École Polytechnique Fédérale de Lausanne

Search for ways of exploiting the discrepancies between the parsing of HTTP requests or responses in the various elements in a Web architecture

by Samuel Bétrisey

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Lucio Romerio
External Expert

Alain Mowat
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

August 25, 2023

Acknowledgments

I would like to thank Alain Mowat, my supervisor at SCRT, for his guidance and insightful feedback throughout this research and Prof. Mathias Payer who has been a significant source of inspiration and support during my time at EPFL.

I am grateful for the vibrant and supportive community at SCRT. Working alongside my colleagues has been both an educational and enriching experience. Their knowledge and collaboration made this venture both productive and enjoyable.

I would like to thank my CTF teams, polygl0ts and organizers, for our shared experiences and the skills we developed together.

Lastly, my family has been a strong backbone through this journey, and to my girlfriend, Jasmin, thank you for always supporting me.

Lausanne, August 25, 2023 Samuel Bétrisey

Abstract

Modern web applications are using reverse proxies to improve performance and security. However, as technology advances, new security challenges arise. In this thesis, we review the evolution of the HTTP protocol and how it led to HTTP Request Smuggling vulnerabilities.

Our approach is to automate the process of finding these vulnerabilities by developing a fuzzer that can generate many requests and test the HTTP parser of the reverse proxy for discrepancies with the backend server. We also developed a web interface to control the fuzzer and explore the results. Our tool found a significant vulnerability in HAProxy related to the FastCGI protocol. Our research highlights the importance of continuous testing, especially with tools that evolve along with the ever-changing web landscape.

Résumé

Les applications web modernes utilisent des reverse proxies pour améliorer les performances et la sécurité. Cependant, avec l'avancement technologique, de nouveaux défis de sécurité apparaissent. Dans cette thèse, nous examinons l'évolution du protocole HTTP et comment cela a conduit à des vulnérabilités de HTTP Request Smuggling.

Notre approche est d'automatiser la recherche de ces vulnérabilités en développant un fuzzer capable de générer beaucoup de requêtes et de tester le parser HTTP du reverse proxy pour des écarts avec le serveur backend. Nous avons également développé une interface web pour contrôler le fuzzer et explorer les résultats. Notre outil a découvert une vulnérabilité significative dans HAProxy, liée au protocole FastCGI. Nos recherches soulignent l'importance des tests continus, en particulier avec des outils qui évoluent parallèlement au domaine du web en constante évolution.

Contents

Chapter 1

Introduction

Modern web applications are using reverse proxies to improve performance and security. For example, a reverse proxy can forward requests to multiple backends to distribute the load. It can also be used for access control and authentication by verifying a header value before forwarding a request.

However, reverse proxies can introduce vulnerabilities if they are not configured properly or if their HTTP parser interprets requests differently than the backend server.

Previous research has shown discrepancies between the HTTP parsers of different web servers and how an attacker can exploit them to perform cache poisoning, XSS, and other attacks. Traditionally, these vulnerabilities were manually found by security researchers.

Our approach is to automate the process of finding these vulnerabilities by developing a fuzzer that can generate a large number of requests and test the HTTP parser of the reverse proxy for discrepancies with the backend server. We also developed a web interface to control the fuzzer and explore the results.

One of our standout results was the identification of a significant vulnerability in HAProxy related to the FastCGI protocol. When paired with certain configurations, especially older versions of PHP's FastCGI process, this vulnerability could lead to remote code execution.

Our core contribution is not just the discovery of new vulnerabilities but also providing the web security community with a robust tool that can evolve with the fast-changing web technologies.

Chapter 2

Background

In this section, we give an overview of the modern web architecture, the evolution of the HTTP protocol, and how it led to HTTP Request Smuggling vulnerabilities. Finally, we present the FastCGI protocol that will also be used in our experiments.

2.1 Web Architecture

In the context of modern web architecture, servers are not merely isolated entities waiting for client connections. Instead, many deployments introduce an intermediate reverse proxy server between the client and the backend servers.

Reverse proxy can serve multiple purposes:

- Load Balancing: By distributing incoming client requests across multiple backend servers, it ensures that no single server is overwhelmed with traffic. This can improve the overall performance and reliability of the Web application.

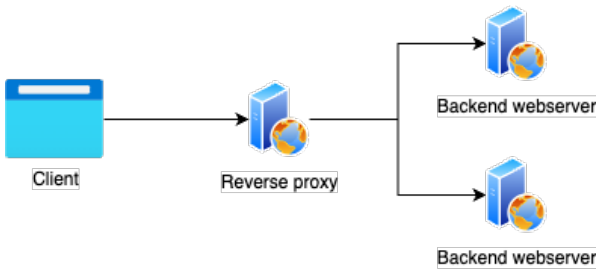


Figure 2.1: Web client connecting to a reverse proxy that load balances the requests to multiple backend servers

- Security: Web Application Firewalls (WAFs) can be deployed as reverse proxies to filter out malicious requests before they reach the backend servers. Other reverse proxies can be used to authenticate and authorize requests before forwarding them to the backend servers.
- Caching: Reverse proxies can store frequently accessed resources, reducing the need to fetch the same data repeatedly from the backend servers, thus improving response times and reducing the load on the backend servers.

However, the introduction of an intermediary layer, such as a reverse proxy, also brings complexity. The exact way in which a reverse proxy, client, and backend server interpret and handle HTTP requests and responses might differ. Discrepancies in these interpretations can lead to vulnerabilities, which will be explored further in this research.

2.2 Evolution of the HTTP protocol

HTTP versions up to HTTP/1.0 only support a single request per connection. This means that the client has to establish a new TCP connection for each request that it wants to send. This can add a lot of overhead, especially if the latency between the client and server is high; that is why HTTP/1.1 [7] introduced persistent connections. With persistent connections, the client can send multiple requests over the same TCP connection by adding a `Connection: keep-alive` header to the request. This version also introduced pipelining, which allows the client to send a batch of requests without waiting for the response of each request.

However, pipelining still does not allow true multiplexing because the responses will be sent in the same order as the requests, leading to head-of-line blocking, where a slow response will block the subsequent responses.

HTTP/2 [1] is no longer a text protocol but a binary protocol; it transmits frames over the TCP connection. This means that the parser no longer has to interpret the text and can be more robust. HTTP/2 also supports true multiplexing, which means that the responses no longer have to be in order; instead, they are matched to the requests by their stream ID. This means that a slow response will no longer block subsequent responses.

2.3 HTTP Request Smuggling

The evolution of the HTTP protocol has brought numerous enhancements to web communication. However, the push for efficiency can sometimes introduce unexpected vulnerabilities. A salient example of this is the persistent connection feature introduced in HTTP/1.1. While it greatly reduces

the overhead of constantly establishing new TCP connections for individual requests, it inadvertently paved the way for a new vulnerability class, HTTP Request Smuggling (HRS), which was first discussed in 2005 by Heled et al. [6] and later popularized by Kettle [5].

HRS or HTTP desync attack is a class of vulnerabilities that arise when there is a difference between the HTTP parser of the reverse proxy and the backend server. The reverse proxy parses the input as a single request and forwards it to the backend server, which parses it as multiple requests. This leads to desynchronization of the request/response queues, meaning that the reverse proxy will forward the response of the previous request to the next request and so on. This can be exploited by an attacker to perform cache poisoning, XSS, and other attacks.

2.4 FastCGI protocol

FastCGI [2] is a protocol that enables web servers to execute programs more efficiently than its predecessor, CGI. It reduces overhead by maintaining persistent processes to handle multiple requests, unlike traditional CGI which starts a new process for each request. For each request, the web server sends the path of the program to be invoked, the parameters, and information from the HTTP request to a FastCGI process via a TCP connection or a Unix socket. The process then returns the response through the same connection, allowing it to manage multiple requests without terminating after each one.

Because the connection is persistent, finding a desync between the web server and the FastCGI server could allow an attacker to send arbitrary FastCGI requests to the backend and execute code from outside the web server's document root, leading to remote code execution.

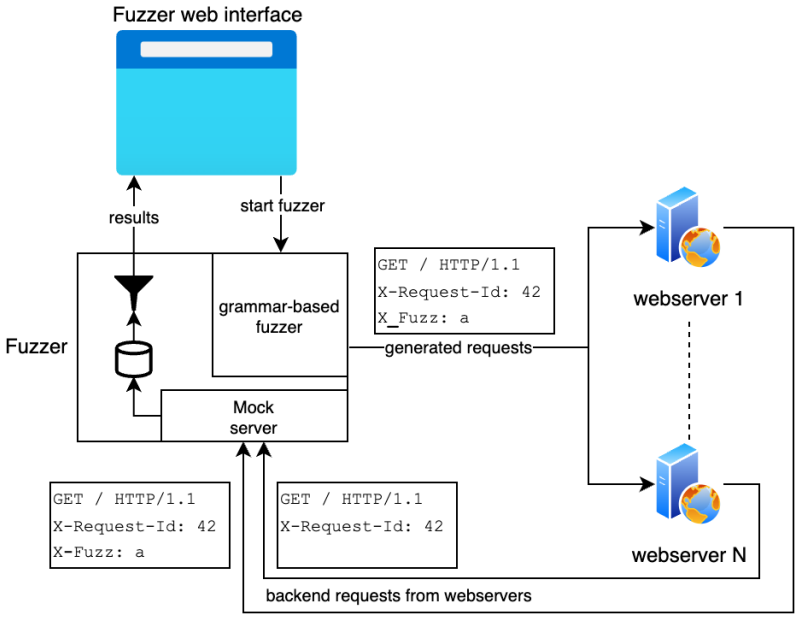
FastCGI is used by many programming languages and frameworks, including PHP, Python, and Ruby. It is supported by many web servers, including Apache, Nginx, HAProxy, and Lighttpd. So, finding a vulnerability in the FastCGI implementation of one of these servers could be impactful.

Chapter 3

Design

To research security vulnerabilities in HTTP reverse proxies and FastCGI web servers, we developed a lab environment that allows us to test these servers in isolation. We also developed an enhanced HTTP fuzzer that can be used to test these servers. The system is composed of three main components:

- The fuzzer that generates the requests sends them to the web servers and collects their requests to the backend;
- The web interface that allows the user to control the fuzzer and explore the results;
- The lab environment that contains the web servers to be tested.



3.1 HTTP fuzzer

The core of the HTTP fuzzer is based on T-reqs [4] which operates in three stages:

- Generating Inputs: T-reqs uses a CFG grammar to produce HTTP requests. This grammar is designed for request line fuzzing, enabling the creation of a variety of valid requests and treating each request line component distinctly for mutation.
- Mutating Inputs: Components can be marked as string mutable or tree mutable. String Mutable allows random modifications to characters within a component, such as deletions, replacements, or insertions. Tree Mutable facilitates broader modifications such as deleting, replacing, or inserting entire components.
- Delivery of the request: T-reqs opens a TCP connection to the target server, sends the request, and collects the response.

We extended T-reqs to support sending HTTP/2 requests and requests over HTTPS. We also added support for web servers that use FastCGI to communicate with the backend. Finally, we implemented a mock backend server that will receive the HTTP or FastCGI request from the web server, associate it with the request sent by the fuzzer, and save it so that everything can be analyzed and viewed on the web interface.

3.2 Web interface

To enable faster experimentation, we developed a web interface that allows the user to control the fuzzer and explore the results.

As depicted in Figure 3.1, the web interface allows the configuration of the grammar of the fuzzer, as well as the mutation strategies, the number of requests to send, the HTTP version, and the target web servers.

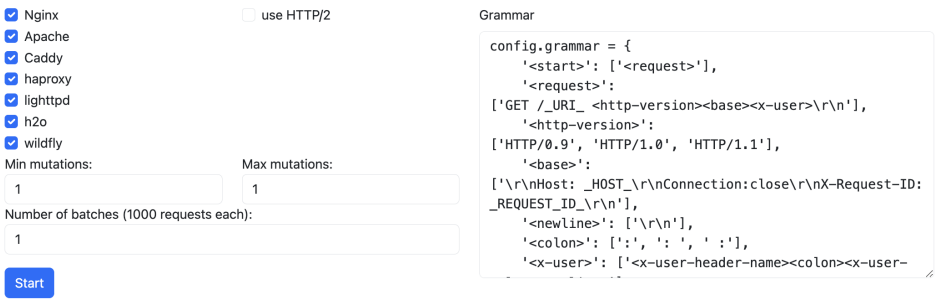


Figure 3.1: Web interface to configure the fuzzer

Each run of the fuzzer generates millions of requests, so we developed features to help the user explore the results:

- Save and load the results of a run to a JSON file;
- Deduplication of similar backend requests emitted by a web server;
- Custom user-defined filters using Python.

Figure 3.2 shows the results on the Web interface with a filter that highlights backend requests where the header name differs between web servers.

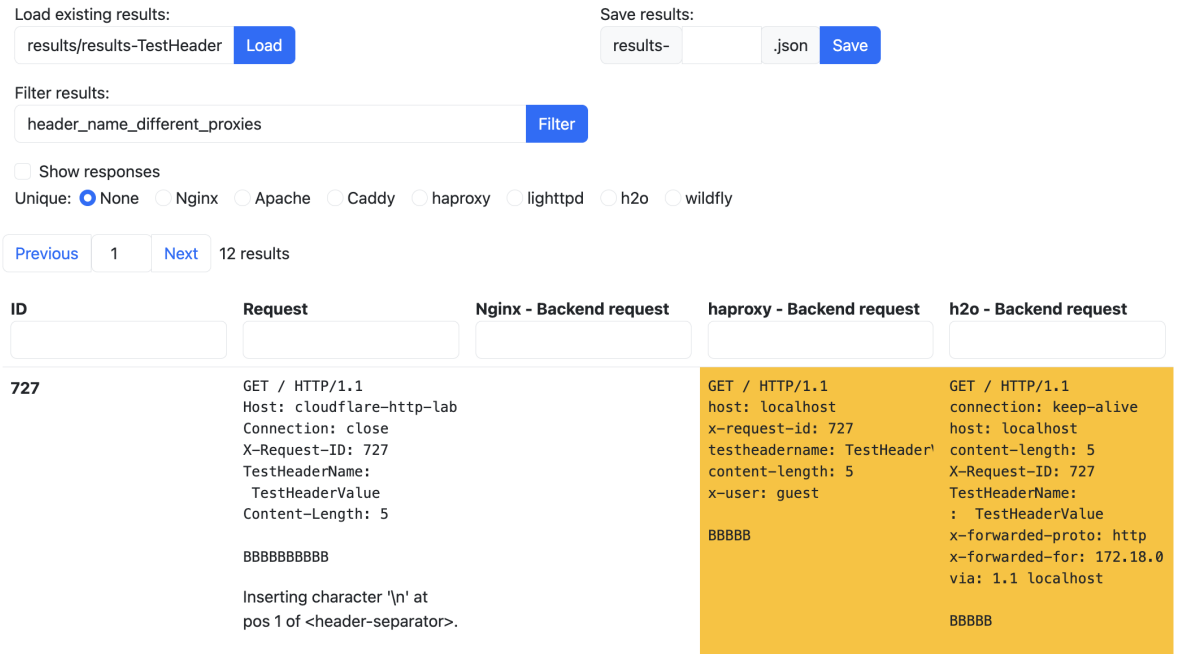


Figure 3.2: Results of the fuzzer in the interface

3.3 Lab environments

We prepared two lab environments with different web servers to run the fuzzer against.

3.3.1 HTTP lab environment

The first lab environment is composed of web servers in reverse proxy mode and is designed to find discrepancies between the HTTP request sent and the request issued by the reverse proxy to the backend server. We selected the most widely used web servers:

- Nginx
- Apache
- IIS
- Caddy
- HAProxy
- Lighttpd

To have a larger range of HTTP parsers to test, we also included less popular web servers:

- WildFly, formerly known as JBoss AS
- H2O

Because a large share of the websites use reverse proxy services and WAFs, we also included a popular service in our test environment:

- Cloudflare

Each web server is configured to forward requests to the mock HTTP server integrated into the fuzzer. The mock server will associate the request sent by the fuzzer with the request received from the web server, so the web interface will then be able to display the requests together.

3.3.2 FastCGI lab environment

The second lab environment is composed of web servers in FastCGI mode and is designed to find discrepancies between the HTTP request sent and the FastCGI request issued by the web server to the backend server. We selected the most widely used web servers that support FastCGI:

- Nginx
- Apache
- Caddy
- HAProxy
- Lighttpd
- H2O

Similarly to the HTTP lab environment, each web server is configured to forward requests to the mock FastCGI server integrated into the fuzzer.

Chapter 4

Implementation

In this section, we describe how the features of the fuzzer and the web interface were implemented. We also describe how the lab environments were configured.

4.1 New features of the HTTP fuzzer

4.1.1 Record backend requests

We wanted to capture the raw HTTP request sent by the web server to the backend server. To do this, we implemented a mock backend server that listens on a different TCP port for each target web server. It receives all the data sent over TCP, extracts the request ID in the HTTP headers to associate it with the request sent by the fuzzer, and saves it. The recorded request is also labeled with the target web server based on the TCP port it was received on.

4.1.2 HTTP/2 support

Because HTTP/2 is a binary protocol, we had to modify the grammar of the fuzzer to support it. The grammar now generates a request with delimiters that separate each header as well as the name and value of each header. There is also a delimiter between the last header and the body of the request. The grammar is specified so that the delimiters are never mutated to ensure that the request can be transmitted correctly. Here is an example of a request generated by the new grammar:

```
[H]:method[SEP]GET[/H]
[H]:path[SEP]/[/H]
[H]:scheme[SEP]https[/H]
```

```
[H]cookie[SEP]a=1[/H]
[H]cookie[SEP]b=2[/H]
[END_HEADERS]data
```

The component of the fuzzer that transmits the request was also modified to support HTTP/2. It will transform the generated payload into HTTP/2 frames using the delimiters. The protocol must use HTTPS, so it establishes a TLS connection with the target server, transmits the frames, and records the response. We use the Python Hyper-h2 library to send the request and receive the response.

The mock backend server did not have to be modified because we are interested in the case where the communication between the client and the reverse proxy is HTTP/2, and the communication between the reverse proxy and the backend server is HTTP/1.1. So, the mock backend server will receive the HTTP/1.1 request from the reverse proxy and associate it with the HTTP/2 request sent by the fuzzer.

4.1.3 FastCGI support

As FastCGI is only used between the web server and the backend, the request sent to the target does not have to be modified. However, the mock backend server had to be modified to support FastCGI. It now listens on additional ports and receives the FastCGI request from the web server.

We implemented a FastCGI parser from scratch to parse the request. At each step, if the request does not match the expected format, a verbose error message is saved in the results, as this could indicate a desynchronization between the web server and the FastCGI server.

As for HTTP requests, the FastCGI request is associated with the request sent by the fuzzer and the target web server.

4.1.4 Filters

To help the user explore the results and narrow down the search on interesting cases, we implemented a filtering system. The user can define custom filters using Python. To implement a filter, the user has to define a function that takes the original request sent by the fuzzer and the backend request received from the web server and returns a boolean indicating whether the request should be kept or not. The function is then applied to all the requests, keeping only the ones that match the filter.

4.2 Web interface

The web interface is implemented using the React framework and WebSocket to receive the results from the fuzzer in real time. It enables the user to explore the results while the fuzzer is running.

We also added the ability to send a custom request to all the targets to manually explore their behavior. This can be useful to test a specific case that was found by the fuzzer. The output of the request is also displayed in the web interface with the backend request sent by the target.

4.3 Lab environments

For both the HTTP and FastCGI lab environments, the setup is similar. To ensure an easily reproducible environment, we used Docker containers to run the web servers when possible.

4.3.1 Web servers running in containers

All web servers, except IIS and Cloudflare, are running in Docker containers. Each web server is configured to forward requests to the mock server integrated into the fuzzer. Configurations are done by mounting configuration files or with a Dockerfile. A Docker Compose file is used to start all the containers at once.

4.3.2 IIS

Because IIS is only available on Windows and the IIS container only runs on Docker for Windows, it could not be included in the Docker Compose setup. Instead, a dedicated Windows Virtual Machine was set up to run IIS. Using the URL Rewrite and Application Request Routing (ARR) modules, IIS was configured as a reverse proxy, directing traffic to our mock server.

4.3.3 Cloudflare

Integrating Cloudflare within our lab environment was more challenging because it is an external cloud-based service and not a typical server. Directly incorporating it like other web servers was not an option.

Our workaround was to set up a new domain on Cloudflare for our tests. This domain was configured to tunnel traffic through a VPN, ensuring that all its incoming requests are seamlessly redirected to our local machine running the fuzzer.

Despite being an external service, Cloudflare was set up in a manner that mirrors the behavior of our other web servers. After processing requests, Cloudflare seamlessly forwards them to the mock server incorporated into our fuzzer. This consistency ensured that our setup remained uniform, regardless of the underlying web server or service.

Chapter 5

Results

HTTP Request Smuggling vulnerabilities have been researched a lot in these past few years, so we did not find many interesting results related to HRS. However, we did find an integer overflow in HAProxy that we were able to exploit and smuggle arbitrary FastCGI requests to the backend.

After the fuzzer sent a request with a very large header, HAProxy sent a malformed FastCGI request to the backend. The size of the FastCGI packet had an overflow and was set to a value much smaller than the actual packet set by the server. It resulted in user-controlled data being sent to the backend. The mock FastCGI server of the fuzzer caught the malformed packet and reported it in the results.

We can now build a FastCGI request that will execute an arbitrary source file. It does not have to be in the document root of the web server; it can be anywhere on the disk. Finally, we include that request in the header of the HTTP request after some padding to align the FastCGI request right after the end of the first FastCGI request.

This vulnerability can lead, in some cases, to remote code execution. It requires that the backend FastCGI process continues to process the next FastCGI request after receiving a malformed packet. This is the case for older versions of PHP-FPM, the FastCGI process of PHP. However, the latest version of PHP-FPM will stop processing requests after receiving a malformed packet.

Chapter 6

Related Work

A lot of early groundwork in uncovering HTTP Request Smuggling (HRS) vulnerabilities was done manually [6]. This was a tedious process that required meticulous attention to detail, and many areas could be missed.

T-reqs [4] emerged as a pioneer, providing a significant leap forward in automation. It became the first fuzzer tailored to uncover HRS vulnerabilities by automating the process of spotting discrepancies in HTTP parsing. While it was groundbreaking in its capacity to evaluate HTTP parsers of web servers, its scope was confined to HTTP/1.1. The same author came up with FRAMESHIFTER [3], a tool targeting the newer HTTP/2 protocol. While these tools have paved the way for better automated security research, they possessed a significant limitation: there was no intuitive means to sift through and analyze the results they generated.

Our contribution, therefore, has been multifaceted. We expanded the capabilities of existing fuzzers by incorporating support for both HTTP/1.1 and HTTP/2. Recognizing the gap in the post-fuzzing phase, we designed a comprehensive web interface. This interface not only makes it easy to configure the fuzzer but also presents a user-friendly way to explore, filter, and find interesting cases from the large number of results generated.

Furthermore, our efforts extended to FastCGI, a topic that had previously been overlooked. By integrating web servers that support FastCGI in our lab environment, we brought to light potential discrepancies and vulnerabilities within this protocol, emphasizing the importance of comprehensive testing across various web technologies.

Chapter 7

Conclusion

This research dove deep into the modern web architecture, specifically focusing on potential security vulnerabilities. We studied how new developments in HTTP protocols and the usage of reverse proxies can sometimes lead to vulnerabilities like HTTP Request Smuggling. Furthermore, we also looked at the FastCGI protocol, often used in web servers, to look for similar vulnerabilities.

To dig deeper, we built a specialized fuzzer to automatically test and find these vulnerabilities. This tool was paired with a user-friendly web interface, making it easier for users to filter the results.

Although HTTP request smuggling has been studied extensively before, our tool found a significant vulnerability in HAProxy related to FastCGI. If exploited, this flaw can sometimes lead to remote code execution, especially with older versions of PHP.

The presence of such vulnerabilities underscores the need for constant vigilance in web security. As technology advances, new security challenges arise. Our research highlights the importance of continuous testing, especially with tools that evolve alongside the ever-changing web landscape.

Going forward, there is potential to refine our fuzzer, perhaps supporting HTTP/3 or other backend protocols.

Bibliography

[1] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/RFC7540. URL: https://www.rfc-editor.org/info/rfc7540.

[2] Mark R Brown. *FastCGI Specification*. Apr. 1996. URL: https://www.mit.edu/~yandros/doc/specs/fcgi-spec.html.

[3] Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, Tommaso Innocenti, Kaan Onarlioglu, and Engin Kirda. "{FRAMESHIFTER}: Security Implications of {HTTP/2-to-HTTP/1} Conversion Anomalies". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 1061–1075.

[4] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. "T-reqs: HTTP request smuggling with differential fuzzing". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 1805–1820.

[5] James Kettle. "HTTP desync attacks: Request smuggling reborn". In: *PortSwigger Research* (2019). URL: https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn.

[6] Chaim Linhart, Amit Klein, Ronen Heled, and Steve Orrin. "HTTP request smuggling". In: *Watchfire* (2005). URL: https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf.

[7] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://www.rfc-editor.org/info/rfc2616.