Bypass firewalls with of-CORs and typo-squatting Truffle Security Co.

Bypass firewalls with of-CORs and typo-squatting Truffle Security Co. - Chris Grayson, trufflesecurity.com.

- Published: date not stated
- Original: <https://trufflesecurity.com/blog/of-cors/>
- Current location: <https://trufflesecurity.com/blog/of-cors>
- Preserved from: https://trufflesecurity.com/blog/of-cors (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Bypass firewalls with of-CORs and typo-squatting Truffle Security Co.

AI agents leak secrets faster than you can fix them - schedule a demo with us at Black Hat 2026, Booth 5727

TRUFFLEHOG

CUSTOMERS

COMPANY

RESOURCES

LOG IN

Contact Us

AI agents leak secrets faster than you can fix them - schedule a demo with us at Black Hat 2026, Booth 5727

Chris Grayson

The Dig

January 3, 2023

Bypass firewalls with of-CORs and typo-squatting

Bypass firewalls with of-CORs and typo-squatting

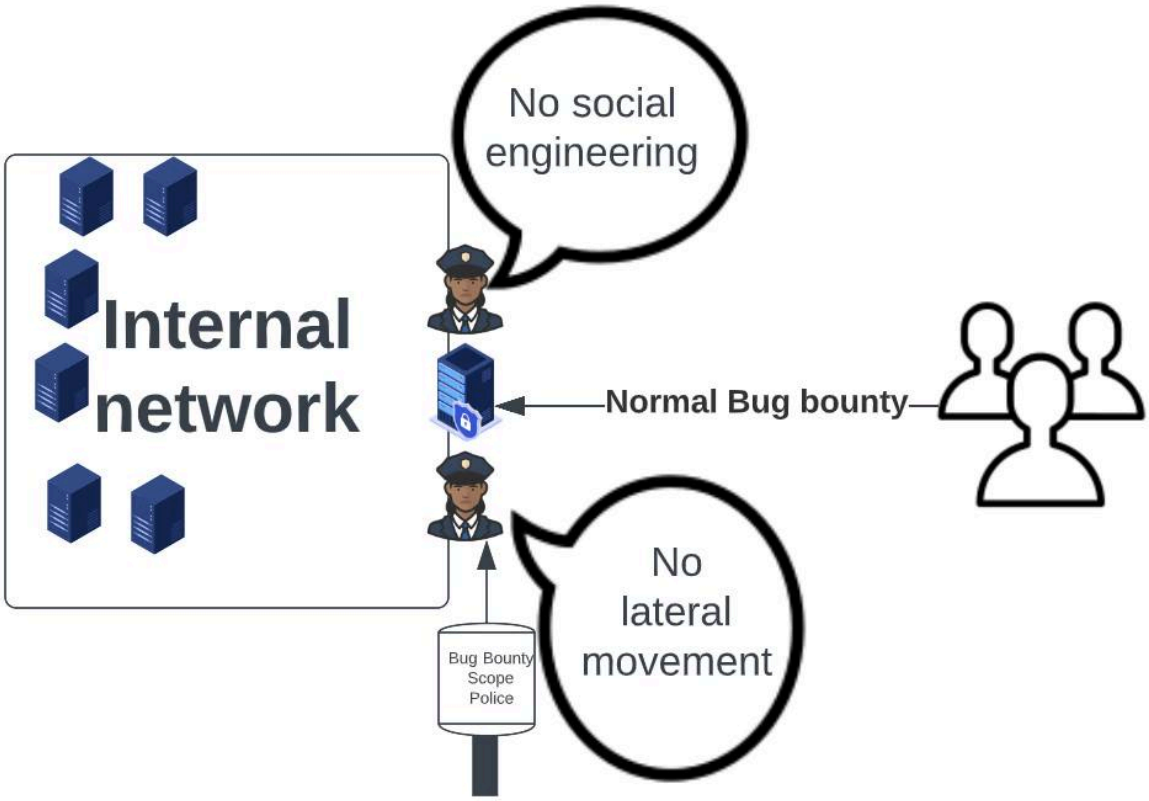
Chris Grayson

January 3, 2023

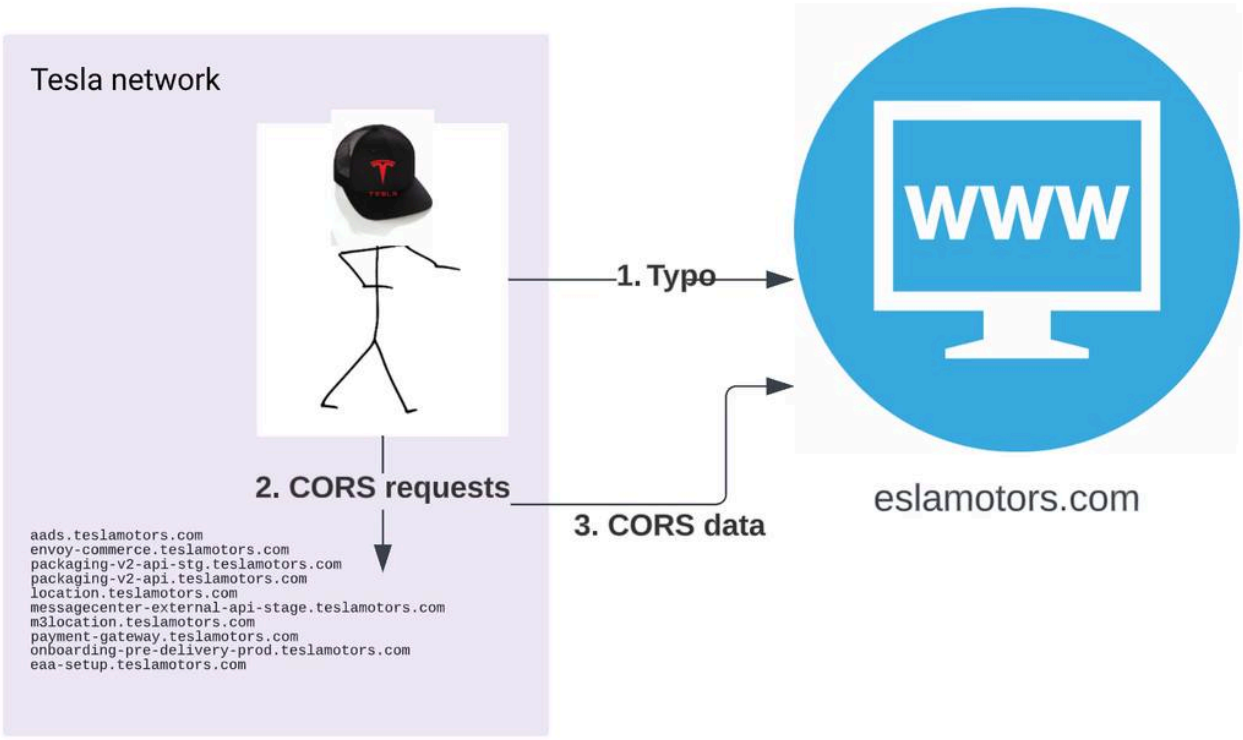
We used a new Appsec combo to get a few thousand dollars from exploiting Cross-Origin Resource Sharing (CORS) misconfigurations on internal networks in bug bounties.

Check out this example of hacking Tesla . It was so much fun that we're here to share our tooling and techniques with everyone. Rest assured that this same approach will work for plenty of other bug bounty targets.

Usually internal networks are out of scope for bug bounties due to the strict rules against lateral movement and social engineering but we devised a method to attack these internal apps directly without the use of either. Internal apps also often contain secrets in their javascript . We're aware we're walking very close to the line, but we don't believe it's been crossed.



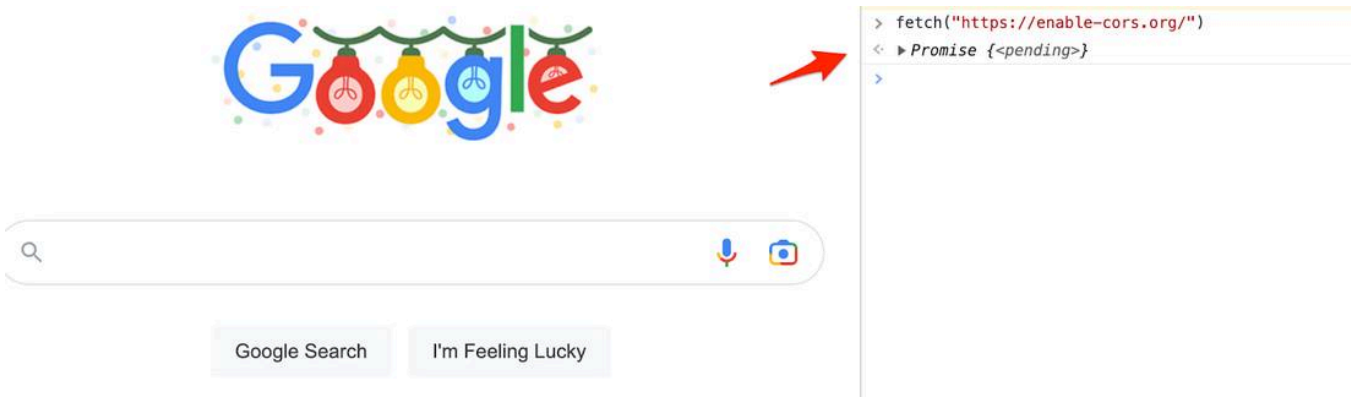
Feeling antsy and just want to get your hands on some code?? Understandable. The code can be found under the Truffle Security GitHub organization . Happy hacking.



A QUICK CORS PRIMER

The same origin policy (SOP) is a foundational element of modern browser security. Without it all of the websites you visit in your browser could access data from one another (we'd be pulling data from your beanie baby fan site page right now if we could).

The SOP, while super important, is a bit more rigid than modern web developers might like. Time and time again there have been clever ways to try to get around the SOP, often to disastrous effect. CORS is an officially supported security standard that tries to address this demand, enabling developers to "opt in" to explicit SOP carve-outs on the web sites they create.



CORS can be configured in two different ways. The first is to specify an origin that can read responses. In this case, logged in users can have their user data accessed. The second is to use a **wildcard, **which will not send login session information and is usually assumed to be safe for that reason. That's often true for externally facing websites but can go horrendously wrong for internal facing web apps that don't use authentication.

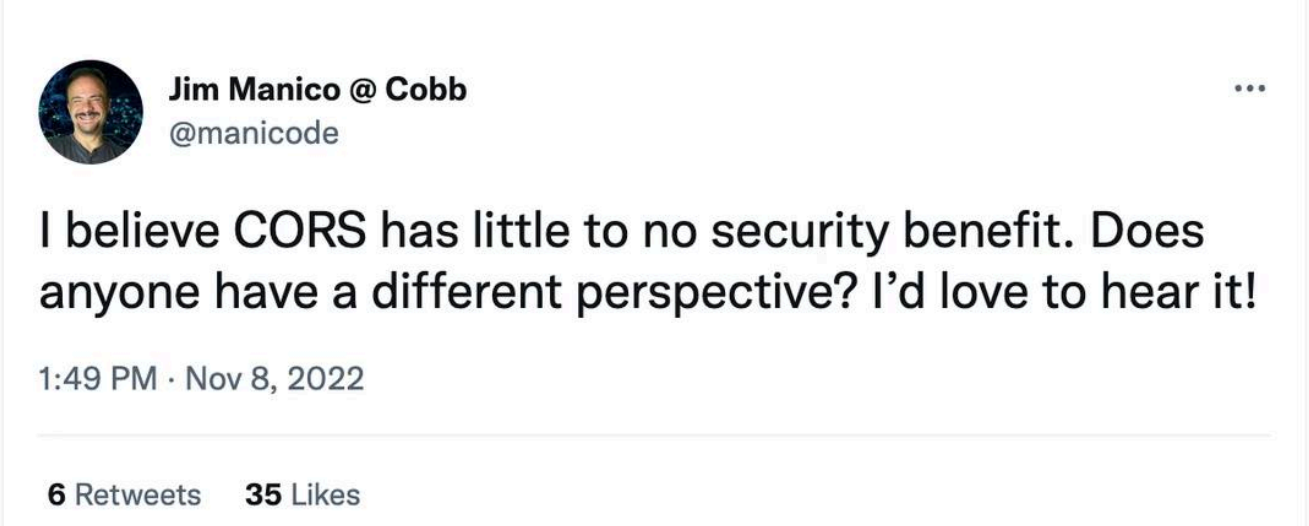
```
GET /hello/world HTTP/1.1
User-Agent: Really Cool Browser .
Origin: some.other.origin.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Mon, 12 Dec 2022 00:00:00 GMT
Server: Some Cool Server
Content-Length: 230
Access-Control-Allow-Origin: *
Content-Type: text/html; charset=iso-8859-1
Connection: Closed
```

PortSwigger has done an excellent job explaining this and talks through these misconfigurations in much more detail. For this post we will just stay focused on the issue of using wildcards for internal apps.

You'll find such an example of an incomplete view of wildcard CORS on websites like https://enable-cors.org/ which fails to mention one of the most dangerous aspects of enabling CORS; its use on internal networks. People's browsers straddle multiple networks so when a victim visits an evil website that evil website can hit all of the internal apps on internal networks.

Take a look at a couple Tweets we found that are born out of a lack of understanding of this threat. Note both posters have over 10,000 followers:



To this lack of clarity, we say “thank you” for our recent bug bounty victory.

CAN YOU FIND CORS MISCONFIGURATIONS? OF-CORS YOU CAN!

We’d like to introduce you to our new CORS exploitation toolkit, affectionately known as of-CORS. of-CORS is a web application and set of scripts that, when spun up, can sneakily prod target corporate networks for CORS misconfigurations using typosquatting and phone home with data when found. With a modicum of configuration and setup you too can poke around in the networks of large bug bounty targets for that sweet, sweet bounty loot.

The core hypothesis of-CORS was built to test was “large internal corporate networks are exceedingly likely to have impactful CORS misconfigurations.” As such we wanted a toolkit that would do all the following:

-

Enumerate likely internal subdomains for target organizations

-

Launch a JavaScript payload to prod for CORS misconfigurations against configured domains

-

Utilize a service worker to make requests long after the victim is redirected off the typosquatting domain

-

Accept results from the victim’s browser and provide some level of result queryability in a UI

-

Make some level of attempt to hide itself

-

Is likely to be visited by employees of target organizations

Achieving these goals took a bit of work in a handful of domains.

The Web Application

The of-CORS web application is a Python3 application built using Django and Django Rest Framework. When a victim visits the web application a lookup is done to determine what internal domains are configured to be probed. A browser service worker is then registered to do the probing and the user’s browser is quickly redirected away to the *assumed* intended destination.

For example, let’s say that we have registered eslamotors.com and pointed it to our of-CORS instance. An internal Tesla employee then accidentally visits https://foobar.eslamotors.com/bing/bang/bong.php (whoops, typo alert!). of-CORS then looks up the core domain of “eslamotors.com” to determine the list of domains that a payload should be launched for. This JavaScript payload for running the probing is then registered as a service worker and the employee’s browser is redirected to https://foobar.teslamotors.com/bing/bang/bong.php . The service worker continues to run in the background, issuing HTTPS requests to all of the configured domains and reporting

back to of-CORS any identified CORS misconfigurations *as well as* the HTML content for the misconfigured domains.

Just like that, we have a setup for probing internal network CORS misconfigurations with minimal indication to the victim. Any internal apps without authentication and permissive CORS will send all their data to your instance of of-CORS.

The Infrastructure

We wanted a toolkit that enabled us to receive HTTPS requests for a myriad of different domains. We also wanted the ability to spin new domains and targets up and down with relative ease. Thus we needed someone else to handle the SSL/TLS certificate management.

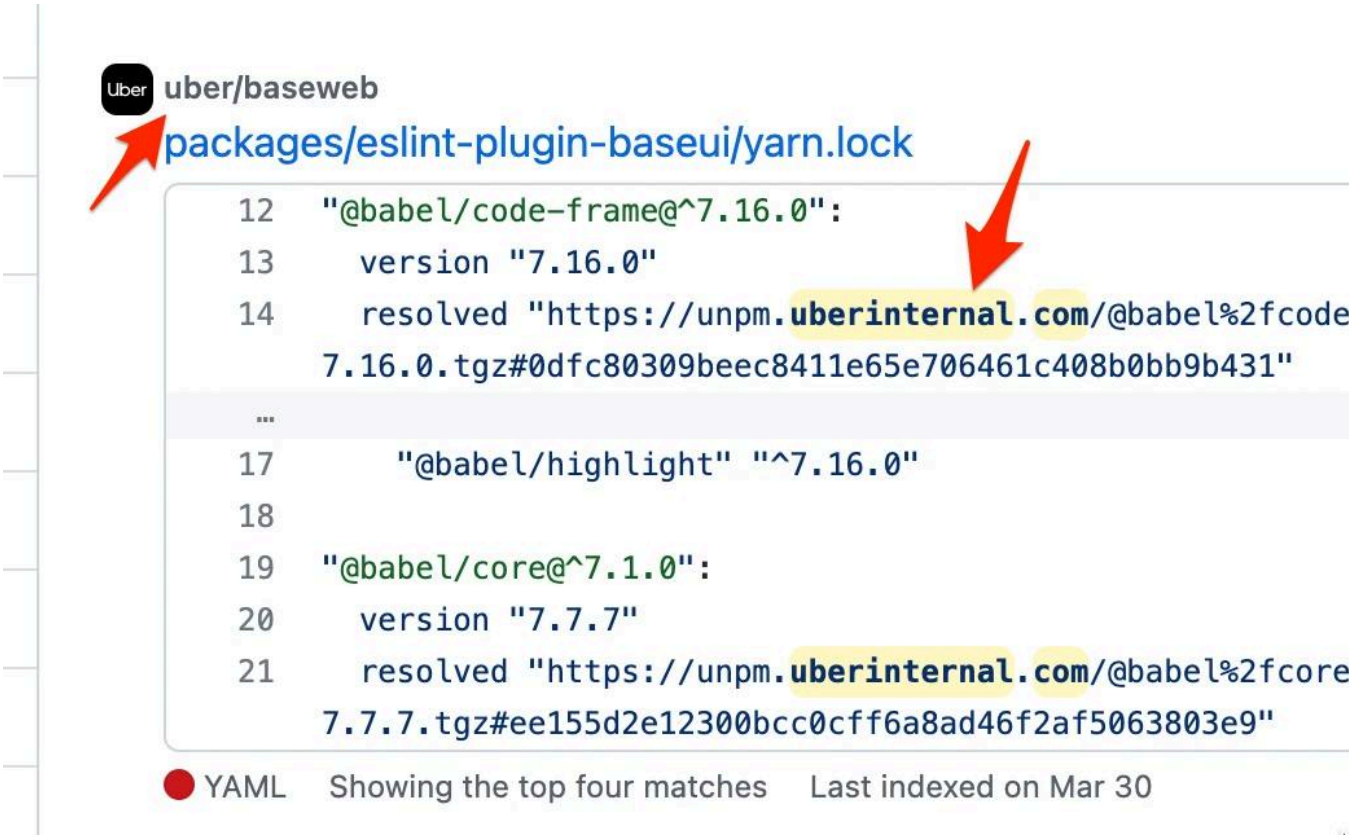
The deployment we landed on was using Cloudflare for SSL/TLS termination and DNS management and Heroku for application hosting. Cloudflare allows for wildcard CNAME routing and Heroku allows for configuring arbitrary domains to point to existing Heroku stacks. Together they achieve the infrastructure flexibility we needed for rapid iteration.

Even with this relatively simple deployment, managing infrastructure can be a real pain. To address this we implemented Terraform configuration that wires up Cloudflare, Heroku, and of-CORS in the correct configuration based on a single YAML file.

Getting Victims to Visit Our Application

Bug bounty rules can vary wildly from one organization to the next, but a common rule even across this variability is that researchers cannot engage in social engineering to aid in their attacks. Thus we were left with the problem of "how do we get victims to visit our malicious website."

Human error and bad typing to the rescue! We decided to go the route of purchasing typo-squat domains that were very similar to the internal domains used by the organizations we targeted. You'll need to do a little reconnaissance to learn what internal second level domains your target company uses. You can often find this in old commits in Github repos, android builds, and sometimes even StackOverflow questions. An example of an internal Uber domain found via GitHub is shown below:



Next you'll need to purchase a common typo of this internal second level domain. We recommend the off-by-one copy paste error that occurs when you drop the first or last character (ex: *orpinternal.com* for a company that owns the domain *corpinternal.com*).

You'll need to setup DNS for your purchased domain to route all subdomains to the of-CORS web server, and you'll also need to configure TLS for the subdomains in something like Cloudflare (we made this a little easier for you to configure as explained in the section below).

Configuring the Stack

So we've got the application, the sneaky domains to get victims to come say hi, and the infrastructure. Now we just need the data! The of-CORS configuration file not only specifies the infrastructure setup, but also which payloads should be generated for which domains. Take a look at the following sample configuration file:

```
terraform:
# You must change this to a unique string that is a valid Heroku app name
heroku_app_name: best-cors-hunter
# Fill this out with your Cloudflare API token
cloudflare_api_token: this-is-my-api-token

hosts:
  testing:
    host_domain: 127.0.0.1:8080
    redirect_domain: google.com
    targets:
      - enable-cors.org
      - example.com
```

Under the hosts.testing configuration directive we see a host_domain value of 127.0.0.1:8080, a redirect_value domain of google.com, and two targets configured for enable-cors.org and example.com. In this configuration of-CORS will expect to receive requests at 127.0.0.1:8080, will subsequently launch a payload targeting subdomains of example.com and enable-cors.org when a request is received, and will redirect the victim's browser to google.com after the browser service worker is registered.

The last piece of the puzzle here is the identification of good candidate subdomains to target under google.com and enable-cors.org. We do this by relying on OWASP's amass tool to perform subdomain enumeration and then we do our own light testing to determine which of the identified subdomains (if any) are likely internal domains. Funny enough, we throw out all domains that are external facing in this step, which is the opposite of what bountiers typically do with amass.

Note that detailed and explicit steps for configuring of-CORS can be found in its GitHub repository's README.md file.

Now that we've given you a quick tour let's see of-CORS in action!

of-CORS in Action, a Tesla Story

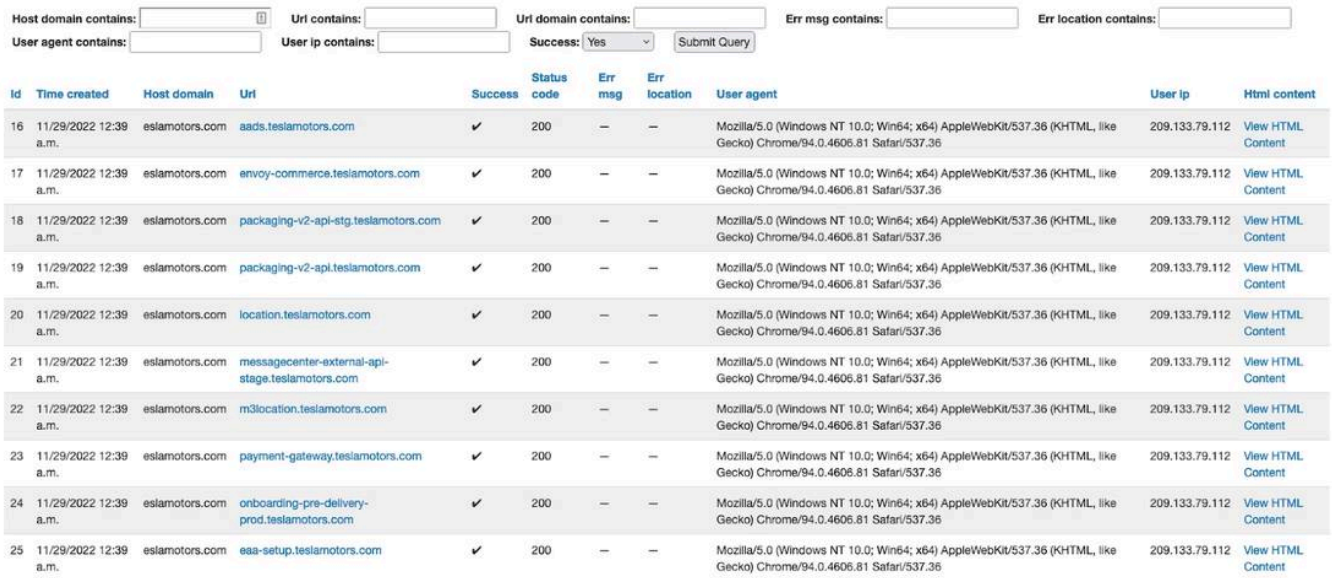
While there were a handful of bug bounty targets that we went after, our engagement with Tesla was the most positive, and they allowed us to publicly disclose the bug! Check it out here:

https://bugcrowd.com/disclosures/0e3f3821-1d26-466b-8599-7cc2f206f4d9/several-internal-applications-have-open-cors-allowing-external-folks-to-access-the-content

We set up of-CORS with the typo-squat domain of eslamotors.com and configured it to probe for CORS misconfigurations across approximately 150 subdomains of teslamotors.com. We only had to wait a few days before we got a hit:

```
http://splunk.eslamotors.com/
X-Forwarded-For: 209.133.79.112
Host: splunk.eslamotors.com
Connection: close
Sec-Ch-Ua: "Chromium";v="94", "Google Chrome";v="94", ";Not A Brand";v="99"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/94.0.4606.81 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/s
igned-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
```

Eureka! They found it. When this unfortunate victim requested the page from of-CORS we registered a service worker that probed all of the configured subdomains. Of those 150 domains that were tested *12* of them were configured to allow cross-origin access with CORS. The affected domains are shown below in the of-CORS UI:



Host domain contains: | Url contains: | Url domain contains: | Err msg contains: | Err location contains:
User agent contains: | User ip contains: | Success: Yes | Submit Query

Id	Time created	Host domain	Url	Success	Status code	Err msg	Err location	User agent	User ip	Html content
16	11/29/2022 12:39 a.m.	eslamotors.com	aads.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
17	11/29/2022 12:39 a.m.	eslamotors.com	envoy-commerce.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
18	11/29/2022 12:39 a.m.	eslamotors.com	packaging-v2-api-stg.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
19	11/29/2022 12:39 a.m.	eslamotors.com	packaging-v2-api.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
20	11/29/2022 12:39 a.m.	eslamotors.com	location.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
21	11/29/2022 12:39 a.m.	eslamotors.com	messagecenter-external-api-stage.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
22	11/29/2022 12:39 a.m.	eslamotors.com	m3location.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
23	11/29/2022 12:39 a.m.	eslamotors.com	payment-gateway.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
24	11/29/2022 12:39 a.m.	eslamotors.com	onboarding-pre-delivery-prod.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
25	11/29/2022 12:39 a.m.	eslamotors.com	eaa-setup.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content

Because of-CORS saves the HTML content that is returned from sites with CORS misconfigurations we also had the ability to review the pages for all of the affected sites. The HTML content for location.teslamotors.com is shown below:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta http-equiv="Content-Type" content="text/html" >
        <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">
        <title></title>
        <link rel="stylesheet" href="https:&#47;&#47;eaa-setup.teslamotors.com&#47;____spx/resources/templates/common/css/dialog2.css">
        <style>
            .spinner-box {
                display: block;
                position: fixed;
                top: 50%;
                left: 50%;
                width: auto;
                height: auto;
                transform: translate(-50%, -50%);
                padding: 2px 2px 2px 2px;
            }
            .spinner {
                width:  120px;
                height: 120px;
                background: url('https:&#47;&#47;eaa-setup.teslamotors.com&#47;____spx/resources/templates/common/img/loading.gif');
            }
        </style>
    </head>
    <body onload="DoAuthenticate()" bgcolor="#EEEEEE">
        <form name="soha" method="post" action="https:&#47;&#47;ztawebapps.teslamotors.com&#47;route">
            <input name="BubbleRequest" type="hidden" value="3WdvUS62LGSOil1b0&#47;4nsA2nGW4ERMfyoei9rsMWo36RNHwz06QshJR28vyIAbwMBfU0iueglazQjNY18oFeKlw63UsEMCR
            <input name="anchor" type="hidden" value="">
        </form>
        <script src="https:&#47;&#47;eaa-setup.teslamotors.com&#47;____spx/resources/templates/common/js/es6-promise.auto.min.js"> </script>
        <script src="https:&#47;&#47;eaa-setup.teslamotors.com&#47;____spx/resources/templates/common/js/common2.js"></script>
        <script>
            function PostAuthRequest() {
                var method = "GET";
                if (method == "GET") {
                    document.soha.anchor.value = window.location.hash;
                }
                var reqBody = "";
                if (reqBody) {
                    var uuid = "9272f376-c803-4fe4-cfba-a92c086da809";
                    var obj = {value: reqBody, timestamp: new Date().getTime()};
                    try {
                        sessionStorage.setItem(uuid, JSON.stringify(obj));
                        sessionStorage.setItem(uuid +"url", "eaa-setup.teslamotors.com&#47;");
                    } catch (e) {
                        if (e.name === 'QuotaExceededError') {
                            console.log('Session storage is full.')
                        }
```

We reported our findings through Tesla's bug bounty program on BugCrowd and the issue was quickly escalated, accepted, resolved, and paid out (a testament to the maturity of Tesla's bug bounty program generally and security teams specifically).

We demonstrated the ability to access and exfiltrate data from Tesla's internal network just by setting an innocuous trap and waiting for employees to wander into it. Delightful.

Conclusion

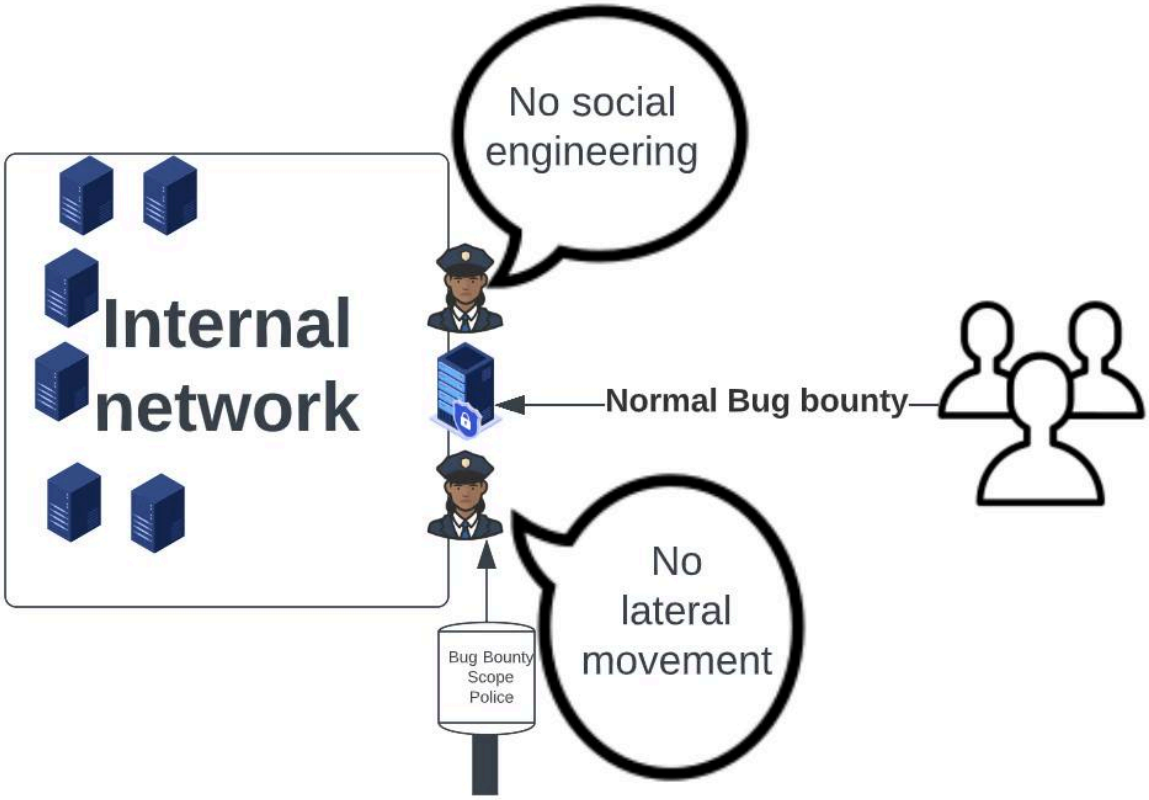
CORS may be an old topic at this point but it is still very relevant when it comes to properly securing your web applications, and this goes for external *and* internal services. With of-CORS you too can help bug bounty programs identify internal misconfigurations and enjoy some bounty loot.

The of-CORS code and documentation can be found on our GitHub page. A cheekier version of this story is told in our recent video on the topic .

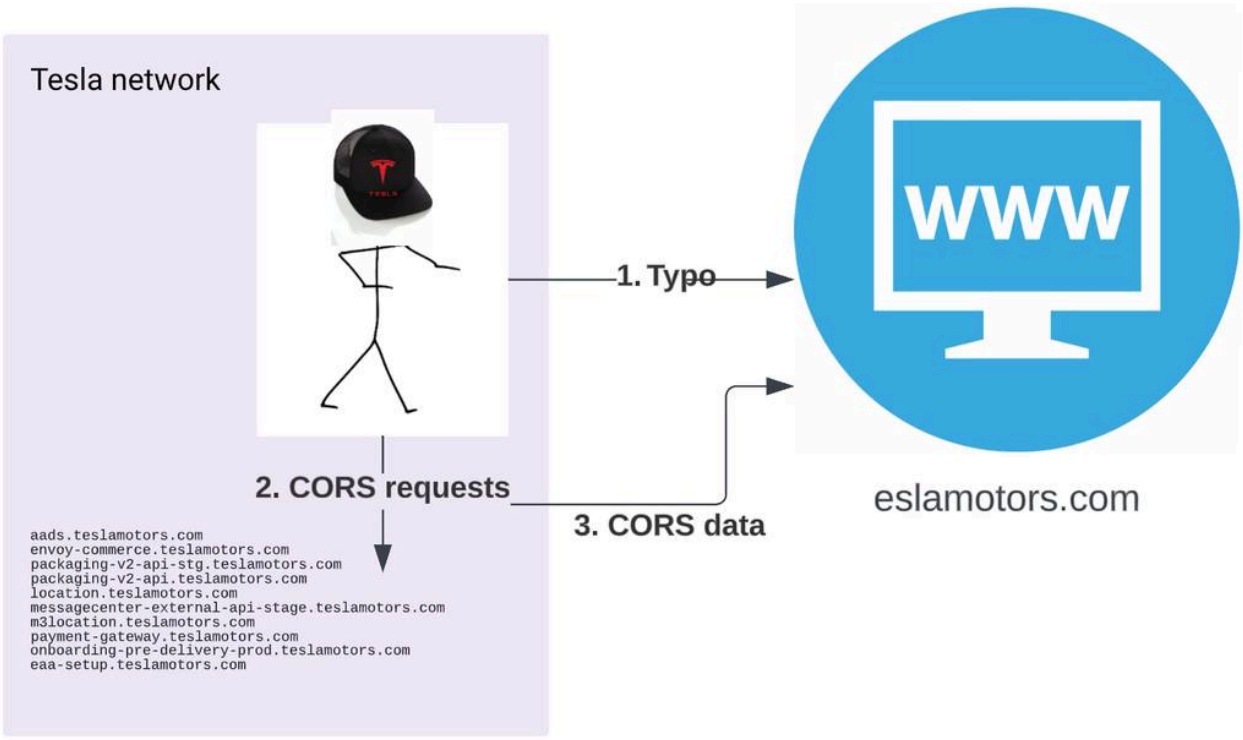
We used a new Appsec combo to get a few thousand dollars from exploiting Cross-Origin Resource Sharing (CORS) misconfigurations on internal networks in bug bounties.

Check out this example of hacking Tesla . It was so much fun that we're here to share our tooling and techniques with everyone. Rest assured that this same approach will work for plenty of other bug bounty targets.

Usually internal networks are out of scope for bug bounties due to the strict rules against lateral movement and social engineering but we devised a method to attack these internal apps directly without the use of either. Internal apps also often contain secrets in their javascript . We're aware we're walking very close to the line, but we don't believe it's been crossed.



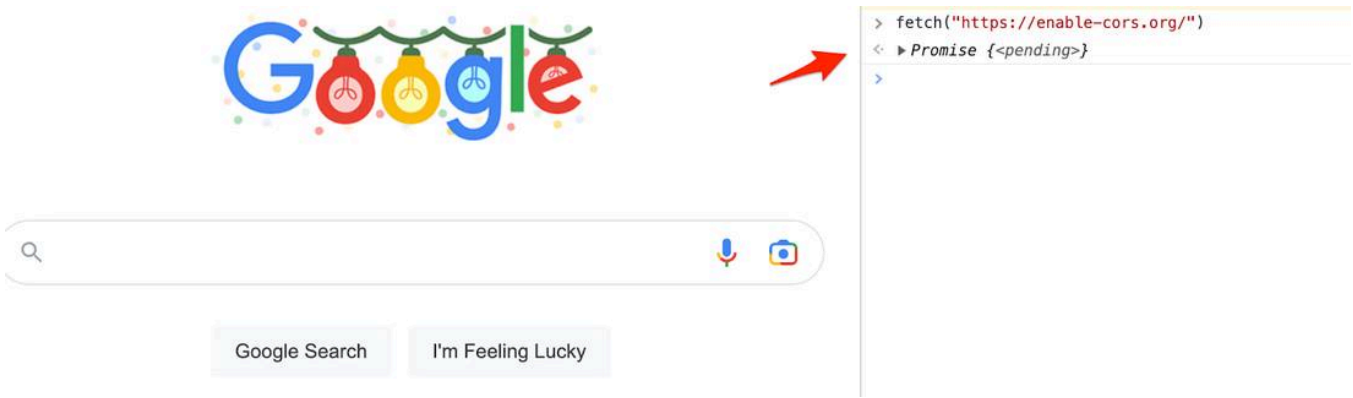
Feeling antsy and just want to get your hands on some code?? Understandable. The code can be fo und under the Truffle Security GitHub organization . Happy hacking.



A QUICK CORS PRIMER

The same origin policy (SOP) is a foundational element of modern browser security. Without it all of the websites you visit in your browser could access data from one another (we'd be pulling data from your beanie baby fan site page right now if we could).

The SOP, while super important, is a bit more rigid than modern web developers might like. Time and time again there have been clever ways to try to get around the SOP, often to disastrous effect. CORS is an officially supported security standard that tries to address this demand, enabling developers to "opt in" to explicit SOP carve-outs on the web sites they create.



CORS can be configured in two different ways. The first is to specify an origin that can read responses. In this case, logged in users can have their user data accessed. The second is to use a **wildcard, **which will not send login session information and is usually assumed to be safe for that reason. That's often true for externally facing websites but can go horrendously wrong for internal facing web apps that don't use authentication.

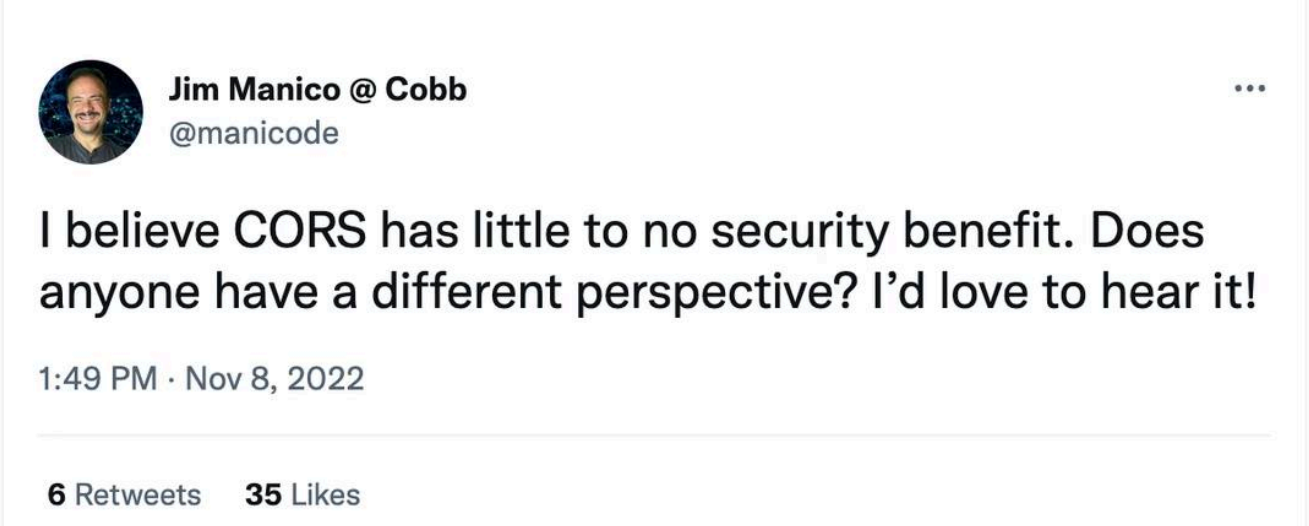
```
GET /hello/world HTTP/1.1
User-Agent: Really Cool Browser .
Origin: some.other.origin.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Mon, 12 Dec 2022 00:00:00 GMT
Server: Some Cool Server
Content-Length: 230
Access-Control-Allow-Origin: *
Content-Type: text/html; charset=iso-8859-1
Connection: Closed
```

PortSwigger has done an excellent job explaining this and talks through these misconfigurations in much more detail. For this post we will just stay focused on the issue of using wildcards for internal apps.

You'll find such an example of an incomplete view of wildcard CORS on websites like https://enable-cors.org/ which fails to mention one of the most dangerous aspects of enabling CORS; its use on internal networks. People's browsers straddle multiple networks so when a victim visits an evil website that evil website can hit all of the internal apps on internal networks.

Take a look at a couple Tweets we found that are born out of a lack of understanding of this threat. Note both posters have over 10,000 followers:



To this lack of clarity, we say “thank you” for our recent bug bounty victory.

CAN YOU FIND CORS MISCONFIGURATIONS? OF-CORS YOU CAN!

We’d like to introduce you to our new CORS exploitation toolkit, affectionately known as of-CORS. of-CORS is a web application and set of scripts that, when spun up, can sneakily prod target corporate networks for CORS misconfigurations using typosquatting and phone home with data when found. With a modicum of configuration and setup you too can poke around in the networks of large bug bounty targets for that sweet, sweet bounty loot.

The core hypothesis of-CORS was built to test was “large internal corporate networks are exceedingly likely to have impactful CORS misconfigurations.” As such we wanted a toolkit that would do all the following:

-

Enumerate likely internal subdomains for target organizations

-

Launch a JavaScript payload to prod for CORS misconfigurations against configured domains

-

Utilize a service worker to make requests long after the victim is redirected off the typosquatting domain

-

Accept results from the victim’s browser and provide some level of result queryability in a UI

-

Make some level of attempt to hide itself

-

Is likely to be visited by employees of target organizations

Achieving these goals took a bit of work in a handful of domains.

The Web Application

The of-CORS web application is a Python3 application built using Django and Django Rest Framework. When a victim visits the web application a lookup is done to determine what internal domains are configured to be probed. A browser service worker is then registered to do the probing and the user’s browser is quickly redirected away to the *assumed* intended destination.

For example, let’s say that we have registered eslamotors.com and pointed it to our of-CORS instance. An internal Tesla employee then accidentally visits https://foobar.eslamotors.com/bing/bang/bong.php (whoops, typo alert!). of-CORS then looks up the core domain of “eslamotors.com” to determine the list of domains that a payload should be launched for. This JavaScript payload for running the probing is then registered as a service worker and the employee’s browser is redirected to https://foobar.teslamotors.com/bing/bang/bong.php . The service worker continues to run in the background, issuing HTTPS requests to all of the configured domains and reporting

back to of-CORS any identified CORS misconfigurations *as well as* the HTML content for the misconfigured domains.

Just like that, we have a setup for probing internal network CORS misconfigurations with minimal indication to the victim. Any internal apps without authentication and permissive CORS will send all their data to your instance of of-CORS.

The Infrastructure

We wanted a toolkit that enabled us to receive HTTPS requests for a myriad of different domains. We also wanted the ability to spin new domains and targets up and down with relative ease. Thus we needed someone else to handle the SSL/TLS certificate management.

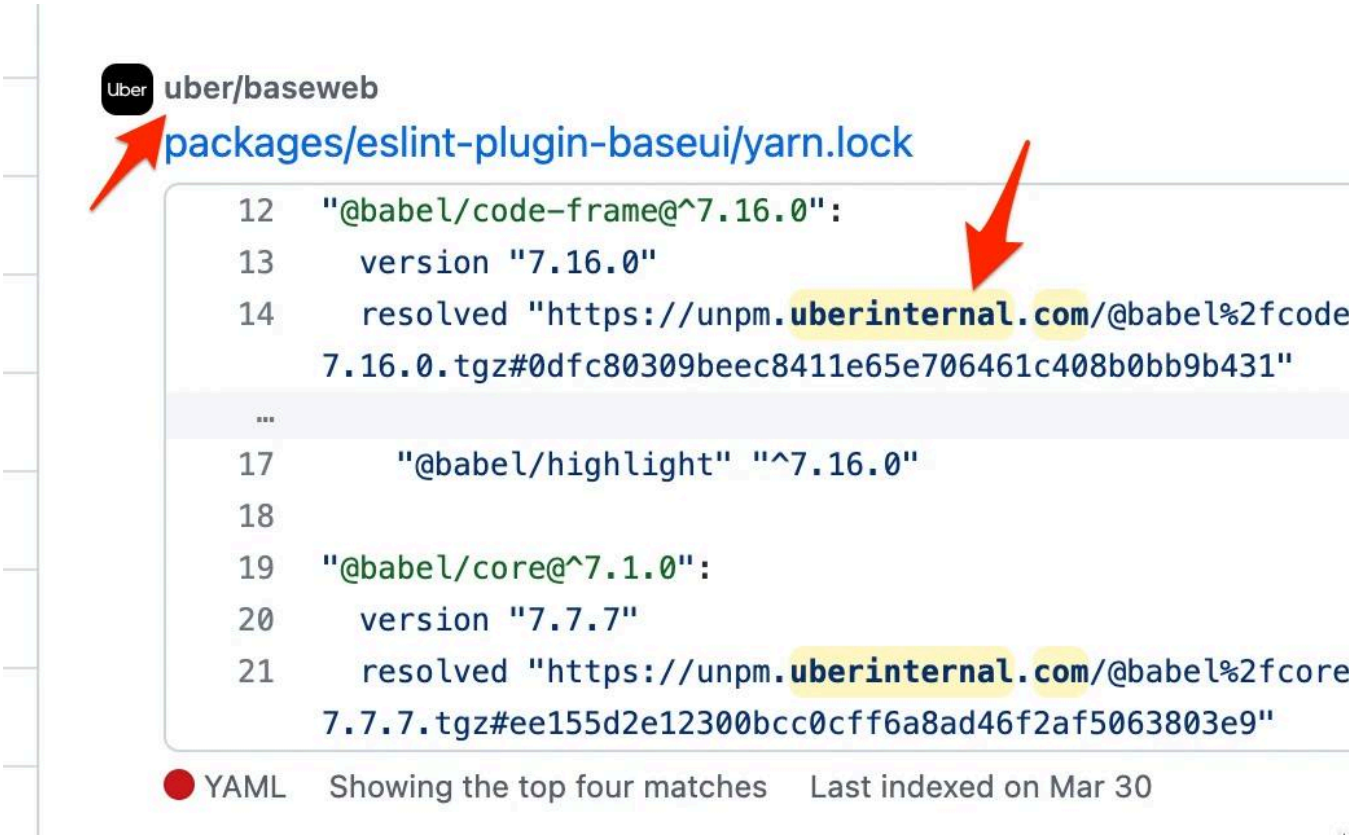
The deployment we landed on was using Cloudflare for SSL/TLS termination and DNS management and Heroku for application hosting. Cloudflare allows for wildcard CNAME routing and Heroku allows for configuring arbitrary domains to point to existing Heroku stacks. Together they achieve the infrastructure flexibility we needed for rapid iteration.

Even with this relatively simple deployment, managing infrastructure can be a real pain. To address this we implemented Terraform configuration that wires up Cloudflare, Heroku, and of-CORS in the correct configuration based on a single YAML file.

Getting Victims to Visit Our Application

Bug bounty rules can vary wildly from one organization to the next, but a common rule even across this variability is that researchers cannot engage in social engineering to aid in their attacks. Thus we were left with the problem of "how do we get victims to visit our malicious website."

Human error and bad typing to the rescue! We decided to go the route of purchasing typo-squat domains that were very similar to the internal domains used by the organizations we targeted. You'll need to do a little reconnaissance to learn what internal second level domains your target company uses. You can often find this in old commits in Github repos, android builds, and sometimes even StackOverflow questions. An example of an internal Uber domain found via GitHub is shown below:



Next you'll need to purchase a common typo of this internal second level domain. We recommend the off-by-one copy paste error that occurs when you drop the first or last character (ex: *orpinternal.com* for a company that owns the domain *corpinternal.com*).

You'll need to setup DNS for your purchased domain to route all subdomains to the of-CORS web server, and you'll also need to configure TLS for the subdomains in something like Cloudflare (we made this a little easier for you to configure as explained in the section below).

Configuring the Stack

So we've got the application, the sneaky domains to get victims to come say hi, and the infrastructure. Now we just need the data! The of-CORS configuration file not only specifies the infrastructure setup, but also which payloads should be generated for which domains. Take a look at the following sample configuration file:

```
terraform:
# You must change this to a unique string that is a valid Heroku app name
heroku_app_name: best-cors-hunter
# Fill this out with your Cloudflare API token
cloudflare_api_token: this-is-my-api-token

hosts:
  testing:
    host_domain: 127.0.0.1:8080
    redirect_domain: google.com
    targets:
      - enable-cors.org
      - example.com
```

Under the hosts.testing configuration directive we see a host_domain value of 127.0.0.1:8080, a redirect_value domain of google.com, and two targets configured for enable-cors.org and example.com. In this configuration of-CORS will expect to receive requests at 127.0.0.1:8080, will subsequently launch a payload targeting subdomains of example.com and enable-cors.org when a request is received, and will redirect the victim's browser to google.com after the browser service worker is registered.

The last piece of the puzzle here is the identification of good candidate subdomains to target under google.com and enable-cors.org. We do this by relying on OWASP's amass tool to perform subdomain enumeration and then we do our own light testing to determine which of the identified subdomains (if any) are likely internal domains. Funny enough, we throw out all domains that are external facing in this step, which is the opposite of what bountiers typically do with amass.

Note that detailed and explicit steps for configuring of-CORS can be found in its GitHub repository's README.md file.

Now that we've given you a quick tour let's see of-CORS in action!

of-CORS in Action, a Tesla Story

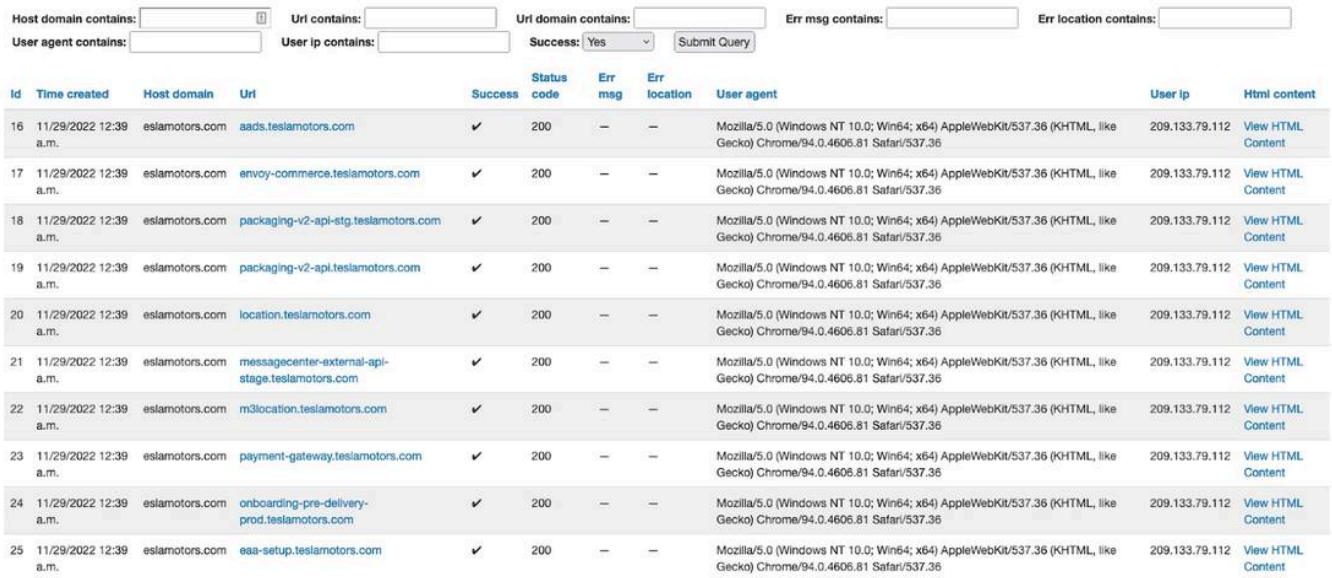
While there were a handful of bug bounty targets that we went after, our engagement with Tesla was the most positive, and they allowed us to publicly disclose the bug! Check it out here:

https://bugcrowd.com/disclosures/0e3f3821-1d26-466b-8599-7cc2f206f4d9/several-internal-applications-have-open-cors-allowing-external-folks-to-access-the-content

We set up of-CORS with the typo-squat domain of eslamotors.com and configured it to probe for CORS misconfigurations across approximately 150 subdomains of teslamotors.com. We only had to wait a few days before we got a hit:

```
http://splunk.eslamotors.com/
X-Forwarded-For: 209.133.79.112
Host: splunk.eslamotors.com
Connection: close
Sec-Ch-Ua: "Chromium";v="94", "Google Chrome";v="94", ";Not A Brand";v="99"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/94.0.4606.81 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/s
igned-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
```

Eureka! They found it. When this unfortunate victim requested the page from of-CORS we registered a service worker that probed all of the configured subdomains. Of those 150 domains that were tested *12* of them were configured to allow cross-origin access with CORS. The affected domains are shown below in the of-CORS UI:



Host domain contains: | Url contains: | Url domain contains: | Err msg contains: | Err location contains:
User agent contains: | User ip contains: | Success: Yes | Submit Query

Id	Time created	Host domain	Url	Success	Status code	Err msg	Err location	User agent	User ip	Html content
16	11/29/2022 12:39 a.m.	eslamotors.com	aads.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
17	11/29/2022 12:39 a.m.	eslamotors.com	envoy-commerce.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
18	11/29/2022 12:39 a.m.	eslamotors.com	packaging-v2-api-stg.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
19	11/29/2022 12:39 a.m.	eslamotors.com	packaging-v2-api.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
20	11/29/2022 12:39 a.m.	eslamotors.com	location.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
21	11/29/2022 12:39 a.m.	eslamotors.com	messagecenter-external-api-stage.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
22	11/29/2022 12:39 a.m.	eslamotors.com	m3location.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
23	11/29/2022 12:39 a.m.	eslamotors.com	payment-gateway.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
24	11/29/2022 12:39 a.m.	eslamotors.com	onboarding-pre-delivery-prod.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content
25	11/29/2022 12:39 a.m.	eslamotors.com	eaa-setup.teslamotors.com	✔	200	–	–	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/94.0.4606.81 Safari/537.36	209.133.79.112	View HTML Content

Because of-CORS saves the HTML content that is returned from sites with CORS misconfigurations we also had the ability to review the pages for all of the affected sites. The HTML content for location.teslamotors.com is shown below:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta http-equiv="Content-Type" content="text/html" >
        <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">
        <title></title>
        <link rel="stylesheet" href="https:&#47;&#47;eaa-setup.teslamotors.com&#47;____spx/resources/templates/common/css/dialog2.css">
        <style>
            .spinner-box {
                display: block;
                position: fixed;
                top: 50%;
                left: 50%;
                width: auto;
                height: auto;
                transform: translate(-50%, -50%);
                padding: 2px 2px 2px 2px;
            }
            .spinner {
                width:  120px;
                height: 120px;
                background: url('https:&#47;&#47;eaa-setup.teslamotors.com&#47;____spx/resources/templates/common/img/loading.gif');
            }
        </style>
    </head>
    <body onload="DoAuthenticate()" bgcolor="#EEEEEE">
        <form name="soha" method="post" action="https:&#47;&#47;ztawebapps.teslamotors.com&#47;route">
            <input name="BubbleRequest" type="hidden" value="3WdvUS62LGSOil1b0&#47;4nsA2nGW4ERMfyoei9rsMWo36RNHwz06QshJR28vyIAbwMBfU0iueglazQjNY18oFeKlw63UsEMCR
            <input name="anchor" type="hidden" value="">
        </form>
        <script src="https:&#47;&#47;eaa-setup.teslamotors.com&#47;____spx/resources/templates/common/js/es6-promise.auto.min.js"> </script>
        <script src="https:&#47;&#47;eaa-setup.teslamotors.com&#47;____spx/resources/templates/common/js/common2.js"></script>
        <script>
            function PostAuthRequest() {
                var method = "GET";
                if (method == "GET") {
                    document.soha.anchor.value = window.location.hash;
                }
                var reqBody = "";
                if (reqBody) {
                    var uuid = "9272f376-c803-4fe4-cfba-a92c086da809";
                    var obj = {value: reqBody, timestamp: new Date().getTime()};
                    try {
                        sessionStorage.setItem(uuid, JSON.stringify(obj));
                        sessionStorage.setItem(uuid +"url", "eaa-setup.teslamotors.com&#47;");
                    } catch (e) {
                        if (e.name === 'QuotaExceededError') {
                            console.log('Session storage is full.')
                        }
```

We reported our findings through Tesla's bug bounty program on BugCrowd and the issue was quickly escalated, accepted, resolved, and paid out (a testament to the maturity of Tesla's bug bounty program generally and security teams specifically).

We demonstrated the ability to access and exfiltrate data from Tesla's internal network just by setting an innocuous trap and waiting for employees to wander into it. Delightful.

Conclusion

CORS may be an old topic at this point but it is still very relevant when it comes to properly securing your web applications, and this goes for external *and* internal services. With of-CORS you too can help bug bounty programs identify internal misconfigurations and enjoy some bounty loot.

The of-CORS code and documentation can be found on our GitHub page. A cheekier version of this story is told in our recent video on the topic .

More from THE DIG

Thoughts, research findings, reports, and more from Truffle Security Co.

[[image: image] Jun 1, 2026 ###### Scanning 7.6 Petabytes of HuggingFace Training Data for Secrets](https://trufflesecurity.com/blog/scanning-7-6-petabytes-of-ai-training-data-for-secrets)

Jul 22, 2026 ###### Securing the Supply Chain: Cache Vulnerability in RubyGems

Jul 17, 2026 ###### Preventing a Supply Chain Attack: A Leaked Docker Token in a Public CI Artifact

T he Dig

Thoughts, research findings, reports, and more from Truffle Security Co.

[[image: image] Jun 1, 2026 ###### Scanning 7.6 Petabytes of HuggingFace Training Data for Secrets](https://trufflesecurity.com/blog/scanning-7-6-petabytes-of-ai-training-data-for-secrets)

Jul 22, 2026 ###### Securing the Supply Chain: Cache Vulnerability in RubyGems

STAY STRONG

DIG DEEP

TRUFFLEHOG

Open-source

Enterprise

Analyze

GCP Analyze

NEW!

Forager

Security

Integrations

Pricing

CUSTOMERS

COMPANY

About

Careers

Press

FAQ

Partners

NEW!

Contact us

RESOURCES

Blog

Newsletter

Library

Events

Videos

GitHub

Enterprise docs

Open-source docs

How to rotate

Brand assets

NEW!

DOING IT THE RIGHT WAY

SINCE 2021

#trufflehog-community #Secret Scanning

Privacy policy

Terms and conditions

Data processing agreement

Acceptable use policy

STAY STRONG

DIG DEEP

#trufflehog-community #Secret Scanning

© 2026 Truffle Security Co.

Privacy policy

Terms and conditions

Data processing agreement

Acceptable use policy

infra