEC ’17, February 05-07, 2017, Jaipur, India
Let us consider a hypothetical example of how a browser-based DDoS attack can be launched by an evil webpage loaded from A.com against a genuine server, G.com. The evil page contains the malicious JavaScript shown in Listing 1. When a user visits the evil page loaded from A.com, the JavaScript code executes in the user’s browser. This code triggers cross-origin requests unwittingly to G.com for every 100 milliseconds i.e., 10 requests per second (rps). This exemplifies how an adversary can launch browser-based DDoS attacks using JavaScript.

```javascript
function HitTarget () {
    var image = document.createElement("img");
    image.src = "http://www.G.com/img.jpg";
    image.id = "image";
    document.getElementsByTagName("head") [0].appendChild(image);
}
setInterval(HitTarget(), 100);
```

Listing 1: A malicious JavaScript that triggers requests to the server G.com for every 100 milliseconds, i.e. 10 requests per second.

Table 1 shows a comparison between the damages various DoS attacks can cause. In the table, **Attacking units**, refer to the number of servers that are used to generate the attack. In the case of a browser-based DDoS attack, **Attacking units** (Layer 2 in Figure 2) refer to the number of websites that reference the malicious JavaScript hosted on A.com (Layer 1 in Figure 2). **Vectors** refer to the number of active users (Layer 3 in Figure 2) that use these websites. **Rate** refers to the number of requests per second one attacking unit can trigger to the genuine server, G.com. A DoS attack uses a single attacking unit for flooding the targeted system at a rate \( R \) and no additional vectors. Therefore, the total number of requests per second (rps) is \( R \). A DDoS attack is more powerful than a DoS attack since it uses multiple attacking units, \( N \), each generating \( R \) rps. Hence, the total number of requests per second is \( N \times R \).

In a browser-based DDoS attack, apart from multiple attacking units, \( N \), there are multiple vectors, \( M \), that leverage the attacking units. Each attacking unit generates \( R \) rps on the server G.com. Therefore, the total number of requests per second is \( M \times N \times R \). Figure 2 depicts the attack scenario. If we assume: the rate, \( R \), at which malicious JavaScript sends cross-origin requests to G.com is 10 rps, the number of websites, \( N \), referencing the malicious JavaScript hosted on A.com is \( 10^4 \), the number of active users, \( M \), using these websites is \( 10^5 \). The total number of requests on the server will be \( H = M \times N \times R = 10^7 \times 10^5 \times 10^1 = 10^{13} \) rps. Considering network congestion, the actual number of requests per second on the server G would be around \( H/10 = 10^9 \) rps, which is still huge. Due to this multiplicity, this class of attacks are more powerful than DoS and DDoS attacks. We have explained a model of browser-based DDoS attack in detail in Section 2.

<table>
<thead>
<tr>
<th>Type of attack</th>
<th>Attacking units</th>
<th>Vectors</th>
<th>Rate</th>
<th>Total requests per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoS</td>
<td>1</td>
<td>-</td>
<td>( R )</td>
<td>( R )</td>
</tr>
<tr>
<td>DDoS</td>
<td>( N )</td>
<td>-</td>
<td>( R )</td>
<td>( N \times R )</td>
</tr>
<tr>
<td>Browser-based DDoS</td>
<td>( N )</td>
<td>( M )</td>
<td>( R )</td>
<td>( M \times N \times R )</td>
</tr>
</tbody>
</table>

Table 1: Comparison between the impact of types of DoS attack.

Instances of browser-based DDoS attack: There have been many cases of browser-based DDoS attacks in recent years which has affected several websites.
In 2013, a browser-based DDoS attack was launched on Incapsula [32]. The attack continued for over 150 hours, during which the website recorded malicious visits from over 180,000 IPs worldwide. The attack reached 6,000 requests/second for an average of +690,000,000 requests a day. Incapsula mitigated the attack by identifying the type of browser being used from the request header and blocked requests from all such instances. This mitigation measure, however, is not feasible in the scenario where the attack is launched from different browsers. In Section 2 we describe a model of such type of attack.

On March 27, 2015, Github has witnessed a massive DDoS attack, the largest in Github’s history till date [29, 15, 16] which prevailed for over 3 days. As per the analysis of security researchers [15], the attack is a browser-based DDoS attack caused by continuous cross-origin JavaScript calls to Github. Github has about 22 million users and 69 million public repositories as of September, 2016. We found it by using Github’s APIs for users [6] and repositories [5]. The status messages of Github [17] indicate how difficult it was for them to mitigate the attack.

In 2015, a browser-based DDoS attack was launched on CloudFlare [20, 21]. This instance is similar to the attack scenario described in Figure 2. Here the attack was triggered from Ads website (equivalent to A.com in the figure) served in mobile browsers. The attack peaked at 275,000 requests per second. The website experienced 4.5 billion requests on that day issued by 650 thousand unique IP’s.

In 2014, CloudFlare witnessed a massive DDoS attack of 400 Gbps, one of the largest attacks till date [27]. As described in the blog, to generate an attack of this magnitude the attacker must use 4,529 NTP (Network Time Protocol) servers running on 1,298 different networks and each of these servers should send 87 Mbps of traffic to the target server. This shows that a lot of resources are required in order to generate an attack of such magnitude. In 2015, CloudFlare posted a blog [14] describing browser-based DDoS attacks where they explain that in this type of attack any computer with a browser can be enrolled, making the potential of this attack nearly unlimited, even larger than 400 Gbps.

Limitations of existing browser security policies: Currently, there is no mechanism to prevent the web infiltration attack generated from tags, such as img, script, iframe, from reaching the server. Browsers as early as 1995 [35] introduced the Same Origin Policy (SOP) [36], which was designed to prevent scripts from accessing DOM, network and storage data belonging to other web origins. SOP doesn’t block cross-origin requests generated from tags like img, script, iframe. Content Security Policy (CSP), introduced in 2010 [23] improves on SOP in mitigating the exfiltration attack, as described in Figure 1 by disabling inline scripts, restricting the sources of external scripts. The root cause of browser-based DDoS attacks is infiltration requests, as described in Figure 1 which is not prevented by existing browser security policies. We have discussed about the limitations of other approaches in mitigating the attack in detail in Section 3.

Simulation and Analysis of the attack: To gain deeper insights about browser-based DDoS attack we have simulated the attack in a lab environment. We have two primary servers - genuine and attacker server, and several other websites which reference a malicious JavaScript hosted on the attacker server. The JavaScript continuously sends cross-origin requests to a genuine server (equivalent to Code 1). We load around 100 instances of websites, which reference the JavaScript of attacker server, in a web browser to simulate the attack on the genuine website. From the simulation of the attack, we have observed that the root cause of the attack is infiltration request. We have explained the simulation and analysis of the attack in detail in Section 4.

Mitigating the attack using CORP Telikicherla et al. proposed CORP [34] in 2014 to mitigate infiltration attacks, such as CSRF, clickjacking, cross site timing. Based on our detailed analysis of browser-based DDoS attacks, we propose to mitigate this class of attacks using CORP. CORP enables a server to control cross-origin interactions initiated by a browser. On deploying CORP the browser intercepts the infiltration requests and blocks the unwanted requests by the server. We have explained the approach in detail in Section 5.

When the number of requests per second on the genuine server reach a particular threshold, as set by the server, the server sends CORP as response header to the browser to block unwanted cross-origin requests. Our experimentation shows that the server mitigates the attack effectively, in less than a second, after deploying CORP. We have explained the experimentation and analysis of mitigating the attack using CORP in detail in Section 5.3.

Organisation of the Paper: The remaining sections of the paper are organised as follows: Section 2 explains the model of browser-based DDoS attack. Section 3 describes the limitations of other mitigation techniques against this class of attacks and their limitations. Section 4 describes our simulation and analysis of a browser-based DDoS attack. Section 5 explains how CORP can be used to mitigate this class of attacks, the experimentation and analysis of the approach.

2. DESCRIPTION OF BROWSER-BASED DDoS ATTACK

In this section, we describe browser-based DDoS attack in detail. We also discuss about HTTP transactions and cross-origin requests, which are essential for understanding this class of attacks, in the following subsection.

2.1 HTTP Transaction and Cross-origin request Model

In a simple HTTP transaction, a client makes a request to the server by typing the URL in the address bar and the server sends a response back to the client. Figure 3 is a simple model of browser which shows an HTTP transaction (arrow 1 is request and arrow 2 is response). HTTP requests can also be initiated by HTML elements e.g. script, img, iframe, etc. We classify HTML elements into 2 types: the elements which cannot trigger HTTP requests are Passive HTML Elements (e.g., Div, Span, Textbox etc.) while those which can trigger HTTP requests are Active HTML Elements (e.g., iframe, script, img, etc.). HTTP transactions can be triggered by active HTML elements without any explicit interaction from the user.

An HTTP request can be of two types: same-origin request and cross-origin request. An origin consists of: protocol (http, https, etc), port (default port is 80 if nothing is specified) and host (domain name of server), e.g. in http://www.A.com/resource.jpg: protocol is http, port is 80 (since no port is mentioned), host is www.A.com. In a same-
origin request, the source origin and the destination origin of the request is same, while in a cross-origin request the source origin and the destination origin of the request is different. Cross-origin requests are initiated by active HTML elements. The source origin of such a request is the server from which the active HTML is fetched in the document and the destination origin is the server to which the request is made.

Figure 4 shows a cross-origin request transaction. The example here shows how a user and a browser can be used as vectors to initiate cross-origin requests on a genuine server, G.com. A user requesting the page of the attacker’s website, A.com, can fetch an active HTML element (arrow 1 and 2 of the figure) to include a resource from G.com. This initiates a cascading cross-origin request to G.com (arrow 3 of the figure). Currently there is no mechanism to block such requests from hitting the desired server. If a large number of such cross-origin requests are made on G.com from different users, the server will be under a DDoS attack. We classify this type of attack as browser-based DDoS attack. The primary cause of these attacks are cross-origin requests. We describe the mitigation for this class of attacks in detail in Section 5.

Figure 3: An abstract model of browser showing HTTP transaction

2.2 Browser-based DDoS attack

Figure 3 explains the model of browser-based DDoS attack. The setup has a website that hosts JavaScript, A.com, multiple vector sites (site1.com, site2.com, etc.), which reference the JavaScript hosted on A.com and a victim site, G.com, which receives malicious JavaScript requests. There is a MITM (Man in the middle) agent which injects a malicious JavaScript (equivalent to Code in Listing 1) in response sent from server, A.com. The malicious JavaScript sends cross-origin requests to the victim server, G.com, at the rate of 10 rps.

The steps involved in the attack, which correspond to the numbers in the figure are as follows:

1. A genuine user browses a website http://site1.com in a browser.
2. The user makes an HTTP request to the server site1.com
3. The server site1.com responds with a web page, which is loaded into the user’s browser.
5. The server A.com responds with a genuine JavaScript file.
6. A malicious MITM device intercepts the response from A.com and tampers the script with malicious content (equivalent to the Code in Listing 1). The tampered script gets embedded into user’s browser. The script would then inherit the origin http://site1.com.
7. The tampered script continuously triggers cross-origin HTTP requests to the genuine server, G.com, for every few seconds using JavaScript’s setInterval function. The site http://site1.com launch a DoS attack on the server G.com. If the JavaScript file is embedded by multiple sites (e.g., http://site2.com, http://site3.com etc.), then the users browsing those sites will unwittingly launch a DDoS attack on G.com. This makes the attack a browser-based DDoS attack.

This model is comparable with Figure 2 where there is a central website (A.com), multiple websites (site1.com, site2.com, etc.) which reference the JavaScript hosted on A.com and multiple users who visit these websites. A MITM device intercepts the response of A.com to inject a malicious JavaScript (equivalent to Code in Listing 1) which continuously sends cross-origin requests to the target server at the rate of 10 rps (requests per second). In this example, there are 4 websites which reference the JavaScript hosted on A.com and there are 4 active users on these websites. So, the total number of requests per second on the target server will be = \(4 \times 4 \times 10 = 160\) rps. Here we can visualize how the damage which the malicious script can cause amplify from 10 rps to 160 rps. As shown in Table 1, there is no multiplicity involved in a DoS attack and a single layer of multiplicity involved in DDoS attack. In browser-based DDoS attacks there is an additional layer of multiplicity involved which makes it more dangerous than DoS and DDoS attacks.

3. RELATED WORK

In this section, we describe existing defense techniques against this class of attacks and infiltration requests.

3.1 Existing browser security policies

Browsers as early as 1995 [35] introduced the Same Origin Policy (SOP) [36], which was designed to prevent scripts from accessing DOM, network and storage data belonging
to other web origins. Any call originated from JavaScript to access DOM elements, network call, or local storage of other web origin is blocked by default. SOP doesn’t block cross-origin requests generated from tags like `img`, `script`, `iframe` from reaching the desired server. The earlier problem of cross-origin requests through automatic form submissions or content inclusion was, however, left unanswered by SOP.

However, SOP alone cannot mitigate the attacks such as cross-site scripting (XSS) [2], clickjacking [31], CSRF [1], etc. Declarative security policies [33] helps in mitigating some of these attacks. The idea behind this model is that servers specify security headers in HTTP response. The browser receives the HTTP response, parses the security headers and tries to enforce the security module. This model requires cooperation from both client as well as the server side.

One of the declarative policies is Content Security Policy (CSP), introduced in 2010 [23] which improves on SOP in mitigating the exfiltration attack (as shown in Figure 1) by disabling inline scripts and restricting the sources of external scripts. The cause of exfiltration attacks is browser’s inability to distinguish between the code intended to be part of the application and the code maliciously injected in the application. Consider a scenario where a genuine website, G.com, is injected with a code to include a malicious javascript hosted on attacker’s website. Once, the code is fetched by the browser it will have the origin of the document in which it is loaded, i.e. G.com. The code fetched will be executed in the context of G.com’s origin as it obeys SOP. CSP allows a website to instruct the browser to load resources from specific sources only. However, CSP can prevent only exfiltration attacks, such as cross-site scripting (XSS) [2], and not infiltration attacks (as shown in Figure 1) which is the root cause of browser-based DDoS attacks.

### 3.2 Subresource Integrity (SRI)

SRI [9, 14] prevents a resource from being executed in the browser if it is tampered by an MITM agent. In this technique, the user precomputes the hash of the resource and writes it along with its declaration (as shown in Listing 2). When the resource is fetched from the server, the browser computes the hash of the resource and executes the resource only if the computed hash matches with the declared hash (the `crossorigin` attribute ensures the proper enforcement of Same Origin Policy [7] with this script and prevents Cross-Site Scripting (XSS) attacks).

```html
<script src="Javascript source file">
  integrity="Hash of JavaScript file"
</script>
```

2 `<script src = "http://evil.com/malicious.js">`
3.3 Partial Cross-Origin Resource Sharing

From cross-origin requests originating from `img`, `script`, `iframe` tags, there is no mechanism to prevent the request from reaching the desired server. Partial CORS [30] aims in relaxing cross-origin resource sharing. It specifies that a browser should allow a cross-origin request to reach the target server only if its source origin and target origin are under the same administrative control, e.g. `youtube.com` and `google.com`, since both domains are managed by the same operator. In order to verify whether the domains belong to the same operator, DNSSEC [4] can be employed. The downside of this approach is that it would block genuine cross-origin requests between websites which are under different operators. Many websites use CDN (Content Delivery Network) to host their resources (Image, JavaScript, etc). When users access these websites, browsers send cross-origin requests to fetch the resources hosted on CDN. It’s highly likely that the websites, which references the resource hosted on CDN, and the CDN are present under different administrative operators.

3.4 Server-Side Filters

A valid cross-origin HTTP request has an origin header which specifies the website that triggers the request. This technique [30] is to be applied on the targeted server side, where the server can block the origin of the website from which the request is triggered. Consider a scenario, equivalent to Figure 2 where a malicious JavaScript code is hosted by an attacker’s website, `A.com`, multiple websites reference the JavaScript code hosted on `A.com` and there are multiple users visiting these websites. The malicious JavaScript code sends cross-origin requests continuously to a genuine server, `G.com`. This mitigation technique is not feasible in this scenario since it is a distributed attack, as described in Figure 2 launched from multiple websites.

3.5 Rate Limiting

In this technique [28] the threshold of incoming requests per second from a particular source is set in router. If the incoming requests per second per source is more than the threshold, the router blocks the source. However, the attacker can find the threshold of the server by using a test bot and steadily increasing the outgoing requests. After this discovery, it can generate an attack that goes below the radar, and increase the amplitude of attack by using multiple bots.

4. SIMULATION AND ANALYSIS OF BROWSER-BASED DDOS ATTACKS

In this section we describe a small-scale simulation of browser-based DDoS attack and explain our observations.

4.1 Simulation

As shown in Figure 2 the setup has a website that hosts malicious JavaScript (equivalent to Code in Listing 2). `A.com`, three vector sites (`site1.com`, `site2.com`, `site3.com`), which reference the malicious JavaScript hosted on `A.com` and a victim site, `G.com`, which receives malicious JavaScript requests. The malicious JavaScript sends cross-origin requests to the victim server at the rate of 10 rps. We use `apache` server to host the victim site, `G.com`, and `python` servers for hosting the websites `A.com, site1.com, site2.com, site3.com`. Initially, only same-origin requests are made to `G.com`. We open 100 instances of vector sites gradually in the browser to simulate the attack.

4.2 Analysis

![Figure 6: Number of Active HTTP Requests per second on Server vs Server Uptime (in seconds)](image3.png)

Figure 6 shows how the number of active HTTP requests per second (rps) varies with time on the victim server during the experimentation. Till around 63 seconds same-origin requests are made to the server, after which we gradually open the instances of vectors sites. As shown in the figure the number of HTTP requests per second on the server increases after 63 seconds. The malicious JavaScript sends cross-origin requests to the victim server at the rate of 10 rps. There are 100 instance of vector sites opened in the browser, hence the total number of requests per second on the server will be \(10 \times 100 = 1000\) rps. Considering network congestion, the actual number of requests per second on the server will be around \(1000/10 = 100\) rps.

Figure 7 shows the response time of the victim server, `G.com` for same-origin genuine user requests versus server uptime (in seconds) while the attack is simulated on `G.com`. We have done the analysis by sending continuous iterative requests for a single user. Initially till around 63 seconds, the response time is very low, around 0.006 seconds, since the server is not under additional load (as can be seen in Figure 6). The response time increases as the active number of HTTP requests on the server increases.
5. MITIGATING BROWSER-BASED DDoS ATTACKS USING CORP

In this section we describe: CORP, how CORP can be configured to mitigate a browser-based DDoS attack, and the experimentation performed to verify the approach.

5.1 Understanding CORP

Telikicherla et al. proposed CORP [34] which mitigates infiltration attacks, such as CSRF, clickjacking, cross site timing. In Section 3 we explain how CORP can be used to mitigate a browser-based DDoS attack.

Figure 7: Server Response time (in seconds) for a single genuine request vs Time (in seconds) at which the request is sent

The malicious requests received by victim site are cross-origin infiltration requests (as shown in Figure 1). From the experimentation we infer that the root cause of the attack is infiltration requests. We have discussed the limitations of existing mitigation techniques in Section 3.1. Telikicherla et al. proposed CORP [34] which mitigates infiltration attacks, such as CSRF, clickjacking, cross site timing. In Section 5 we explain how CORP can be used to mitigate a browser-based DDoS attack.

5.1.1 Setting the policy

Telikicherla et al. proposed CORP (Cross Origin Request Policy) to mitigate web infiltration attacks such as Cross-Site Request Forgery (CSRF) [34], clickjacking [31], and cross-site timing attacks [22]. CORP can be seen as a policy that controls who, i.e., which site or origin, can access what, i.e., which resources on a cross-origin server (e.g. /img/*, /img/xyz.jpg, etc), and how, i.e., through which browser event (e.g. <img>, <script> or other tags). CORP is declarative; it can be added as an HTTP response header of the website. The web browser enforcing CORP would receive the policy in the response header of the website. The browser would store the CORP in memory accessible to all tabs in the browser such that when any cross-origin request goes to this website, browser will intercept it and allow only if it complies with the CORP of the website.

Figure 8 shows the model of a browser which supports CORP. It shows the difference between exfiltration and infiltration attacks, thereby explaining how CORP differs from CSP. The figure shows a genuine server G, with origin http://G.com, an attacker’s server A, with origin http://A.com and a browser with two tabs - t1 and t2. A general browsing scenario, which is also the sufficient condition for a cross-origin attack, where a user logs in at G.com in t1 and (unwittingly) opens A.com in t2 is depicted in the model.

5.1.1 Setting the policy

Once a user requests the genuine site G.com by typing its URL in the address bar of t1, an HTTP request is sent from t1 to G and in response, along with content, the declarative security headers CSP and CORP are sent by G (shown by arrows 1 and 2 in the figure). The tab t1 receives these policies, stores CSP in its local store (as the current CSP-enabled browsers do) and sends CORP to a shared policy store P_s. P_s ensures that CORP is available to every tab/instance (arrows 3 and 4 in the figure) of the browser. Now, when a user unintentionally opens a malicious page loaded from A in t2 (arrows 5 and 6 in the figure), every HTTP request initiated by the page in t2 to G will be scrutinized and restrictions in CORP will be enforced (location 7 in the figure). Requests from t2 to G will be allowed only if they comply with the configuration in the policy. CORP needs to be configured on the server such that when a request is sent to the server, it returns CORP in the response header.

5.1.2 Syntax of CORP

CORP can be configured by specifying the set of rules in "Who What How Access/Deny", which specifies who can/cannot access what resource on the server and how through cross-origin requests. CORP rules are evaluated from top to bottom. The last (default) rule is set to "* * * Allow", which means "Allow everything". If a server sends an empty policy, it is the same as not configuring CORP at all. In such cases, the default rule is evaluated and all cross-origin requests are allowed. Complete guidelines on the declaration of CORP can be found in [34].

5.1.3 Deleting the policy

There might be a scenario when the website administrators would want the server’s CORP to be removed from user’s browser. In order to do this the website can set a max-age attribute in the policy which will force the browser
to delete the policy after the specified time. For example, a max-age value of 2592000 seconds ensures that the policy is active for 30 days, while a max-age of 0 deletes the policy immediately.

5.1.4 CORP and CSP - How They Differ:

CORP and CSP together complement each other and help in fixing infiltration and exfiltration. CSP was designed to enforce restrictions on HTTP traffic leaving a genuine webpage, as shown by location 8 in Figure 5. CORP was designed to enforce restrictions on HTTP traffic sent by a malicious web page to a genuine server (location 7 in the figure). Also, CSP expects origins as directive values, since they are sufficient to control exfiltration. CORP brings in the notion of types by specifying a 3-way relation between elements, paths and origins. In a nutshell, CORP configured on a website $A.com$ defines who (i.e., which origins) can probe what (i.e., which type of resource) on $A.com$.

5.2 Configuring CORP to mitigate Browser-based DDoS attacks

We propose using CORP to mitigate browser-based DDoS attacks based on the following observations:

- The root cause of the attack is infiltration request (arrow 7 of Figure 5) on the victim server.
- Infiltration request directly reaches the desired server without performing any check at the browser level.

From Observation 1 we infer that unwanted infiltration requests by the server should be blocked. From Observation 2 we infer that infiltration requests should be blocked at the browser level thereby taking the load off the server to block the request. However, the server must decide the type of infiltration requests which it wants to be blocked.

Based on the above inferences, we propose using CORP to mitigate the attack. In this subsection we explain how CORP can be configured to mitigate a browser-based DDoS attack.

Figure 9 shows a simplified example of browser-based DDoS attack which is mitigated using CORP. In this diagram, $A.com$ is the website containing malicious JavaScript (equivalent to code in Listing 1), that sends cross-origin requests continuously to the genuine server, $G.com$, which is targeted for the DDoS attack. $P_s$ is the policy store which is in a shared memory store, accessible to all tabs in the browser.

The steps involved in the mitigation of attack which corresponds to the number in the diagram are as follows:

1. A user browses the website $A.com$ by typing the URL in a browser. An HTTP request is made for the same to the server $A.com$.
2. The server responds with the web-page which is loaded in the browser. It also fetches a malicious JavaScript which sends cross-origin requests continuously to the genuine server, $G.com$.
3. A cascading infiltration request is made to $G.com$ to launch a DDoS attack.
4. $G.com$ sends its CORP as response, which is defined as mentioned in Table 2. It means that no server can send cross-origin requests to $G.com$ via JavaScript.

5. The browser then saves the CORP in the shared memory store ($P_s$).
6. When further cross-origin requests to $G.com$ are triggered, the browser will pull the CORP of $G.com$ from shared memory store.
7. The browser will enforce the CORP on the cross-origin request. In this case the cross-origin request will be blocked and the server $G.com$ would be immune from further cross-origin requests via JavaScript.

5.3 Experimentation and Analysis

5.3.1 Experimentation

We have done a small scale simulation of a browser-based DDoS attack in a lab environment and shown how it can be mitigated using CORP. As shown in Figure 2 the setup has a website that hosts malicious JavaScript (equivalent to Code in Listing 1). $A.com$, two vector sites ($site1.com, site2.com, site3.com$), which reference the malicious JavaScript hosted on $A.com$ and a victim site, $G.com$, which receives malicious JavaScript requests. The malicious JavaScript sends cross-origin requests to the victim server at the rate of 10 rps. We use apache server to host the victim site, $G.com$, and python servers for hosting the websites $A.com, site1.com, site2.com, site3.com$. We open 100 instances of vector sites gradually in the browser to simulate the attack. When the number of requests per second (rps) on the victim server increases a particular threshold, 75 rps, the server deploys CORP to mitigate the attack.

5.3.2 Analysis

The victim server statistics in Figure 10 shows how the number of active HTTP requests per second (rps) on the victim server, $G.com$, varies with time, while 100 instances of vector sites were gradually opened in the browser. The malicious JavaScript sends cross-origin requests to the victim server at the rate of 10 rps. The total number of requests
per second on the server will be \(10 \times 100 = 1000 \text{ rps}\). Considering network congestion, the actual number of requests per second on the server will be around \(1000/10 = 100\) rps. The victim server has set a threshold of 75 rps for a new CORP to be active. As soon as the Number of requests reach the threshold of 75 rps (at 61 seconds), the server sends the CORP to the browser as response header. The browser stores the CORP of victim server in a shared memory and blocks further unwanted cross-origin requests from reaching the server. The attack is mitigated within a second after the policy is set thereby blocking cross-origin requests from reaching the server. The server sets CORP’s `max-age` attribute, as described in Section 5.2 to 1 minute which forces the browser to remove CORP after the specified time. The browser removes the CORP at 121 seconds after which the load on the server increases. The server does not send CORP after that since the number of active HTTP rps is below 75 rps.

Figure 11 shows the response time of the victim server, G.com, for same-origin genuine requests versus server uptime (in seconds) while the attack is simulated on G.com. We have done the analysis by sending continuous iterative requests for a single user. Initially till around 18 seconds, the response time is very low, around 0.006 seconds, since the server is not under additional load (as can be seen in Figure 10). The response time increases as the active number of HTTP requests on the server increases till 61 seconds. The response time for requests made after 61 seconds is low which shows that the server has effectively mitigated the attack. Since CORP was set for autoremoval after 1 minutes, the response time increases after 121 seconds with the load on the server.

Figure 10: Number of Active HTTP Requests per second vs Server Uptime

The simulation shows that CORP can be effectively used to mitigate the attack. CORP does not block the genuine requests made directly to the website. However, If the victim server has a CDN service, CORP would block the cross-origin requests to the CDN service as well. This is one of the limitations of CORP. However, in absence of CORP, the website would be under DDoS attack unable to handle any request. Using CORP, the website would be able to mitigate the DDoS attack thereby serving the genuine requests directly reaching the server.

6. CONCLUSION AND FUTURE WORK

Our simulation shows that websites can mitigate browser-based DDoS attacks using CORP within a second, thereby serving genuine requests directly reaching the website. CORP makes it very easy for website administrators to mitigate this class of attacks.

The limitations that this approach has are:

- If the server has a CDN service, CORP would block the cross-origin requests to the CDN service as well.

- Though CORP can be easily implemented on the server side, users have to install the CORP extension in their web browser to mitigate this class of attacks, or they have to use a CORP compliant browser.

In future we aim to integrate CORP inside chromium and carry out this experiment. We also plan on finding the performance overhead and introducing CORP as a web standard. One of the recent DDoS attacks showed how Ads served in mobile browsers can be used as vectors to launch a browser-based DDoS attack. Hence, we also aim to integrate CORP with mobile version of chromium and carry out this experiment.

7. REFERENCES

[1] Cross-Site Request Forgery (CSRF)).

https://en.m.wikipedia.org/wiki/Cross-site_scripting