Localhost Detour from Public to Private Networks

Yehuda Afek[1,3], Anat Bremler-Barr[1,2,4], Dor Israeli[1,5], and Alon Noy[1,6]

[1] Tel Aviv University, Israel
[2] Reichman University, Israel
[3] afek@post.tau.ac.il
[4] bremler@idc.ac.il
[5] dorisraeli@mail.tau.ac.il
[6] alon.neuhaus@gmail.com

Abstract. This paper presents a new localhost browser based vulnerability and corresponding attack that opens the door to new attacks on private networks and local devices. We show that this new vulnerability may put hundreds of millions of internet users and their IoT devices at risk. Following the attack presentation, we suggest three new protection mechanisms to mitigate this vulnerability. This new attack bypasses recently suggested protection mechanisms designed to stop browser-based attacks on private devices and local applications [21, 23].

Keywords: Browser Based Attack · Private Network · Localhost · IoT

1 Introduction

Internet web pages work by instructing the user's browser to issue requests and execute code coming from many different sources that are not necessarily the original source (origin) of the page being rendered. This behavior is vital in order to allow rich and modular web pages. For example, a news website needs to embed images from a photos website, and a store website advertises using Google Ads and exchanges data with Google's ads servers. Attackers have been abusing the above capability to launch *browser-based attacks* on internet users by embedding malicious requests in a website the victim is tricked to visit. Several mitigation mechanisms were introduced to mitigate different variants of this type of abuse, e.g., Same Origin Policy (SOP) [37], Cross-Origin Resource Sharing (CORS) [36], Same Site Cookies [42] and the recently proposed Private Network Access [23]. In this paper we first present a new and sophisticated browser-based attack scheme that circumvents the Private Network Access mitigation mechanism, and secondly, suggest three new mitigation mechanisms that protect against the new attack.

In the early days of the Internet, an attacker could, for example, steal a victim's bank password by simply tricking the victim to visit the attacker's malicious website. In the malicious website, the attacker embeds a Javascript code that makes the user's browser requesting the bank's web page with the user credentials. The browser then renders the bank page, and the attacker code can simply read the content of the rendered web page and retrieve the victim's credentials and secretive information.

To counter attacks like the above, most browsers have integrated the Same Origin Policy (SOP) [37] mitigation mechanism. SOP blocks the attacker's website from reading content arriving from an origin (e.g., *http://bank.com*) that is different from the initiating origin (e.g., *http://attacker.com*).

However, SOP does not mitigate all browser based attacks. It addresses attacks from one origin against a different origin, but does not completely prevent malicious requests on a target server, as it prevents a different origin *reading* the responses of such requests, unless specifically permitted by the destination origin's server (using a relaxation mechanism called Cross Origin Resource Sharing [36]). However SOP does not prevent the request from reaching the target, and variants of browser based attacks still persist, like DNS Rebinding attacks [25].

Newer browser based attacks on local applications and local devices have been discovered and noticed by the security community [1, 20, 21, 23], enabling the penetration and attack of many IoT devices [24, 26] and PCs [32, 33]. An example attack against a local application is a vulnerability in Trend Micro's software [1]. The software installed a local web server on users' PCs, and it contained a vulnerability in which if the user visits a specific link (for example `http://localhost:4321/api?cmd`) on her browser, her PC is compromised (executing the `cmd` command on her PC, where `cmd` is any batch command, and 4321 is the Trend Micro local server port number). Therefore, attackers could embed such a link in their malicious website in order to compromise and

penetrate Trend Micro's users' PCs. Again, SOP does not prevent this attack even though the origins differ, because the attacker does not need to read any information from localhost in order to compromise the user's PC.

Several mitigations have been recently proposed to prevent these attacks on private end points, i.e., "Private Network Access" (PNA) [23] by Mike West and Titouan Rigoudy of Chromium, and "Internal Network Policy" (INP) [21]. These mitigations extend the Cross Origin Resource Sharing (CORS) [36] mechanism in order to block by default such requests that are initiated from public origin web servers and are destined to private origins (as defined by IANA [27, 28], e.g., *http://10.0.0.6*), and only allow the request if the destination server opts in. West et. al. even propose treating IoT devices more suspiciously, and block by default requests from private origins to local origins (e.g., to *http://127.0.0.1:8888*). We note that PNA successfully raises the bar against basic attacks in which the malicious website requests private and local resources directly. We have disclosed our finding with the PNA team [22] that have adopted one of the suggested mitigations to future versions of PNA.

In this paper we discover a three step attack against private networks, bypassing the recent mitigations, and demonstrating that the localhost may pose a greater threat to Internet users than previously thought. The key of the presented attacks is using a localhost web server as a stepping stone into the private network. Thus showing that localhost web servers are a special case of private network servers, and as suggested by [23] should be treated differently than IoT devices or other private network assets, due to several unique attributes of localhost servers that we point out and demonstrate in here. We analyzed several popular applications that install a local web server and found out that the total number of users with such applications may reach a billion (e.g., Dropbox, Steam, etc.). The severity, ease of exploitation, and wide availability of such local softwares, make this new attack a serious threat.

The basic attack has several variants. An elaborated variant is depicted in Figure 1 and 2 which is a triple step detour reaching into a private network device using a localhost as a stepping stone. Simpler variants maliciously penetrate into the user PC to steal resources or make any other damage. The basic requirement is a localhost web server (e.g., *http://localhost:1234*) listening on the victim's PC with a corresponding public web site (e.g., *https://www.allowed.com*) that following [23], is opted-in and is allowed to make cross site requests to the local web server. In the first step of any of the variants, the victim is tricked to visit the attacker's malicious website (e.g., *https://evil.com*) which embeds an iframe to the allowed server which in turn is tricked the user browser to make a request to the corresponding local web server within the iframe. The last request is approved by the preflight exchange because the local web server permits accesses from *https://allowed.com* origin (the opt-in part). This special request triggers a chosen vulnerability in the local web server, bypassing the Private Network Access mitigation. The attack continues causing the local web server to issue a request from the user browser to an IoT device in the private network. We demonstrate such an attack using a vulnerability we found in a local web server

and disclosed privately to its product security team, which has patched the vulnerabilities.

In the second part of the paper we propose three independent simple mitigation mechanisms against each step of the elaborated attack:

1. Mitigate lateral attacks against the private network as shown in our attack, by always requiring CORS [36] Preflights when requesting private or local resources, even if the origin is local or private. This mitigates the final step of the above attack, and protects the private network from an exploited local web server.
2. Protect local web servers from abuse using a default restricted rendering policy, utilizing existing mechanisms e.g., CSP [35]. This way we disable the attacker from harming the local application and also help mitigate some variants of the triple step attack.
3. Limit even further the requests from public origins to private network origins (and from private to local). Currently, Private Network Access [23] enables the private servers to allow-list their public counterpart defined by the public origin (using CORS). Our attack abuses this by fooling the allowed public website to be our allowed mediator to the local web server, and exploiting a vulnerability in a specific local page. The new mechanism that we propose here, enables the public allowed-server to specify to the browser which of its pages can initiate a request and to what private resource. This way, the attack is limited and security developers can focus on these special pages, notice however that this mitigation by itself does not block all possible attacks in the first step of the triple step attack. The mechanism can be implemented by a new dedicated HTTP server header, or by updating the Content Security Policy mitigation mechanism.

In Section 2 we describe West et. al's recent Private Network Access mitigation mechanism. Then, in Section 3, we show that localhost web servers are special and more problematic then thought before. The Localhost Detour attack that bypasses the Private Network Access mitigation and allows attacking the private network is described in Section 4. Section 5 reports on public websites with corresponding local web servers, and an example vulnerability we disclosed. Then in Section 6 three mitigation mechanisms against localhost attacks are proposed. In Appendix A we briefly review relevant browser based vulnerabilities and corresponding mitigation mechanisms.

In this paper we review Private Network Access' security model where it's authors assume there is some security hierarchy (public space is less secure than private space, and private space is less secure than local space). Thus PNA initiates a CORS preflight only when an origin of less secure network space initiates a request to an origin of a more secure network space (e.g., public to local). In reality there is no such security hierarchy, and all network spaces are both dangerous and in danger. In our scenario, a local origin initiates a request to the private network. Because PNA assumes local origins are more secure than private origins, the mitigation does not initiate a CORS preflight. As we show,

this means that local devices put the private space in danger. The solution is to relax the hierarchy assumption, and enforce preflights between all origins.

Ethical Disclosure The Localhost Detour attack scheme (Figure 2) has been disclosed to the Chromium Private Network Access repository [22] and is acknowledged by a team member that has confirmed the issue. They referenced to another post with a different scenario where an attacker could abuse the mitigation (e.g., between LAN devices). They also conclude that the problem arises because preflight is not enforced in these cases. Note that in their scenario only private-to-private preflights are required. In their response, and in a more recent post, they rightfully explain that their first priority is to roll out the PNA mitigation as is, and in a later version to upgrade the mitigation. The next version will supposedly include preflights whenever the recipient is private or local as also suggested here.

Moreover, in this paper we demonstrate the Localhost Detour Attack using a vulnerability we discovered in the Folding At Home software. As we note in the sequel, we made a responsible disclosure to F@H's security team and helped fix the problem. They were responsive and the fix was adopted into the recent version of the F@H client software.

2 Private Network Access

West and Rigoudy of the Chromium's security research team recently proposed a set of browser mitigation schemes, called Private Network Access (PNA) [23]. The goal of the PNA scheme (previously named CORS1918), is to protect local IoT devices, routers, and localhost servers from browser based attacks in the context of different, public or local origins. It significantly raises the bar for attackers and protects against most attacks in which a malicious website requests private or local resource.

This browser mitigation schemes essentially utilizes the already-adopted CORS [36] preflight protocol [34] to require the destination servers to opt-in to the special requests coming from their associated partners.

PNA employs the widely adopted IANA IPv4 [27] and IPv6 [28] address space conventions, to partition the IP address space into three categories:

1. *Local addresses* are those unique for the current device, usually referred as `localhost` (e.g., `127.0.0.0/8`).
2. *Private addresses* are those unique to the private network (e.g., `10.0.0.0/8`).
3. *Public addresses* are all other addresses, that are global to all devices (e.g., `8.8.8.8`).

They proceed to define *private network request* as all requests that are destined to either a local address and initiated by either a private address or a public address, or destined to a private address and initiated by a public address (i.e., $3 \to 2$, $3 \to 1$, and $2 \to 1$ in Table 1).

Initiator \ Destination	① Local	② Private	③ Public
① Local			
② Private	HTTPS + Preflight		
③ Public	HTTPS + Preflight	HTTPS + Preflight	

Table 1: Chromium's proposed Private Network Access mitigation logic as to when to always enforce secure context and CORS preflight. Empty boxes mean that PNA will not initiate preflights upon these cases. This logic does not cancel other mitigations such as SOP.

It seems however that the scope of the current design of the PNA mechanism excludes the security level of the initiator and destination. For example, an IoT device should protect itself against CSRF attacks (see Appendix at A.2). PNA thus focuses on the crux of the problem, and makes an assumption regarding the security level of the initiator and the destination.

With PNA the browser identifies private network requests and validates that each is of secure context as defined in the WHATWG standard [44], i.e., using HTTPS scheme for the request. Next, the browser initiates a CORS preflight to the request's destination (even for simple CORS requests), with a newly defined header, as shown in table 1. The new header and value is simply

```
Access-Control-Request-Private-Network: true
```

The browser then issues the original request only if the preflight response allows it according to the CORS header and value, e.g.,:

```
Access-Control-Allow-Private-Network: true
```

The mitigation secures the destination web servers by default, as old servers or non supporting devices will not respond with the expected header, and thus will be protected by the browser. This secure-by-default approach means that PNA is not backwards compatible, because old non-public servers will not allow preflight requests initiated due to private network requests.

3 Localhost, the Achilles Heel

The critical step on which all of the attacks presented in this paper are based is the request from an allowed public server to a corresponding local web server. The public website could be for example http://www.allowed.com and the local web server is connected to localhost:1234, see Table 2 for several examples.

On today's PCs there are likely to be several local web servers installed by different products, e.g., Steam gaming client, Dropbox local client, JetBrains programming IDEs, NVIDIA Web Helper, Folding@Home client, Arduino Create agent, and until recently also Spotify. These applications create a local web

server in order to execute code outside of the browser. For example, Folding At Home [8] has a program that allows the user to utilize their computer for distributed computing, something impossible within the browser. Thus these local servers usually execute with the installing user privileges and open the door to several different vulnerabilities, such as XSS (see Appendix at A.2).

3.1 Localhost is special

While some IoT devices also provide a web server listening in the private network at some private IP address (e.g., 10.0.0.126) or domain, there are at least four reasons to treat the localhost servers differently:

1. Local web servers are often an internal feature, and thus remain undocumented and run in the background unbeknown to the PC user. This is unlike a physical IoT device the user usually sees and usually had to buy and install. This implies that users might not be aware of a security issue in one of the local servers installed with or without their knowledge.
2. Local web servers are accessible through a known loopback IP address and port (e.g., *http://127.0.0.1:12345*), while IoT devices are usually dynamically assigned network-specific IP addresses by the router.
3. Local web servers run at a more privileged device, the user's PC, and at a higher privilege than regular browsers. E.g., Dropbox's local server can access and manipulate the file system, Steam's local server can run processes.
4. Local web servers interact with a public website, which even under PNA [23] can be approved to make requests on them.

3.2 Localhost: a gateway to the local and private network

As reviewed in Table 1 in Section 2, a public domain can request local or private resources only if the public origin is approved by a preflight. As said above, localhost servers usually interact with their corresponding public website, e.g., the public website requests information from the localhost software. With the introduction of PNA, the local servers would need to approve the new preflight request from their corresponding public websites. But then the approved web server might behave somewhat like a Trojan horse, letting in any XSS or other browser vulnerabilities found on the local server. Then, as can also be seen in Table 1, the local origin can request any origin without Private Network Access preflights. This means that a localhost web server can initiate requests to other localhost or IoT web servers in the private network.

4 Localhost Detour Attack

The Localhost Detour Attack presented here, manipulates a public web site that passes the PNA checks since it is authorized to make requests to its corresponding local web server. This could be achieved using an XSS vulnerability in the

public website, and sometimes even without a vulnerability demonstrated in the sequel, in a full attack flow. Then the attack exploits a vulnerability in the local web server that was authorized by the public request, to maliciously access a device in the private network. In the notation of Table 1 the attack goes $3 \rightarrow 1$ and then $1 \rightarrow 2$ (or $1 \rightarrow 1$ for other attack options).

The attack is out of the PNA mitigation scope, as PNA assumes the initiator and destination are secure. We circumvent the mitigation by abusing common vulnerabilities and/or behaviours in the initiator and the destination. Thus, although the PNA mitigation does raises the bar and makes penetration much harder, ways around it are shown here.

4.1 Attack Threat Model

We assume the victim has some localhost HTTP web server S running on some local port P, usually installed with some popular software. We further assume that due to PNA, the local web server S allows some public domain D_A to initiate requests to it.

As for the attackers, we assume they can trick the victim to visit a malicious website D_E (i.e., phishing), and that they know that server S is running on a local port P on the victim's computer. This assumption is valid because the programs that run such local web servers are popular and use hard-coded ports which are the same for all computers, as we show in the next chapter.

Finally, we assume the attackers find a vulnerability in the local web server S and some way to trigger it from the allowed public website D_A. The former assumption is plausible as vulnerabilities exist and we back it up with CVE examples in the next section. The latter assumption is even weaker than the former, as it does not require a full vulnerability but just a way to request the local endpoint (i.e., redirect).

4.2 Attack Flow Example

Activating the vulnerability from the public website is almost always possible with an XSS vulnerability, but even posting a link on the allowed public website that points to the local address and phishing the victim to click it is sufficient, as the request's initiator's origin is allowed. The basic attack flow begins by the attacker exploiting an XSS found in the allowed public website to request the local web server. Sometimes the attacker does not even need a vulnerability in the allowed public website. A simple example is a public website that has a page that opens an iframe to the localhost web server (as we demonstrate in Section 5). A more complex example could be posting a link on the allowed public website that points to the local address and phishing the victim to click it is enough, as the request's initiator's origin is allowed.

The attack is illustrated in diagram 1, and its sequence of steps is:

1. User (victim) is tricked to visit *https://evil.com/*.

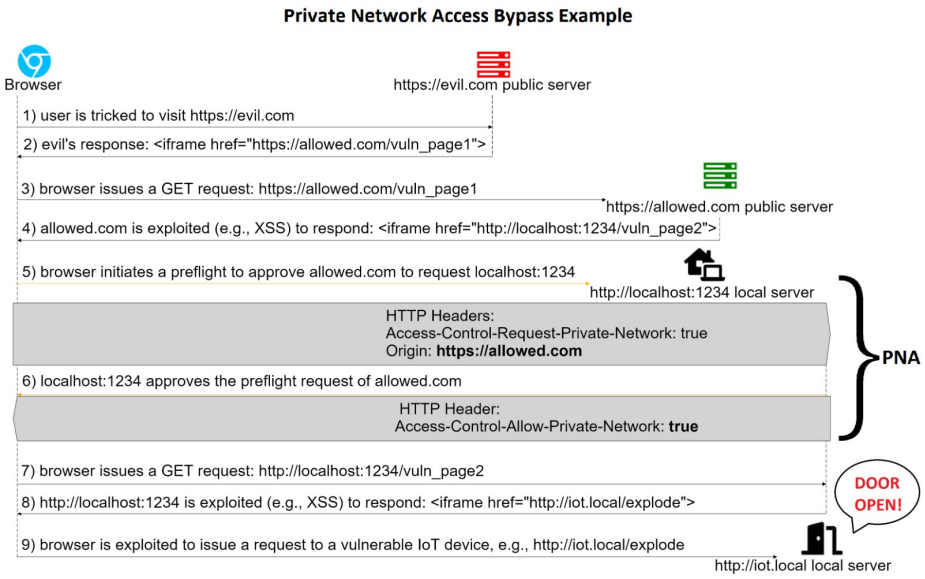

Figure 1: Request-response illustration of the "Localhost Detour" Private Network Access mitigation bypass; The victim visits a malicious website evil.com, e.g. by phishing, and then the attacker starts the attack. The attack makes use of a page in allowed.com that activates a vulnerability in the appropriate localhost server, and from there the attacker can request private resource on the LAN and even the PNA mitigation scheme does not stop it.

2. The server at evil.com responds with an HTML iframe pointing to: *https://allowed.com/vuln_page1*, where *vuln_page1* is a vulnerable page (e.g., an XSS) that is found on allowed.com's origin.
3. The browser renders the iframe and issues a request to the vulnerable page at: *https://allowed.com/vuln_page1*.
4. The public server of *https://allowed.com/vuln_page1* is exploited to return an attacker controlled response (e.g., due to an XSS). The response is an HTML iframe pointing to allowed.com's corresponding local web server listening on *http://localhost:1234/vuln_page2*. We assume *vuln_page2* is a vulnerable page on the local server's origin (XSS).
5. The browser renders the response from allowed.com and because allowed.com is a public origin that requests a private origin (localhost:1234), the browser initiates a preflight (due to PNA).
6. The local web server approves the preflight request.
7. The browser receives the approving preflight response and sends the request to the local server at address *http://localhost:1234/vuln_page2*.
8. The local web server handles the request and responds with an attacker controlled HTML. The localhosts' response is an HTML iframe pointing to

http://iot.local/open_door, a vulnerable IoT device's private web server. Here we assume that *open_door* is vulnerable page on the IoT origin.
9. The victim's browser then renders the malicious response and issues a request to the vulnerable IoT endpoint, e.g., *http://iot.local/explode*. Private Network Access does not issue a preflight when the initiator's origin is local.

Although Private Network Access' RFC states that the devices' developers are responsible for their own security measures, we think this is not a realistic assumption to be made. The conclusion we take from our attack is that a vulnerable local server allows a resourceful attacker to circumvent the Private Network Access mitigation, and thus we must not leave the devices security unaccounted for.

4.3 General Attack Cookbook

Now that we have reviewed the attack example, we provide the attack cookbook recipe, as also illustrated in figure 2:

1. Find an application software that installs a local web server with a public counterpart (See Table 2 for a few examples). This could be a specific, prechosen software (e.g. a company wants to harden their software), or any such application. These programs can be found by scanning PCs for local ports it listens on, or by crawling the Internet for public websites that request localhost and investigating their software.
2. Find a vulnerability in that software's local web server. The vulnerability type could be anything, and this will define the capabilities of the attack. A different vulnerability like a redirect page, could allow an attacker to send a CSRF request to a neighbouring IoT device.
3. Understand the capabilities of the previous vulnerability to devise the final step of the attack. This could be for example to execute bash code on the victim's machine, to use known CSRF attacks against IoT devices, to scan the network etc.
4. Find a page in the public website counterpart that can request the local resource. This could be done in two ways, either by the page the developer intended to request the local web server, or by finding a minimal vulnerability in the website. For example, an XSS in the website could possibly allow an attacker to request any such local resource, but even a weaker vulnerability like open-redirect is sufficient.
5. Combine ingredients to a complete attack. This could usually be done by registering a domain and phishing the victim to visit it. The malicious website will include an iframe to the allowed website, which in turn will request the local resource. Additional methods exist, for example if the software's website allows users to post messages with links (like Amazon Stores), an attacker could post a malicious link to local resource, and phish victims to visit the link ("Click here for more deals!"), actually initiating a request to the local resource from the counterpart website's origin.

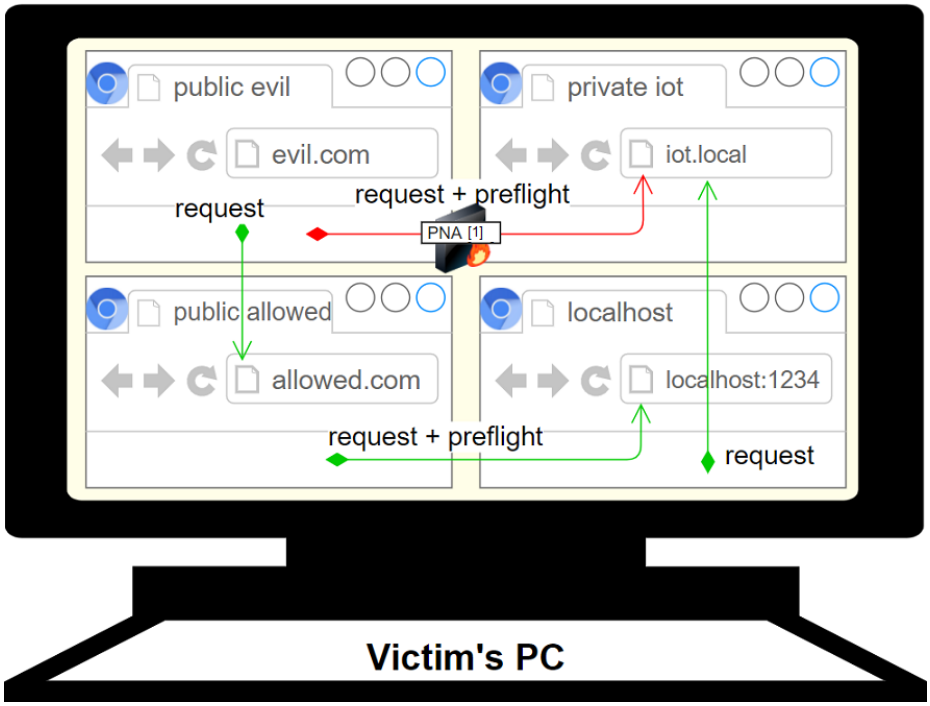

Figure 2: PNA (red arrow) prevents public websites like evil.com from requesting private or local resources like iot.local. The Localhost Detour attack (green arrows) circumvents the mitigation by abusing a localhost webserver that allows some public origin like allowed.com. Thus evil.com requests iot.local indirectly by requesting allowed.com, which in turn requests localhost:1234, passing the PNA preflight check. Finally localhost requests iot.local without needing a preflight.

5 Public-Local Matching Web Server Pairs

We noticed three main HTTP request methods by which a public web server requests the local one:

1. Iframe: some public domains (e.g., *Foldingathome.org* [8]) embed their local web server in an HTML iframe, making the local web server's interface part of the public website.
2. AJAX API: some public websites (e.g., *Dropbox.com* [6]) use the local web server as an API endpoint to retrieve information regarding the user (e.g., automate website login) or to instruct the product to act in some manner (e.g., open a file in a local editing application). The public websites usually use Fetch or Websocket requests.

3. HTML Links: some public websites (e.g., *Kubernetes.io* [16]) have HTML links pointing to the localhost server, as part of a usage guide, blog posts, or an interactive way to instruct the user to visit the application.

Table 2 lists several examples of a public and local web servers, where the first makes HTTP requests to the second. Most usually these servers belong to the same product, like Steam or Dropbox, etc.

Product Name and Origin	**Port**	**Usage**	**#Users**	**examp. CVEs**
Dropbox [6]	17600	API	700M	2019-12171 2018-20819
Steam Client [18]	27060	API	120M	2019-14743, 2019-15315
Jetbrains InetlliJ [7]	63342	Links	8M	2019-15037 2019-15848
Folding@Home [14]	7396	iframe	>4M	-
Spring.io [17]	8080	Links	>1M	2018-1270 2018-1271
Logitech Media Server [11]	9000	Links	-	2017-16568 2017-15687
Kubernetes CTL [16]	8001	Links	-	2019-1002100
Swagger [4]	3200	Links	-	2016-5682
Arduino Create Agent [5]	8992	API	-	-
Jupyter Notebook [13]	8888	Links	-	2015-6938 2018-8768
HedgeDoc [10]	9000	Links	-	2021-21259
MinIO [15]	9000	Links	-	2020-11012

Table 2: List of products that create a local web server and the public origin we suspect will be allowed by the local server after Private Network Access mitigation mechanism is implemented. Each product's local HTTP port is specified, together with the reason the public origin requests it (e.g., Steam.com requests its local product for authentication API, and Kubernetes.io has HTML links to its localhost endpoint). We provide estimates to number of users and example CVEs in the local application to demonstrate the wide spread and availability of this critical issue.

5.1 Three examples

We provide three examples of products with such public-local web server pairs, one for each of the above three request types: iframe, AJAX API, and HTML

links. Each of these examples would have to be approved by the CORS-preflight exchange if and when PNA is implemented in the browsers, in order not to break the corresponding product functionality. We further point out for each example a vulnerability an attacker may utilize in order to attack the localhost server.

Note that this type of vulnerabilities is common and we give several examples. We start with a vulnerability we have recently discovered, and verified in our lab (i.e., *folding at home*), and then continue to illustrate possible scenarios in known products (i.e., *Dropbox* and *InteliJ*). Notice we have not yet searched or know of open specific vulnerabilities in the later products, but only claim that these products are potentially open to the risk of being abused as part of an attack.

Folding At Home Folding@Home [8] is a distributed computing software designed to help scientists find new therapeutics. The software raises a local web server on port 7396, and it servers as a dashboard (e.g., to view current research process). Users can access the dashboard through the browser directly (i.e., *http://localhost:7396*) or by using the software's official website at *https://client.foldingathome.org/*. We have successfully implemented and tested the following attack in our lab, and responsibly disclosed it to the company. Folding At Home acknowledged the issues and fixed the vulnerability in the official release.

The public website works by automatically opening an HTML iframe pointing to the local web server's index page. Thus, if an attacker had a vulnerability stored in the local server, they would not need a vulnerability in the public website, since the public website triggers it by itself.

In the Folding At Home website, researchers can create research teams, and add team information (e.g., name, image and website's URL). These details can be modified by anyone with the team's password, through the public website. The modification form does not validate that the input is legal (e.g., that the new team website URL does not contain illegal URL characters), i.e., the team name could be a javascript string. Right away as a user opens the local website index page, the information regarding their account is shown including their research team's information and team's website URL. The code that displays the team URL is shown below:

```
if (typeof data.team_url != 'undefined')
    team_name = '<a target="_blank" href="'
        + data.team_url + '">' + team_name + '</a>';
$('#box-points-team').html(team_name);
```

This code is vulnerable to an XSS (see Appendix at A.2) attack because the team URL is not sanitized or validated. An attacker with the victim's team's password, can change the team URL to be:

```
    "><script>MALICIOUS CODE</script><"
```

Then, when the user opens the local index page, the malicious javascript code is executed in their browser.

An attacker then could abuse the fact that the public website opens an iframe to the private local server, and utilize the above XSS vulnerability to attack the user private network. A malicious code attacking an IoT device with a CSRF (see Appendix in Section A.2) vulnerability might look roughly as follows:

```
"><script>
    fetch("http://iot.local/explode")
</script><"
```

We disclosed this vulnerability report privately to Folding At Home's security team, and they have quickly responded. They fixed the issues, both the XSS vulnerability in the local web server and the validation issue in the public website.

Dropbox Dropbox [6] is a popular cloud storage company with over 700 million users. The users can store files in the cloud by installing a software on their PC, and they can then manage their data through the public website (*https://dropbox.com*).

The software starts a local web server that listens on port 17600, and the public web site sends Websocket requests to it (e.g., to open a file locally).

An attacker could utilize an open-redirect vulnerability in the public website (which she yet has to find), e.g., [31], and embed it in an iframe in their malicious website, redirecting to the localhost web server. Thus the attacker can trigger a vulnerability in the local web server (again yet needs to be found), bypassing Private Network Access.

Jetbrains IntelliJ Jetbrains' IntelliJ IDEs [12] are popular programming editors. Their software raises a local web server at port 63342. This local port serves several purposes, e.g., to open a file X in a local editor when a user visits a link of a specific form, (e.g., *http://localhost:63342/open?file=X*).

Jetbrains has an official public website used for bug tracking (at https://youtrack.jetbrains.com), where users and employees can describe their bugs and issues. Many times the issues regard the local web server and HTML links are posted with the intention to be clickable (as an interactive evidence). For this reason this origin will probably be trusted by the software.

Thus, attackers could utilize a vulnerability in the public domain, e.g., [2], and successfully request the local server. After that, the attacker can exploit vulnerabilities in the local server, e.g., [33], bypassing the Private Network Access.

6 Proposed Mitigations

In this section we describe three mitigation mechanisms that work independently to mitigate this family of attacks. The full chain of mitigation mechanisms is depicted in Figure 3.

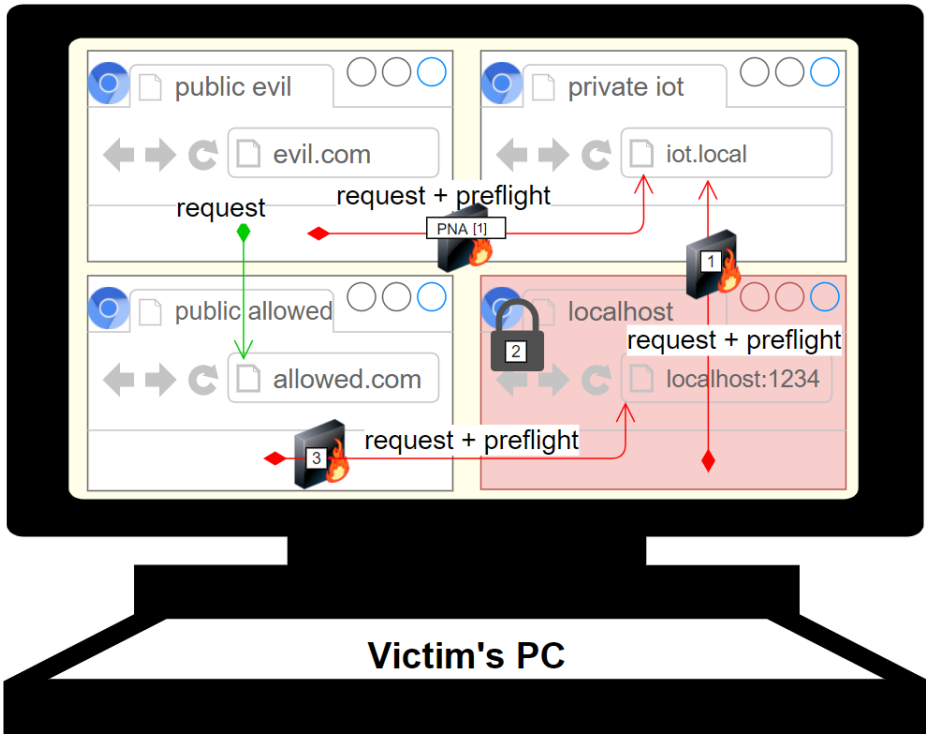

Figure 3: Illustrative summery of our proposed mitigations against the Localhost Detour attack: (1) Always issue preflights to private and local origins in order to mitigate lateral movement (see also Table 3), (2) Default restrictive policies to protect local web servers from exploitation, and (3) Explicit server side destination declaration in order to minimize attack surface to specific pages.

6.1 Lateral Movement Mitigation

The final and third step of the attack is an access from the local server to a honest private server (e.g., on an IoT device). This could be mitigated by setting an equal footing between all sub IP spaces, as can be shown in Figure 3 at mitigation number `(1)`.

This mitigation can be enforced by simply extending the PNA mechanism to always issue preflight request/response to non-public origins, as already done in PNA from public origins to private and local origins.

The main goal of this mitigation is to block by default unintended requests, and this is achieved by initiating a preflight. Thus a requested private or local origin will have to knowingly inspect and allow the request. Unfortunately there is currently no good solution to certificates inside a private network, and this is why it is currently unrealistic in our opinion to require secure context. Note this

is not perfect as it means the destination should validate the initiator's integrity on its own.

The proposed extension to PNA logic is depicted in table 3.

Initiator \ Destination	① Local	② Private	③ Public
① Local	Preflight$_{new}$	Preflight$_{new}$	
② Private	HTTPS + Preflight	Preflight$_{new}$	
③ Public	HTTPS + Preflight	HTTPS + Preflight	

Table 3: Summery of our proposed mitigation mechanisms against lateral movement from the exploited localhost to the private network. The suggestion modifies Private Network Access (PNA) (see Table 1) to always initiate a preflight authorization when the destination is local or private.

6.2 Private Servers Protection

The second mitigation mechanism we propose is to impose restrictive default policy in the browser when it issues requests to the private network. This mitigation is designed to protect private and local server from cross site scripting and misuse (label `(2)` in figure 3), and is secure by default. If a product needs to disable the mitigation in order to implement some functionality, it can relax the mitigation.

6.3 Attack Surface Minimization

Finally, we propose to restrict the attack surface by what we call "explicit private destinations", as illustrated in Figure 3 with indicator (3). The goal is to specify the public pages that are allowed to issue requests to the private network. This mitigation minimizes the public pages that an attacker can abuse, and the private pages the attacker can exploit. For more details, see Appendix C.

7 Conclusions

This paper highlights a weak (perhaps the weakest) link in the chain of browser mitigation mechanisms, a localhost web server, and suggests new methods to patch this link. We reviewed browser-based attacks, and build upon Chromium's recent Private Network Access [23] mitigation mechanism. The Localhost Detour attack that is presented here, bypasses this mitigation and therefore points out a dangerous vulnerability. The attack abuses a vulnerability in the local web server and its corresponding public website in order to attack the private network. Moreover, developers of localhost web servers should consider this security aspect in their design.

Acknowledgements: We thank Dr. Amit Klein for very helpful discussions and advice.

APPENDIX

A Browser-Based Attacks and Mitigations, Background

We review several specific examples of browser based attacks employing a localhost web server, following with a quick overview of basic browser based attack techniques (CSRF, XSS), and background on basic browser mitigation mechanisms, SOP, CSRS, and CSP. A detailed review of the recent PNA mitigation mechanism suggested by West and Rigoudy is provided in the following section.

A.1 Browser-Based Vulnerabilities Examples

1. An attack using a local web server was mounted against *EE 4GEE* wireless router [3], a portable cellular router that supplies a wifi hotspot. Hemmings [26] demonstrated how an attacker can break the EE router by sending the device's SIM a malicious SMS message that contained a javascript XSS [40] vulnerbaility. When the router owner views the router administration page, the malicious script automatically executes on the victim's browser and can reconfigure or cause any other damage to the router.
2. *ZOOM [19]:* In [32] Leitschuh demonstrates a CSRF vulnerability that enables an attacker to forcibly add a victim to a Zoom meeting with camera on without their consent. The user is tricked into a web page with `http://localhost:19421/launch?action=join& confno=[X-Zoom-meeting]` iframe request in it.
3. *Geth [9]* is an Etherium cryptocurrency wallet software, that has a local web server and listens on local HTTP ports. Recently it was shown [30] that the software is using a common API that is vulnerable to DNS Rebinding [25] attacks, which allowed attackers to steal victims' money simply by fooling the victim to visit the attckers' website.

Additional notable vulnerabilities and attacks against localhost were reported, e.g., [1, 33, 38, 43].

A.2 Known Browser Based Attack Techniques

Cross-Site Scripting: in Cross-site Scripting (XSS) [40] an attacker exploits a vulnerability in a trusted website and injects malicious code into that website. For example, a website *https://example.com* has a welcome page that retrieves the user name from the URL, as follows:

https://example.com/?name=Jake

Then the website prints a message in bold font to the screen:

`<b>Welcome Jake!<b>`.

An attacker could craft the following malicious URL:

https://example.com?name=<script>malicious code</script>
Then, if a victim is tricked to visit this link, the malicious code will execute in their browser.

Cross-Site Request Forgery In the Cross-Site Request Forgery (CSRF) [39] attack, a victim is tricked to visit an attacker's website in which the attacker embeds a request to a different website, and makes the requested website to believe this request is user initiated, thus this site acts as if the user herself requested the action. The attack usually abuses the fact that if the user is signed into some critical website (bank) and the malicious website initiates a request to it, the browser appends the logged-in user's cookies. Several best practices (e.g., CSRF tokens [41]) and mitigations (e.g., Same Site Cookies [42]) have been proposed and implemented, but notice that there is no silver bullet and developers must take this attack into considerations.

DNS Rebinding In DNS rebinding [25], an attacker fools the browser to think the attacker controls an IP address which they in fact do not own (e.g, the router's IP), and allows the attacker to circumvent the Same Origin Policy mitigation mechanism. Essentially the attacker come from an approved origin but associates it with a different IP address, the IP address of the attack target. This attack does not assume the attacker controls the DNS server, only the DNS registry of the malicious website.

A.3 Known Browser Mitigation Mechanisms

Same Origin Policy (SOP) [37] permits an origin to read data only from requests/frames of exactly the same origin or origins approved by a CORS preflight exchange (see next point); Origin is defined in RFC6454 [29] as the combination of scheme, host-name and port number. For example, in the URL *https://www.example.site:4321/a/b.c*, the scheme is HTTPS, the host-name is www.example.site and the port is 4321. This mitigation guarantees that an attacker cannot violate the user's privacy in some other website (origin). Notice that SOP does not stop an origin from issuing a request to a different origin, but rather blocks the initiator from reading the response.

Cross-Origin Resource Sharing (CORS) [36] originally an extension relaxing SOP for certain specified origins, allows a server at origin A to opt-in to allow an origin different than A to read the response from the server at origin A.

CORS makes a distinction between simple requests (i.e., an HTTP request that `form` can initiate) and complex requests (e.g., with special headers), and works differently for each type or request. Simple requests are always accepted by the destination server, but it has to specifically allow different origins reading the response. For complex requests the destination server requires a special preflight request-response handshake before allowing the browser to issue the complex request when it originates from specific origins.

As it turns out, sometimes CORS serves as a secure-by-default cross-site request forgery (CSRF) mitigation, since it requires a positive response from the server.

The CORS request contains headers that describe the special request, such as:

```
Access-Control-Request-Method: POST
```

And the CORS response contains appropriate headers, i.e.,:

```
Access-Control-Allow-Method: *
```

Content Security Policy (CSP) CSP is another browser mitigation mechanism [35], which aims to define allowed behaviours in the current frame (e.g., allowed images origins in this frame), in order to prevent cross-site scripting. It utilizes an HTTP header in the server response, and thus protects the current frame alone and not inner frames. For example, by the following policy header the browser will execute Javascript code only if it is stored on a separate file in the current origin, and err otherwise:

```
Content-Security-Policy: script-src 'self';
```

B Appendix: Details on Private Servers Protection

The key point is not to trust content loaded from a private network request. The implementation should provide flexibility and enable the integration of additional mechanisms.

For example, we propose to add a strict default Content-Security-Policy, as follows:

```
Content-Security-Policy: script-src 'self';
```

In addition, we propose to add a default textual-only Content-Type header to each private network response in order to restrict the default behavior of such services to only basic API queries, e.g.,:

```
Content-Type: text/plain
```

A more strict approach, albeit extreme, could be to prompt the user with a warning before requesting or rendering a private network origin, as done by TLS when a malformed certificate is detected. Notice that this mitigation is hard to implement correctly since the prompt should be understood by a layman and not only to tech-savvy security professionals. Furthermore, such a prompt has the danger of inducing fatigue and becoming ineffective.

This mitigation protects the user from having malicious code executing in a private origin. It is important to note however, that it does not block server-side vulnerabilities like DOS attacks.

Also, it is secure by default, but developers can override and cancel the default protection, as in most mitigations.

C Appendix: Details on Attack Surface Minimization

Finally, we propose to restrict the attack surface by what we call "explicit private destinations", as illustrated in Figure 3 with indicator (3). The goal is to specify the public pages that are allowed to issue requests to the private network. This mitigation minimizes the public pages that an attacker can abuse, and the private pages the attacker can exploit. Thus the mechanism helps websites focus their security efforts on these specific pages. This minimization could also be achieved by embedding the page that requests the private resource in a different dedicated sub-domain. Thus the private destination only allows requests initiated by the dedicated sub-domain and not by the parent domain. Thus attackers could not abuse vulnerabilities in the parent domain and are restricted to the few pages in the dedicated sub-domain. This approach requires more effort from developers, as they need to create and manage sub domains for each such request, and even then it only shifts the problem to a different sub-domain and does not change things drastically. Thus it is less likely to be adopted and we propose the Attack Surface Minimization mitigation.

Therefore the browser blocks all requests from a pubic page to a private resource, unless the public page is explicitly allowed to request the private resource. This specification can be implemented as a new CSP token or as a new HTTP header. Note that it is preferable to reduce complexity and implement the mitigation using the existing mechanism (i.e., CSP), since new headers are error prone.

In our attack example, assume that the public server allows only a specific public page */A.html* to request a private page */X.html*. This blocks an attacker from issuing a malicious request from the public page */A.html* to a different local page */Y.html*, or from a different public page */B.html* to the local page *http://localhost:1234/X.html*.

In this example the header returned in page */A.html* is:

```
Explicit-Private-Destinations:
    http://localhost:1234/X.html
```

Conversely, this could also be achieved by adding a new token to the CSP (see the background Section A.3) scheme, in the same notion as the header. Note that if implemented using CSP, then the browser should enforce a default CSP. We suggest the new CSP token as follows:

```
Content-Security-Policy: private-network-src
    http://localhost:1234/X.html
```

Although this solution does not completely mitigate the Localhost detour attack, it does mitigate some attacks and allows to focus the security efforts on specific pages.

D Future Work

There are several ways to extend this paper :

D.1 Cross Network Request Forgery Tokens

In order to mitigate malicious origins from issuing requests to the private network, PNA makes a distinction between allowed initiator origins and prohibited origins. In order to mitigate attackers from abusing allowed public origins, our third mitigation tries to make a further distinction between allowed pages from other pages (within an allowed initiator origin). An attacker can still abuse the allowed origin's allowed page, and trick it to request the local web server.

A future research could be to make an even further distinction between allowed and prohibited requests, and to enforce that the public website intended to issue the request to the private network. A natural way to implement such mitigation is by using some proof of authenticity. This is an extension of the concept of tokens as a mitigation against old-school CSRF attacks, as previously depicted in e.g., [41]. The main difference is that in standard CSRF-tokens the generator of the token (the public server) is also the request destination. In our case, the generator is again the public website, but the token validator is not the public website, but either the browser or local server.

D.2 Local Non-Web Servers Attack

In our research we focused on local web servers since the browser naturally allows such requests. However in our experiments we observed many products that have a local server that does not communicate in HTTP, or even in TCP. For example, several products have local UDP servers for real time communication. Some examples include: "Zoom" for video chat, "Discord" for voice chat, and the popular online game "World of Warcraft" as well for voice chat. These are motivated by real time communications that may allow some packet loss, but require very low latency, and thus in that case UDP is a better option than TCP.

Further research on extending our methods to this kind of non-web servers is required. The main challenge in this research are to issue non-http requests, and to inspect such local non-web servers.

D.3 Specific Product Vulnerability Research

As the paper demonstrates, local web servers are a weak point in private networks. Several highly popular products have a local web server (e.g., Kubernetes, Dropbox, Jupyter, etc.), and vulnerabilities in these products put hundred of million of users at risk of an attack like the one described here.

Although we propose mitigation within the browser, it is a good idea to research and develop fortification of product-specific local web server implementations.

References

1. Trendmicro node.js http server listening on localhost can execute commands (2016), https://bugs.chromium.org/p/project-zero/issues/detail?id=693&redir=1

2. Cve-2020-7913: Xss vulnerability on jetbrains youtrack (2020), https://cvedata.com/cve/CVE-2020-7913/
3. 4gee wifi mobile broadband (2021), https://ee.co.uk/help/help-new/phones-and-devices/ee/4gee-wifi
4. Api documentation & design tools for teams — swagger (2021), https://swagger.io
5. Arduino digital store (2021), https://create.arduino.cc
6. Dropbox is building the world's first smart workspace. (2021), https://www.dropbox.com/home
7. Everything - jetbrains youtrack (2021), https://youtrack.jetbrains.com
8. Folding@home (2021), https://foldingathome.org/about/
9. Go ethereum: Official go implementation of the ethereum protocol (2021), https://geth.ethereum.org/
10. Hedgedoc (2021), https://docs.hedgedoc.org/
11. Home - welcome to mysqueezebox.com! (2021), https://mysqueezebox.com
12. Jetbrains (2021), https://www.jetbrains.com
13. Jupyter blog (2021), https://blog.jupyter.org
14. Local folding@home web control (2021), https://client.foldingathome.org
15. Minio — the minio quickstart guide (2021), https://docs.min.io/
16. Production-grade container orchestration (2021), https://kubernetes.io/
17. Spring — home (2021), https://spring.io
18. Welcome to steam (2021), https://store.steampowered.com
19. Zoom: Video conferencing, screen sharing (2021), https://zoom.us
20. Acar, Gunes and Huang, Danny Yuxing and Li, Frank and Narayanan, Arvind and Feamster, Nick: Web-based attacks to discover and control local iot devices. In: Proceedings of the 2018 Workshop on IoT Security and Privacy. pp. 29–35. ACM (2018)
21. Afek, Y., Bremler-Barr, A., Noy, A.: Eradicating attacks on the internal network with internal network policy. CoRR **abs/1910.00975** (2019), http://arxiv.org/abs/1910.00975
22. Chromium: Chromium's response to an issue (2021), https://github.com/WICG/private-network-access/issues/38#issuecomment-766720523
23. Chromium, Mike West and Titouan Rigoudy: Private Network Access (2021), https://wicg.github.io/private-network-access/
24. Dikla Barda, Roman Zaikin, Y.S.: Keeping the gate locked on your iot devices: Vulnerabilities found on amazon's alexa (2020), https://research.checkpoint.com/2020/amazons-alexa-hacked/
25. Dorsey, B.: Attacking private networks from the internet with dns rebinding (2018), https://medium.com/@brannondorsey/attacking-private-networks-from-the-internet-with-dns-rebinding-ea7098a2d325
26. Hemmings, J.: Ee 4gee mobile wifi router – multiple security vulnerabilities writeup (2017), https://blog.jameshemmings.co.uk/2017/08/24/ee-4gee-mobile-wifi-router-multiple-security-vulnerabilities-writeup/#more-276
27. IANA: IPv4 Special-Purpose Address Registry (2021), https://www.iana.org/assignments/iana-ipv4-special-registry
28. IANA: Ipv6 special-purpose address registry (2021), https://www.iana.org/assignments/iana-ipv6-special-registry
29. IETF: The Web Origin Concept (2011), https://tools.ietf.org/html/rfc6454
30. Jazzy: How your ethereum can be stolen through dns rebinding (2018), https://blog.hacker.af/how-your-ethereum-can-be-stolen-using-dns-rebinding

31. kumar: Dropbox open redirect - misconfigured regex (2017), https://www.kumar.ninja/2017/07/dropbox-open-redirect-misconfigured.html
32. Leitschuh, J.: Zoom zero day: 4+ million webcams & maybe an rce? just get them to visit your website! (2019), https://infosecwriteups.com/zoom-zero-day-4-million-webcams-maybe-an-rce-just-get-them-to-visit-your-website-ac75c83f4ef5
33. Milne, J.: Jetbrains ide remote code execution and local file disclosure (2016), http://blog.saynotolinux.com/blog/2016/08/15/jetbrains-ide-remote-code-execution-and-local-file-disclosure-vulnerability-analysis/
34. Mozilla : Preflight request (2019), https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request
35. Mozilla: Content Security Policy (2019), https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
36. Mozilla: Cross-Origin Resource Sharing (CORS) (2019), https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
37. Mozilla: Same-origin policy (2019), https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
38. Ormandy, T.: Avast: A web-accessible rpc endpoint can launch "safezone" (also called avastium), a chromium fork with critical security checks removed. (2015), https://bugs.chromium.org/p/project-zero/issues/detail?id=679&redir=1
39. OWASP: Cross-Site Request Forgery (CSRF) (2018), https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
40. OWASP: Cross-site Scripting (XSS) (2018), https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
41. PORTSWIGGER: CSRF Tokens, https://portswigger.net/web-security/csrf/tokens
42. Rowan Merewood, C.: SameSite cookies explained (2019), https://web.dev/samesite-cookies-explained/
43. Röttger, S.: Ycm remote code execution vulnerability (2014), https://groups.google.com/g/ycm-users/c/NZAPrvaYgxo/discussion
44. WHATWG: HTML: Secure Context (2021), https://html.spec.whatwg.org/multipage/webappapis.html