Master's Thesis Cyber Security

# Local Network Access Standard: A Security Evaluation

Stefan van Ieperen
s1045156

24 February 2026

*Supervisor:*
Güneş Acar

*Second reader:*
Xavier de Carné de Carnavalet

Radboud University

## Abstract

Following the disclosure of large-scale web-to-app tracking by Meta and Yandex, browser vendors are adopting the Local Network Access (LNA) standard to mitigate cross-context privacy risks. The LNA standard, like its predecessor Private Network Access (PNA), aims to restrict how websites can communicate with devices that are considered local. It achieves this by requiring the user to give explicit permission. While these permission prompts effectively mitigate traditional DNS rebinding attacks, they offer less protection against adversaries coordinating both a website and a native application. By merely exchanging a short unique identifier, such attackers can easily link web and native identities on mobile devices.

In this work, we review known bypasses to the LNA standard and introduce a novel technique that further circumvents its protections. This new bypass uses the Server Name Indication (SNI) field in a failed TLS handshake as a side-channel to allow for one-way communication. The effectiveness of this channel is greatly increased when using a DNS wildcard, configured such that any subdomain resolves to localhost.

We also crawl 45,000 websites to chart which of them make requests to local devices and for what reasons. Many such requests were found, which are mostly caused by bot detection scripts, developer error or attempted communication with a native client. One website was found to send requests similar to the discovered SNI bypass, however our investigation was inconclusive.

We conclude that while LNA specification is a step towards a safer internet, its current iteration contains several flaws. On a technical level, we recommend to display the permission prompt *before* a local TLS connection is even attempted. This would mitigate our bypass. From a usability perspective, the prevalence of false positives risks inducing prompt fatigue, where especially less technical users habitually accept any permission prompt.

# Contents

# Chapter 1

# Introduction

In June 2025, researchers publicly disclosed large-scale tracking on hundreds of millions of Android users by Meta and Yandex [1]. Both tech conglomerates provide tracking SDKs that website owners can install to better understand their visitors' online behavior [2, 3]. These scripts also unknowingly connected to associated native Android applications (e.g. Instagram) in order to exchange tracking cookies. By exchanging these cookies, the more persistent Android Advertising ID (AAID) can be linked to the ephemeral web IDs, thus creating a far better advertising profile on a user. These web-to-app communications bypass many privacy countermeasures, such as clearing browser cookies, using incognito mode or running the browser on a separate Android profile [4]. This novel tracking method was dubbed *Local Mess*.

It is clear that the companies went to great lengths to cover their tracks. On the web SDK side Meta abuses STUN, a protocol normally used for establishing peer-to-peer connections. Such traffic does not appear on Chrome's regular debugging tools, which may be the reason why it was chosen in the first place [1]. The Yandex SDK first retrieves obfuscated parameters from an external server and then sends these via HTTP and HTTPS to its native applications. On the application side, in both cases, a trigger is required before the app starts listening for incoming communications. The Yandex application waits several days before it begins to listen, whereas Meta's applications require a user to be logged in, before it eventually activates [1].

While Meta and Yandex stopped these tracking practices shortly after they were disclosed, browser vendors also decided to intervene. Chrome and Firefox disallowed the ports involved and Chrome additionally blocks the specific STUN protocol abuse that Meta utilized [1]. Short term, these measures prevent the offending tech companies from restarting these exact tracking practices. However, rigorous long term solutions are desirable in order to prevent the occurrence of new workarounds.

The proposed web standard for Local Network Access (LNA) could be the solution [5]. While the creation of this standard predates the public disclosure of Local Mess, its adoption seems to have been accelerated by the discovery. Days after the disclosure, Chrome announced that it will start shipping LNA in their upcoming browser releases [6]. Firefox also includes LNA in their nightly browser releases and Apple has expressed its intent to support the standard in their WebKit engine [7, 8].

LNA builds upon the work of its predecessor, Private Network Access (PNA) [9]. Under PNA, servers running locally had to explicitly opt-in to allow requests from external sources. This could be achieved by returning the correct HTTP response headers to a CORS preflight request [10]. Putting the responsibility on the receiving party caused a number of compatibility issues [11, 12]. It also does not prevent coordinated attacks such as local mess, as the server can simply include

the required headers. Local Network Access instead puts the responsibility at the user level. Before a website is allowed to communicate with a local device, it must first get explicit permission from the user. This is done by prompting the user with a permission dialog, as shown in Figure 1. Such an approach is not without its own shortcomings. The contents of the dialog may be difficult to comprehend for a non-technical user and excessive prompting may cause consent fatigue, where a user will blanket allow any prompt they encounter.

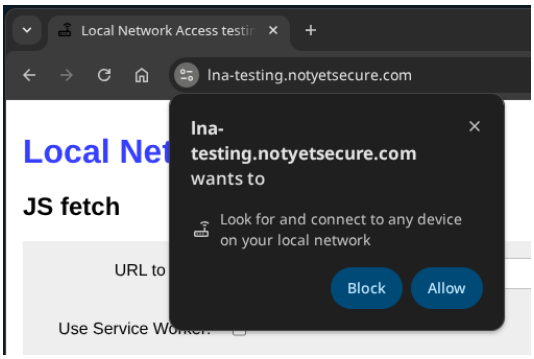


Figure 1: The Local Network Access dialog in Chromium 144.0. Triggered by a JavaScript `fetch` request to `http://localhost:8000`.

Blocking local network access entirely is not possible, as there are legitimate use cases for LNA. Some identity providers incorporate a request to an authentication client running locally as part of their login flow [13, 14]. Conversely, applications such as eduVPN use OAuth to delegate authentication. Here, a successful login on the web gets forwarded to the native application [15]. These are instances of requests to localhost, but there also exist valid use cases for accessing other devices on the local network. For example, Plex is a popular software that lets users run their own media server. These servers run locally, but can be accessed via the public Plex website and are therefore subject to LNA restrictions [16].

In the aforementioned examples, the user may expect a website to request Local Network Access, but there have also been cases were the dialog was unexpected, leading to confused end users [17]. Such occurrences are likely caused by JavaScript code loaded in by third-parties via advertisements.

## 1.1 LNA Specification

In this section, we will dissect the Local Network Access specification to understand when exactly a request is considered an LNA request. We start by looking at how the IP address space is handled. Note that the specification is not yet finalized. It is actively being worked on and therefore it does not yet cover all web protocols. Current browser implementations primarily focus on HTTP(S) requests, which are described [5]. In Section 3.2.1, we will elaborate on what protocols the specification currently does not cover.

### 1.1.1 IP address space

The Local Network Access specification distinguishes between three different address spaces [5].

**Loopback** Addresses only accessible on the local host. For example `127.0.0.1`.

**Local** Addresses that have meaning only within the current network. Depending on network configurations, this could for example be `192.168.1.1`.

**Public** All other IP addresses. These addresses refer to the same target globally on the IP network. For example, `https://ru.nl` resolves to 131.174.78.60 (at the time of writing), which can be accessed from anywhere.

The address space of an IP address can be determined by the following steps [5]:

1. If an address falls in the `::ffff:0:0/96` range, then it is an IPv6 address which maps to an IPv4 address. In this case, we can take the last 32 bits of the IPv6 address and continue.
2. For each row in Table 1:
   - If the IP address falls in the described range, return that row's adress space
3. Return *public*

What this means is that if a public website tries to access a resource on the local network, it requires the user to allow Local Network Access beforehand. The same holds for requests to the address `127.0.0.1`, which go to localhost. Requests originating from a local address targeting a public one, will not be subject to LNA restrictions. For example, if someone is developing a website locally and this website loads a picture from a public location, then no LNA permission is needed. This is because in this case, the request is going from a private context to a more public one.

| Address block | Name | Reference | Address space |
|---|---|---|---|
| 127.0.0.0/8 | IPv4 Loopback | [RFC1122] | *loopback* |
| 10.0.0.0/8 | Private Use | [RFC1918] | *local* |
| 100.64.0.0/10 | Carrier-Grade NAT | [RFC6598] | *local* |
| 172.16.0.0/12 | Private Use | [RFC1918] | *local* |
| 192.168.0.0/16 | Private Use | [RFC1918] | *local* |
| 198.18.0.0/15 | Benchmarking | [RFC2544] | *loopback* |
| 169.254.0.0/16 | Link Local | [RFC3927] | *local* |
| ::1/128 | IPv6 Loopback | [RFC4291] | *loopback* |
| fc00::/7 | Unique Local | [RFC4193] | *local* |
| fe80::/10 | Link-Local Unicast | [RFC4291] | *local* |
| fec0::/10 | Site-Local Unicast | [RFC3513] | *local* |
| 0.0.0.0/32 | IPv4 null IP address | [RFC1884] | *loopback* |
| 0.0.0.0/8 | IPv4 null IP addresses | [RFC1884] | *local* |
| ::/128 | IPv6 unspecified address | [RFC1884] | *loopback* |
| 2001:db8::/32 | IPv6 documentation addresses | [RFC3849] | *local* |
| 3fff::/20 | IPv6 documentation addresses | [RFC9637] | *local* |
| ::ffff:0:0/96 | IPv4-mapped | [RFC4291] | see mapped IPv4 address |

Table 1: Non-public IP address blocks [5]

In the following chapter, we will cover related works. This includes an analysis of the Local Mess research, as well as an overview of software exploits that relied on access to local devices. Often these use a method called DNS rebinding. In Chapter 3, we summarize which bypasses are currently known and explain which techniques we use to attempt to find new bypasses to the LNA checks. We then introduce a new bypass which uses the Server Name Indication (SNI) field of a failed TLS handshake. Chapter 4 contains an explanation of our crawling setup, which we use to crawl the top 45,000 websites to study the nature of any local requests they make. In Chapter 5 we review these local requests in detail, making a distinction between direct requests to local IP addresses and requests to domains that resolve to local IP addresses. Finally, in Chapter 6, we summarize our findings and provide directions for future work.

# Chapter 2

# Related work

## 2.1 Covert tracking on Android devices

In 2025, Vlummens et al. found that Meta apps (like Facebook and Instagram) and Yandex apps (like Yandex Maps and Yandex Browser) were secretly monitoring user activity via a local server on Android devices. By running this local server, the apps can receive browsing data such as cookies from websites that a user visits on their phone. This web-to-app communication works even when browser cookies are cleared, incognito mode is used or the browser is installed on a separate Android profile [4]. In this section we will describe in technical detail how Meta and Yandex performed this tracking.

### 2.1.1 Meta's tracking activities

The Meta Pixel is a small piece of JavaScript code that is included in many websites. It tracks users to provides advertisers with insight on how effective their ad campaigns are. Additionally, it gives website owners information about how users navigate their website [2]. The Meta Pixel is prevalent across the internet, with roughly a quarter of the top 1 million websites using it in 2024 [18]. The widespread usage of the technology allows Meta to track users on a large part of the internet. The Meta Pixel generates a `_fbp` cookie, which is a first-party cookie with a lifespan of 90 days. In theory, the fact that the cookies are first-party, should mean that each website gets a unique cookie and therefore sessions can't be linked. In practice, this restriction is bypassed by sending these unique session cookies to the same destination on the local device. The Meta Pixel sends this cookie in a STUN message, which is a protocol that does not show up in Chrome's developer tools [4].

The receiving end works as follows: any Android application with `INTERNET` permissions can start a server that listens on the localhost interface. The Meta apps used this to listen on UDP ports in the range 12580–12585. Here, it can receive incoming STUN messages, which contain the `_fbp` cookie generated by the Meta Pixel script. Once received, this cookie is then sent to a centralized Meta server for further aggregation [4].

### 2.1.2 Yandex's tracking activities

Russian tech giant Yandex also provides a tracking solution called Yandex Metrica. Similar to the Meta Pixel, this is also JavaScript code that tracks the effectiveness of advertisements and user's browsing behavior [3]. The Yandex applications listen on TCP ports 29009, 30102, 29010 and 30103. The former two ports are being used for HTTP traffic, while the latter two are used for HTTPS. Instead of connecting to localhost directly, Yandex Metrica uses the `yandexmetrica.com` domain,

which resolves to localhost. This additional step makes detection harder. The application will send encrypted device IDs back to the Metrica script, which in turn sends it back to a centralized server. This differs from the Meta implementation, where the application sends data to the central server, instead of the website [1].

## 2.2 DNS Rebinding

DNS rebinding is an attack that attempts to evade the Same Origin Policy (SOP). With SOP, a malicious website is not allowed to send requests to any other domain, unless this domain specifically allows it [19]. This means that a website does not have access to e.g. your router's web portal. DNS rebinding can be used to bypass these protections by changing where a domain name resolves to on-the-fly. The attack works as follows:

1. The attacker has malicious DNS setup for their own domain `evil.com`.
2. The DNS server is configured to respond with the IP address of the public server that hosts a webpage containing malicious JavaScript.
3. The attacker tricks a victim into navigating to `evil.com`.
4. The initial DNS response contains a very short Time To Live (TTL) record and therefore the browser does not keep the IP address in cache for long.
5. The malicious JavaScript waits until the DNS cache is expired and then makes another request to the `evil.com` domain.
6. This time, the DNS server responds with another address. For example, it may respond with `192.168.1.1`, which is often used by the router on home networks.
7. Because the domain name, protocol and port are unchanged, the Same Origin Policy does not take effect.

Throughout the years, many critical DNS rebinding vulnerabilities have been found. In 2017, researcher Tavis Ormandy discovered that the Transmission torrent client allows for arbitrary remote code execution via such an attack [20]. The Transmission application handles torrent downloading and seeding in the background by running a daemon process. DNS rebinding allowed a malicious website to communicate with this background process and change Transmission's configuration or start new torrents. One configuration option allows users to run a command once a torrent finishes. Attackers could abuse this to gain remote code execution.

More recently, an exploit has been discovered in the MCP Inspector software [21]. The Model Context Protocol (MCP) is a protocol that provides a standardized way to give an LLM the context it needs to be integrated in software projects. MCP Inspector is a debugging tool that can be used to test and inspect this connection with the AI model. This application also uses the client-server model and due to improper authentication, it too is vulnerable to a DNS rebinding attack. Like the previous example, this allowed for remote code execution [22].

One of ways to protect against a DNS rebinding attack is by validating the origin of a request. When creating a native application, only traffic coming from a trusted origin should be accepted [23]. In practice, verifying whether an origin is trusted, is not always implemented correctly. For example, a vulnerability was recently discovered ASUS's driver management software. The code that validates the origin only checked if the domain name *included* the string `driverhub.asus.com`, which is not a strong enough check. It allows an attacker to host e.g. `driverhub.asus.com.evil.com` which passes the origin validation. Further weaknesses in the ASUS software allowed the security researcher to gain remote code execution [24].

## 2.3 Characterizing localhost traffic

Kuchhal et al. have done an elaborate analysis on what websites communicate with the local network [25]. Using the Tranco top 100k dataset [26], they found that among the top 100,000 websites in 2020 and 2021, there were 116 and 90 sites making local requests, respectively. Notably, in 2020, 7 out of the 10 most popular domains connecting to localhost belonged to eBay. eBay's port scanning activities did not go unnoticed by security researchers, who concluded that eBay is scanning these ports to check whether your PC is running a remote desktop server, such as RDP [27]. Kuchhal et al. categorize the local traffic into four classes [25]:

**Developer error** Several websites in the top 100k dataset make requests to localhost, which were likely used during development. For instance, to tooling that scans for security issues, or that monitors the files for changes to quickly reload. These calls should have been taken out when the web servers were deployed, but were likely missed by the developers.

**Fraud detection** This strategy is primarily used by e-commerce websites to distinguish between legit transactions and fraudulent ones. The researchers found that the local request were part of a service called ThreatMetrix [28], which is also what eBay used in the earlier example.

**Bot detection** The researchers find several sites performing automated bot detection. While the JavaScript code on these sites is heavily obfuscated, the researchers can speculate on what the script is trying to achieve, based on its behavior. The scripts attempt to connect to ports which are either used by well-known malware, or by browser automation tools.

**Native application** Another set of websites were found to communicate with native applications associated with the website itself. These websites were often related to e-commerce, gaming, online communication or media [25].

# Chapter 3

# Methodology

In this chapter we will describe how we evaluate the security of Local Network Access. In particular, we will explain what approaches we tried to find new bypasses to the LNA restrictions. We will also give an overview of currently known bypasses and introduce a new bypass via the Server Name Indication (SNI) field in TLS.

## 3.1 Web Platform Tests

The Web Platform Tests suite is a collection of tests, written to assess the how well web browsers adhere to the specifications of web standards. The goal of the project is to ensure that all websites display their contents in the same way to the user [29]. The tests include the following:

**Web standards** These are documents that describe how different aspects of the web should look or behave. For example, HTML is a standard that is maintained by the Web Hypertext Application Technology Working Group (WHATWG) [30]. Local Network Access is a feature that comes from the Web Incubator Community Group (WICG), a part of the World Wide Web Consortium (W3C) [5]. The purpose of the WICG is to create a platform where people can experiment and discuss new platform features.

**Conformance tests** These are the actual tests that are included in the Web Platform Tests suite. There are several kind of tests; for example performance tests, security tests or CSS tests. There are also tests for the local network access proposal.

**Web browser vendors** The creators of web browsers use the tests to confirm that their implementation of a feature works according to the standard.

We hypothesized that, since these test cover such a wide range of web features, they may be useful in finding previously unknown bypasses to the Local Network Access restrictions. Especially considering that the test suite also contains tests for Local Network Access. We had hoped that combining the collection of feature tests with the detection of LNA could yield interesting results. To that end, we tried changing the URLs in the tests to loopback IP addresses. Unfortunately, this change caused many tests to fail, likely due to differences in how IP addressed are treated compared to the test domains. Moreover, we found that while there are lots of tests in this framework, many of them are entirely irrelevant to our goal. For example, the status of the accelerometer test is not going to provide new insights into Local Network Access. For these reasons we decided to abandon this approach.

## 3.2 Potential bypasses

In this section we will discuss which known bypasses exist and which protocols are out of scope of the current Local Network Access implementation.

### 3.2.1 Protocols not yet subject to LNA

Currently, LNA restrictions are only applied to HTTP(S) requests. The following protocols are not yet in the scope of the LNA specification, though they will be included at a later stage [31].

**WebRTC** A protocol that can be used for peer-to-peer audio and video communication [32]. To establish this direct connection, a subprotocol called STUN can be used, short for Session Traversal Utilities for NAT. STUN uses an external server for establishing the initial connection. One of the ways Meta covertly tracked users, is by running such a STUN server locally, and sending users' cookies to it [4].

**WebSocket** A bi-directional, persistent communication channel that stays open, reducing overhead compared to HTTP [33].

**WebTransport** A modern, more feature-rich successor of WebSockets that is currently still in development [34].

### 3.2.2 Mishandled IP addresses

The IP address space is large and there exist many addresses that are reserved for special purposes. We have looked at several of these special addresses, both in the IPv4 and IPv6 space, to see if they could be used to bypass LNA. To determine this, we used a Local Network Access test website, also referred to by Google in their developer blog on LNA [6]. With this website it is possible to trigger various kinds of LNA requests, such as HTTP with fetch, WebSockets or WebTransport. We consider the following IP ranges for potential bypasses:

**0.0.0.0** In Private Network Access (the predecessor of LNA), it was discovered that requests to an IPv4 addresses consisting of all zeroes could bypass the restrictions [35]. We tested if this bypass had perhaps been missed when implementing Local Network Access, but testing reveals that this bug has been successfully patched after its discovery; the LNA permission dialog does show up.

**::/0** Similar to its IPv4 counterpart, the IPv6 address containing all zeroes is unable to bypass the LNA implementation.

**::ffff:7f00:0000/112** These are the IPv6 addresses that map to the IPv4 loopback range of 127.0.0.0/8. These were not able to bypass LNA detection. This is not a surprise, as Table 1 suggests that these mapped addresses are converted to IPv4 and are then subject to the same restrictions as a regular IPv4 address.

**2000::/3** IPv6 Global Unicast Addresses uniquely identify a device. It was recently reported that these addresses can be used in the SDP message of the STUN protocol, thereby bypassing LNA restrictions [36].

### 3.2.3 The multicast Domain Name System

mDNS is a UDP protocol that resolves domain names to IP addresses, just like regular DNS. Unlike regular DNS however, mDNS operates by sending IP multicast messages to all machines in the same subnetwork. The machine whose hostname matches the queried hostname can then simply respond with its IP address in another one-to-many message. As this protocol requires hardly any setup from the user, it makes it suitable for devices like printers or smart speakers [37]. In this

normal mode of operation, Local Network Access triggers as expected. If a website requests access to an mDNS domain, e.g. `printer.local`, the IP address of this domain gets resolved to a local address via mDNS broadcast messages. Since the address is determined to be local, an LNA dialog is shown.

On IPv4, mDNS uses the `224.0.0.251` multicast address and on IPv6 it uses `ff02b::fb`. According to the specification of mDNS, any domain name ending in `.local` is considered to be a link-local domain that is to be used for mDNS [37]. The fact that any domain that needs to be resolved will be broadcast to the local network, allows us to bridge the gap between browser and native application on mobile devices. The idea is as follows; inside of a web page, an attacker places a piece of JavaScript that will create a unique alphanumeric identifier. Information about the user's browsing session will be tied to this identifier. Then the JavaScript code makes a request to `<unique-id>.local`, which will get broadcast to all devices on the local network, as it is an mDNS query. The unique ID can easily have enough entropy such that no device on the network will have this hostname, and thus no reply will be sent. An application that is in the control of the attacker can actively listen to the mDNS broadcasts and it can store the unique identifiers it sees [38]. The application then only needs to send this identifier, along with its own persistent application identifier to a centralized server. On this server, the application user data and the browsing data can then trivially be linked.

## 3.3 Bypassing LNA with the TLS Server Name Indication

During our research, we observed an unexpected behavior when attempting to connect to a local HTTP server using an HTTPS request. Such requests will inevitably fail, because there is a mismatch between these protocols. The client wants to create an encrypted communication channel, but the server only knows plain HTTP, which is not encrypted. This discrepancy is detected when the client initiates a TLS handshake by sending a *Client Hello* message. The server does not expect a TLS message, thus it responds with an error that terminates the handshake. Chrome draws the conclusion that no connection has taken place and it does not prompt the user for an LNA permission.

Indeed, no successful connection occurred. However, there is a way we can manipulate the data in the initial TLS message, thus creating a one-way communication channel from the client to the server. The *Client Hello* message contains a field called the Server Name Indication (SNI). Normally, this field allows a server hosting multiple sites to identify the requested domain name and present the corresponding TLS certificate before the encrypted connection is fully established.

In Figure 2, We send an HTTPS request to the domain `lna.quest`, a domain that we own, which resolves to `127.0.0.1`. Wireshark is recording traffic on the loopback interface. It observes a successful TCP handshake and subsequently a failed TLS handshake. Because Wireshark understands the anatomy of the TLS messages, it automatically extracts the SNI from the encrypted packet. For the HTTP listening end, we use Python's SimpleHTTP server that prints all traffic it receives to the terminal. We have highlighted where the SNI appears in this raw binary output. Since the domain name is under our control, we can use it to transmit data, without triggering LNA.

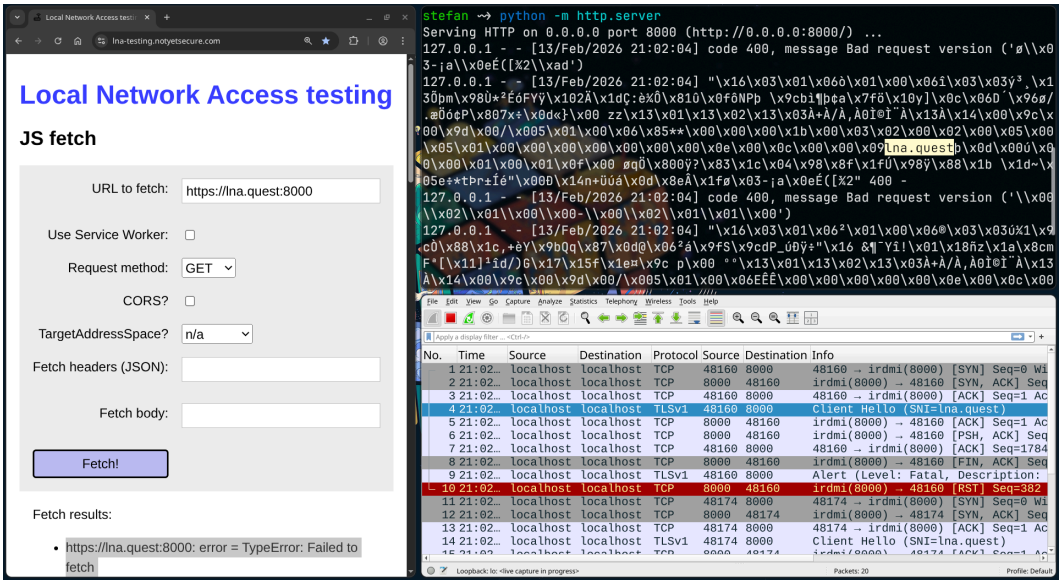


Figure 2: Left: we use a public website to create an HTTPS request to our local server. This request fails, because the TLS handshake could not be completed. Top right: the HTTP server receives the connection attempt, but rejects it. Raw binary data is printed to the terminal, the domain name is highlighted. Bottom right: Wireshark sees the connection attempt and is able to extract the Server Name Indication field from the TLS *Client Hello* message. It also captures the fatal TLS Alert and the subsequent TCP reset packet, which terminate the connection. A second connection attempt follows, which is likely the browser reattempting to establish a connection automatically.

We can improve the effectiveness of this channel by configuring a DNS wildcard for our domain. This means that any subdomain to `lna.quest` will also resolve to `127.0.0.1`. For example, `foo.lna.quest`, `foo.bar.baz.lna.quest` and `really-long-subdomain.lna.quest` all resolve to localhost. The theoretical maximum length allowed for a full domain name is 255 characters, with a 63 character maximum for each subdomain [39]. In practice however, most DNS servers implement security extensions that take up some of the available space, shortening the length to 253 characters [40]. The allowed alphabet for a domain name consists of case-insensitive letters, numbers and the hyphen character, totaling 37 characters.[1] This allows for a substantial amount of data to be transmitted per request.

With this bypass, it is therefore trivially possible to transfer unique identifiers, allowing privacy violations such as Local Mess [1]. To show that this scenario is possible, we developed a proof of concept Android application that hosts an HTTP server. The server receives HTTPS traffic and filters out any messages that start with the byte value `0x16`, as this is indicative of a TLS *Client Hello* message [41]. Then, it parses out the Server Name Indication and shows this on screen. Figure 3, shows a split screen[2] view with our application on the left and a Chrome window on the right. The browser window makes a request to `https://sni-extraction-can-also-bypass-lna-on-android.lna.quest:8080`, which resolves to localhost and is then extracted by our application.

[1] In our testing we found that the underscore character also gets resolved by DNS, even though it is not officially supported. This extends the alphabet by 1 character.

[2] Split screen is only used for demonstration purposes. It is not required for the attack to work, as the server can run in the background.

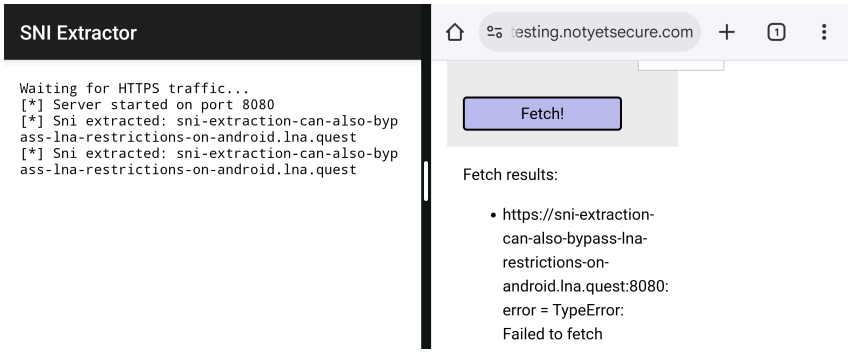


Figure 3: Proof of concept application for the LNA bypass using the Server Name Indication on Android. The duplication of requests is again caused by the browser retrying the failed connection attempt.

The attack was tested on Chromium version 144.0.7559.109 (Linux) and Chrome version 144.0.7559.132 (Android), which at the time of writing are the latest versions available. Firefox nightly also implements LNA restrictions, so we decided to test our bypass against Firefox version 149.0a1 (Linux). Firefox's protections seem marginally better, as the Local Network Access is detected and therefore a pop-up is shown to the user asking for permission. However, the implementation is still flawed; upon denying this permission, the *Client Hello* message is still allowed through. This means our SNI bypass continues to work, albeit less covert due to the fact that the user sees a pop-up initially. Firefox Nightly on Android does not have the LNA restrictions enabled by default, but after enabling them in version 149.0a1 (Android), the same behavior was observed as in its desktop counterpart. We are currently in the process of performing a responsible disclosure to the editors of the Local Network Access specification.

Fundamentally, the requests going to `lna.quest` are local network Requests. The website we use to send the requests has an IP address that is more public than the target IP `127.0.0.1` [5]. This raises the question: if the specification is clear that, in this case, the request is considered a Local Network Access, then why is there no permission dialog? This likely is a result of the fact that the LNA specification is still a work in progress. In contrast to the IP address space definition, the description of how to perform the actual LNA checks is rather ambiguous. The section on integrating Local Network Access with `fetch` contains annotations such as *"What follows is a sketch of a potential solution"* and *"(...) this is still handwaving to some extent"* [5]. The current lack of clarity is possibly what caused the difference in implementations between Chrome and Firefox; it should be more clear when and how the actual LNA check should be performed. To mitigate the Server Name Indication bypass we described in this section, it would make sense to detect (and restrict) Local Network Access exactly after the TCP handshake completes. A successful completion of the TCP handshake indicates the presence of a listening server. Requiring the LNA permission at this point would prevent TLS *Client Hello* messages, and therefore the SNI, from being sent. In the past, one of the editors has expressed intent to move the LNA check to before a connection is established, but provided no timeline as to when this would be implemented [42].

## 3.4 Using the IP loopback range as a side-channel

Implementing the Local Network Access checks before the TLS handshake effectively prevents the SNI side-channel, however, we can still transmit some data in the lower-level packets of the TCP handshake. For historical reasons, the loopback range of `127.0.0.0/8` is very large. Any

request sent to this range will be routed to localhost[3] and we can use this to send an identifier. The server that receives a request can simply read the last 3 octets of the target IP address as a unique identifier. With a bandwidth of 24 bits, this allows for roughly 16.8 million different values. This attack is still possible if the LNA check is performed directly after the TCP handshake, albeit in a less covert manner. During the TCP handshake, the target IP address can already be extracted. However, with our suggested countermeasure, the user will see the Local Network Access permission dialog afterwards.

[3]With the exceptions of `127.0.0.0` (network address) and `127.255.255.255` (broadcast address)

# Chapter 4

# Web Measurements

The next phase of our research is conducting a crawl of popular websites, to determine if any of the bypasses that we discussed are used in-the-wild. The crawler we chose to use for this is Tracker Radar Collector (TRC) [43], an instrumented headless crawler created by DuckDuckGo. The authors of the crawler use it to automatically generate a data set of hidden trackers among popular websites. One way this data can then be used is to generate blocklist for these kinds of trackers [44]. TRC uses a modular design, which can be extended with custom *collectors*. Each one of these collectors collects data such as cookies, web requests or even screenshots of the crawled page. To orchestrate the headless browsers, Tracker Radar Collector makes use of the Chrome DevTools Protocol [45].

## 4.1 Crawling setup

For our crawls with Tracker Radar Collector, we created a custom collector which we will call `LNAQueryCollector`, the source code of this collector is available in Appendix A. It has two purposes:

1. It injects JavaScript code into the crawled web pages that overrides the permission system. Whenever a web page queries the status of the Local Network Access permission, the script will return that this permission was granted. The idea is the following; if the permission is accepted, no pop-up will be shown to a user. A (malicious) script could thus check the status of the LNA permission and only activate its payload once it sees that it is granted.
2. The script keeps track of which origins check the status of the Local Network Access permission. The intercepted access attempts are stored in a separate JSON file for each website.

## 4.2 Capturing packets during the crawl

Tracker Radar Collector captures HTTP(S) requests and responses. For this research, we decided to simultaneously capture lower-layer packets while a website is being crawled. This allows us to capture non-HTTP protocols, which may be used to covertly send data to the local network. An illustrative example of this can be found in the Local Mess research, where Meta used a non-compliant Session Description Protocol (SDP) message to bypass detection. SDP is typically used to establish a WebRTC connection between two hosts, but here it was presumably used because it does not show up in Chrome DevTools [1]. Using the packet capture will paint the full picture, as it should reveal these kinds of covert channels that still allow localhost communication to take place, if they exist.

### 4.2.1 nsntrace

During the setup of the crawler, we investigated using Linux's network namespaces functionality in order to run multiple parallel crawls on the same machine. Network namespaces allow isolation of network traffic, but they require manual setup of virtual ethernet interfaces, assignment of IP addresses to those interfaces, configuration of the routing table and establishment of NAT rules, so that network traffic from outside can reach the right namespace. This setup is rather involved. There is however a command line tool that automates the process. `nsntrace` is a tool that performs the above steps, runs a user-provided command and then cleans up the newly created namespace. In our research, we attempted to use this program, but after a lot of trial and error we did not manage to complete a successful crawl. It seems that the tool does not setup routing for `127.0.0.1` or `localhost`. Listing 1 shows that none of the `ping` packets are able to reach localhost, thus making `nsntrace` unsuitable for this research.

```
$ sudo nsntrace -d any ping -c4 127.0.0.1
Starting network trace of 'ping' on interface any.
Your IP address in this trace is 172.19.237.255.
Use ctrl-c to end at any time.

PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3070ms

Finished capturing 50 packets.
```

Listing 1: Running `ping` to localhost with `nsntrace` on the any pseudo-device. Observe that all `ping` packets are lost. The 50 packets that are captured by `nsntrace` are unrelated to our `ping` command

### 4.2.2 tcpdump

Ultimately, we decided to capture the packets with the command line tool `tcpdump`. Although we are interested in localhost traffic, capturing only on the `lo` (loopback) interface would not be sufficient. These kind of captures do not include packets to e.g. `224.0.0.251`, which would mean that the mDNS bypass discussed in Section 3.2.3 would not be caught. Instead, we capture on the `any` pseudo-interface, which captures the traffic to all interfaces. To reduce the noise somewhat, we filter out any traffic from ports 22 or 9222. The former is used for SSH and the latter is localhost traffic from Tracker Radar Collector to its browser instance. Because we capture all packets when we visit a website, we can no longer run our crawler in parallel. This would intermingle the packet capture and attributing a packet to a certain website would become difficult. Instead, we perform the crawls sequentially with `tcpdump` ran separately on each website.

Sending all Chrome DevTools Protocol traffic to port 9222 is not the default behavior of Tracker Radar Collector. Given the crawler's support for parallelism, hard coding the port may lead to issues since multiple crawler instances would attempt to connect to the same (potentially occupied) port. Instead, TRC handles this by setting the port to 0, as shown in Listing 2. Port 0 is a reserved OS option that automatically allocates an available port for the process attempting to connect. For our crawls, we modified the code to always connect to port 9222, as this is the default port for the Chrome DevTools Protocol. In our sequential runs, this will not cause trouble and it allows us to easily discard the CDP traffic with `tcpdump`.

```
chromeArguments.push(`--remote-debugging-port=0`);
```

Listing 2: The default way Tracker Radar Collector assigns an available port in the file `LocalChrome.js`. In our sequential crawls we fix the port to 9222 so that it can be filtered out from our packet capture easily.

## 4.3 Dataset

The dataset we chose to use in this study is the Tranco 100k. This dataset ranks the top 100,000 websites based on their traffic metrics. Its stability and resilience against manipulation make it suitable for research-oriented purposes [26]. Another benefit of using Tranco is that the study of Kuchal et al. uses the same dataset, dated in 2021 [25]. Because they use an older version, we can make a comparison to see if the amount and behavior of localhost traffic has changed over the years.

The Tranco dataset is updated daily and can be downloaded from their website [46]. For our research, we use the version dated 23 December 2025. We retain the 45,000 highest rated sites of the list. Then we split it into 9 parts such that each crawler instance gets its own sublist of 5,000 websites to crawl.

## 4.4 Server infrastructure

For conducting the crawls, we opted for 9 virtual private servers (VPSs) running on the cloud platform DigitalOcean. Each server has two virtual CPUs, 4Gb of RAM and 80Gb of storage. The full specifications of the servers can be found in Table 2. To ensure that all internet traffic is captured by `tcpdump`, we enable IPv6 on our servers. For the location of the servers, we chose New York (US). This way, any websites we crawl do not have to adhere to the General Data Protection Regulation (GDPR) of the European Union, which may limit local connections websites make for advertising or tracking purposes. The crawls were performed with the Chrome browser, because this is the only browser that has implemented Local Network Access in its stable release. Furthermore, Chrome has a majority of the browser market share, so our crawls will represent most users' experiences.

| | |
|---|---|
| **Operating System** | Debian 13 |
| **Linux kernel** | 6.12.57 |
| **# vCPUs** | 2 |
| **Region** | New York |
| **RAM** | 4Gb |
| **Storage** | 80Gb |
| **IPv6** | Enabled |
| **node version** | 20.19.2 |
| **npm version** | 9.2.0 |
| **Chrome version** | 143.0.7499.109 |

Table 2: Specifications of the virtual private servers used in the web crawls

To run the crawlers, we first copy our custom version of TRC, along with the respective Tranco URL list file into the home directories of the newly created servers. A detailed explanation of the commands that were ran can be found in Appendix B. Tracker Radar Collector outputs a JSON file for every website that it crawls. This file contains the data that each collector collected, as well

as some metadata about the page visit, such as start and end time. We run `tcpdump` separately on every website, so that it too creates individual files as opposed to one large capture file. Though Tracker Radar Collector supports passing in a URL file as a parameter, our decision to collect packet captures individually restricts us to passing the URLs one-by-one instead. Thus starting a separate TRC instance for every URL. While we did not perform rigorous testing, a limited test of 20 URLs took 736 seconds when passing the URLs individually and 671 seconds when passing the URLs in a file. This suggest a time penalty of roughly 9.7%. We acknowledge that this is not insignificant, however we believe that this is an acceptable trade off for the convenience of having a single capture file per URL. To run our crawler, we create a script that iterates over all of the URLs in the URL file and runs `tcpdump` and TRC in tandem. The script can be found in Listing 11.

# Chapter 5

# Results

The result of our scrape is nine directories filled with a combined total of 45,000 packet captures and 45,000 JSON output files from Tracker Radar Collector. The packet capture files collectively take up 278.09GB, compared to 2.52GB output from TRC. While the packet capture files are very large, a lot of this traffic is not very interesting to us. For example, HTTPS traffic is encrypted and therefore not meaningful, unless destination of such a request is a local address. To circumvent this issue, we apply some extra processing steps with `tshark`, a command line application for manipulating packet captures. With this tool we can extract only the fields we are interested in, thus greatly reducing the size of the data we want to work with. Another benefit of using `tshark` is that it can output the data in text format, making it easy to process them with Unix tools. The JSON output from TRC can be directly analyzed with these tools, which will be described in the following section.

## 5.1 Analyzing the Tracker Radar Collector output

To aggregate the output of our custom `LNAQueryCollector`, we will use the command in Listing 3. This command did not return any results, which means that out of all the sites we scraped, not a single one ever checked the status of the Local Network Access permission. This was rather surprising to us,[4] especially because of the large number of sites crawled. Moreover, querying the Local Network Access Permission is also mentioned by developers and large tech companies like Microsoft as a way to provide users with additional context for why your website requires Local Network Access [47, 48].

```
find -name "*.json" | xargs jq '.data."lna-queries" | if length > 0 then . else empty end'
```

Listing 3: This command finds all JSON files in the current directory and then applies `jq` (a JSON processor) to output the result of the `LNAQueryCollector`, if it is non-empty

Next we looked at the results of one of the default collectors that comes with Tracker Radar Collector; the `RequestCollector`. Among other things, this collector logs HTTP(S) and WebSocket data and provides the origin of the request. The collector also shows how a request was initiated. This could be from a JavaScript file, or because the main page includes a resource, such as an image. For us, the interesting requests are those with a destination IP addresses in the local range,

[4] So surprising in fact that we double-checked the validity of our result with a different command: the JSON output of Tracker Radar Collector always represents an empty JSON object as `{}`. We searched all files for the following pattern: `"lna-queries": {[^}]`. Here the caret symbol negates the match on the closing curly brace, thus matching only files where the `lna-queries` field is non-empty. This command also returned nothing, thus confirming our result.

as defined in Table 1. To extract those particular requests, we use regular expressions. Instead of using `grep`, we opted for a more modern alternative called `rg` [49]. This largely functions the same, but it is faster and uses the regular expression syntax of the Rust programming language, which we are more familiar with. Additionally, it searches for files recursively by default. Table 3 shows the commands that we used to extract the requests to local IP addresses, as well as how many results each command gave.

### 5.1.1 Analyzing the types of local requests

By far the largest numbers of requests get made to addresses in the loopback range. All of these were made to `127.0.0.1` specifically. At a distant second place in number of requests, we find the private range `10.0.0.0/8`. All 54 requests originate from the domain `funcionpublica.gov.co` and attempt to retrieve images from the address `https://10.116.8.2:8443`. Since this address is private, these requests are a developer error. The singular mDNS address found in our crawls is also the result of an error. This request is made from the website `jamesclear.com` to the domain `jamesclear.local`, erroneously pointing to some local development environment.

| Data | Extraction command | # Results |
|---|---|---|
| 127.0.0.0/8 addresses | `rg '"url": ".*?127\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}.*?"'` | 520 |
| 10.0.0.0/8 addresses | `rg '"url": ".*?10\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}.*?"'` | 54 |
| localhost | `rg localhost` | 28 |
| mDNS addresses | `rg -i '"url": ".+?\.local.*?"'` | 1 |
| 192.168.0.0/16 addresses | `rg '"url": ".*?192\.168\.[0-9]{1,3}\.[0-9]{1,3}.*?"'` | 0 |
| 172.16.0.0/12 addresses | `rg '"url": ".*?172\.(1[6789]\|2[0-9]\|3[01])\.[0-9]{1,3}\.[0-9]{1,3}.*?"'` | 0 |
| local IPv6 address | `rg ::1` | 0 |
| 0.0.0.0 address | `rg 0\.0\.0\.0` | 0 |

Table 3: Commands used to extract requests to local IP addresses in TRC's output

Though many of the JavaScript files responsible for making local requests are heavily obfuscated, we made a best-effort attempt at classifying these following the same categories as earlier work Table 4 [25]. We see several websites such as `ieee.org` and `sberbank.ru` make requests to the same paths. These requests were made by JavaScript files located at the relative path `/TSPD`. Kuchal et al. also found scripts at this path, which they attributed to the bot detection service from a company called F5 [25, 50]. Interestingly, in our work, the paths that these scripts send requests to are different. This suggests that F5's method for bot detection has changed throughout the years. We also observe new bot detection software on the websites `unsri.ac.id` and `av.by`, coming from the open source SafeLine Web Application Firewall [51]. It sends requests to port 9222, which is typically used for the Chrome DevTools Protocol. Likely for this reason, it is also able to detect, and prevent access to a website when the Chrome debugging tools are opened. An interesting outlier is the domain `d21.team`, which sends an HTTP redirect request to the address `127.0.0.1:1`, only when connecting via an IP address in the United States. Since port 1 is restricted in most browsers, this request will never succeed. The domain seems to be related to digital piracy, so this is likely an attempt to evade detection by authorities or copyright owners.

| Domain | Protocol | Address | Port | Paths | Category |
|---|---|---|---|---|---|
| funcionpublica.gov.co | HTTPS | 10.116.8.2 | 8443 | /documents/* | Developer error |
| saby.ru<br>sabyd.ru<br>sbis.ru | WS<br>WS<br>WS<br>WSS<br>WSS<br>WSS | 127.0.0.1 | 8201<br>8203<br>8205<br>9208<br>9210<br>9212 | /?appName=Plugin | Native application |
| sunrise.ch<br>mcgill.ca<br>ieee.org<br>sberbank.ru<br>datart.cz<br>sbrf.ru<br>vodafone.net.tr<br>president.gov.by<br>t-mobile.pl<br>orange.pl<br>mid.ru<br>fgov.be | HTTP | 127.0.0.1 | 8888 | /200<br>/302<br>/403<br>/404<br>/random_url<br>/400_random_url_with_numbers_403 | Bot detection |
| surferseo.com | HTTP | 127.0.0.1 | 5500 | /src/unwrapped-2025.js | Developer error |
| readli.net<br>megagroup.ru | HTTP | 127.0.0.1 | 47001 | / | Bot detection |
| vseinstrumenti.ru<br>av.ru<br>perekrestok.ru<br>5ka.ru<br>onlinetrade.ru<br>megamarket.ru<br>sbermegamarket.ru<br>megamarket.ru | WSS | 127.0.0.1 | 135, 3389, 49664<br>5900, 5901, 7070 | / | Bot detection |
| hola.org<br>holax.io<br>c6gj-static.net<br>kbz0pwvxmv.com<br>su89-cdn.net<br>x-cdn-static.com<br>yg5sjx5kzy.com<br>zspeed-cdn.com | HTTP | 127.0.0.1 | 6880–6889 | /callback.json?find_port=1 | Native application |
| d21.team | HTTPS | 127.0.0.1 | 1 | / | † Detection Evasion |
| iwin.com | HTTP | 127.0.0.1 | 2080–2082 | /data | Native application |
| iqiyiq.com<br>qy.net | HTTP | 127.0.0.1 | 16422, 16423 | /get_client_id?callback=json*<br>/get_client_ver?<br>callback=app_sdk_callback* | Native application |
| unsri.ac.id<br>av.by | WSS | 127.0.0.1 | 9222 | / | Bot detection |
| brightvpn.com | WS | 127.0.0.1 | 4560 | / | Native application |
| xiaohongshu.com | HTTP | 127.0.0.1 | 9222<br>54345 | /json/version<br>/ | Bot detection |
| ximalaya.com | HTTP | 127.0.0.1<br>127.0.0.1<br>127.0.0.1<br>localhost | 35000<br>40000<br>45345<br>50213 | /<br>/<br>/<br>/status | Native application? |
| dmed.technology<br>dtci.technology<br>spotify.net | HTTP | 127.0.0.1 | 8769, 65111, 65121<br>65131, 65141, 65151 | /probe | Native application |
| qq.com | HTTP | 127.0.0.1 | 11601 | /check | Native application? |
| browserscan.net | WSS | localhost | 22, 3389 | / | ? |

| Domain | Protocol | Address | Port | Paths | Category |
|---|---|---|---|---|---|
| salyk.kg | HTTPS | localhost | 24738 | /JCWebClient.js | Native application |
| raspe.io<br>upraspe.io | HTTP | localhost | 80 | /public/<br>uploads/10426072025111455.png | Developer error |
| gettr.com | HTTP | localhost | 1337, 1338, 2000<br>2222, 3000, 4040<br>4444, 45454, 45691<br>8000, 8080, 80<br>8443, 9222 | / | Bot detection? |
| europepmc.org | HTTP | localhost | 8000 | /piwik.js | Developer error |
| quillbot.com | WS | localhost | 5786 | /qb | Native application |
| theleague.com | HTTP | localhost | 80 | /wp-content/themes/theleague/images/close-button.svg | Developer error |
| slow-manga.com | HTTP | localhost | 80 | /slow/wp-content/<br>uploads/2023/06/3009979_153511.jpg | Developer error |

Table 4: Categorization of private IP requests, following the categories from Kuchal et al. [25]. † does not fit in any category and is therefore labeled *Detection Evasion*

## 5.2 Analyzing the packet captures

Analyzing the Packet captures proved challenging, because `tcpdump` captures all traffic, including packets from automated scanners that probe our crawler instances. For example, we found excessive amounts of Kerberos traffic, likely due to Active Directory related vulnerability scanners [52]. To manage the large volume of data, we apply selective filters, allowing us to focus on one protocol at a time. Using the `tshark` command in Listing 4, we extract all DNS queries that each capture contains. We cross-referenced these with the JSON output of the corresponding TRC visit, to see if this domain was indeed requested by the website, or if the DNS query was random noise. This was done for all domains that resolved to local IP addresses. Table 5 contains an overview of these domains.

```
tshark -r file.pcap \
  -T fields -E header=y -E separator=, \
  -Y dns -e ip.addr -e _ws.col.info > output.txt
```

Listing 4: Commands used to extract DNS queries in each packet capture file. The resulting files were subsequently analyzed with the same `rg` commands as in Table 3

| Domain | Protocol | Address | Port | Paths | Category |
|---|---|---|---|---|---|
| dmed.technology<br>dcti.technology<br>spotify.net | HTTPS | 00o2qlm8bj1rxlbxr0x7<br>.authenticatorlocalprod.com | 8769, 65111, 65121<br>65131, 65141, 65151 | /probe | Native application<br>(Okta Fastpass) |
| onpwe.club<br>shafa.com<br>cubetv.fun<br>piojm.tech | HTTPS | www.beian.gov.cn | 443 | /img/ghs.png | Developer error? |
| setapp.com | HTTPS | setapp.click | 2432, 6385, 18671<br>29302, 43724 | / | Native application |
| onlinekhabar.com | HTTPS | analytics.onlinekhabar.com | 443 | / | Native application |
| 56.com | HTTPS | localhost.tv.sohu.com | 7981 | /status?jsonp=jsonp*&_=* | Native application |
| myapp.com | HTTPS | *.sdk.myapp.com | 12900–12909 | / | ? |
| onlinedown.net<br>360kuai.com<br>360.com | HTTPS | local.info.g9hc4.cn | 51360, 54360 | /?callback=<br>uuidjsonpcb2020&_=* | Native application |
| weiyun.com | HTTPS | localhost.weixin.qq.com | 13013–13015<br>14013–14015 | /api/check-login | Native application |
| ipcheckhost.com | HTTPS | *.t.iprify.com | 443 | /css/null.css?t=* | † Utility |

Table 5: Categorization of requests to domains that resolve to `127.0.0.1` following the categories from Kuchal et al. [25]. Several domains in the Tranco list resolve directly to localhost, these are not included, but instead discussed in Section 5.2.1. † Tests for DNS leakage

### 5.2.1 First-party domains resolving to private IP addresses

There are 58 domains in the top 45,000 domains that directly resolve to `127.0.0.1` (listed in Appendix C). We believe that these are included because of a limitation of the Tranco dataset, specifically that it does not consider subdomains. For example, it includes the domain `127.net`, which resolves to localhost. Its subdomain `urswebzj.nosdn.127.net` is not included, even though it is likely responsible for the popularity of the `127.net` domain. This subdomain is used as a Content Delivery Network (CDN) for the popular Chinese news site `163.com`, which is also in the top 45,000 websites. Another possibility for the occurrence of these local domains is that some of them were seized by authorities because they provided illegal content, such as digital piracy. To disable these seized domains, ISPs sometimes resolve their IP address to localhost [53]. For the domain `kaspersky-labs.com`, we were able to determine that it facilitates communications between the Kaspersky web extension and its native application [54].

### 5.2.2 myapp.com

This website redirects to `sj.qq.com`; one of the leading Chinese app stores owned by Tencent [55]. Then, it creates 10 localhost requests to ports 12900–12909. Interestingly, these requests are sent to a wildcard DNS domain that contains a unique identifier in the URL. Not unlike our attack described in Section 3.3. We analyzed the JavaScript files responsible for these requests.

We identified that the requests are initiated by a file called `user-portrait-sdk.iife.js`. Before executing the requests, the script queries `yybadaccess.3g.qq.com/pcyybopen/getsdkhost`, another Tencent domain, where it retrieves the 'sdkHost' variable. This variable is in the following format: `<uid>.sdk.myapp.com`, where `uid` is a 48 character hexadecimal string. Then ten HTTPS GET requests are made to this sdkHost. Because DNS records for `*.sdk.myapp.com` are configured to resolve to `127.0.0.1`, all requests are sent to localhost.

During our testing, we discovered that after some timeout, another three HTTPS requests are made to the `<uid>.sdk.myapp.com` domain, with a fresh value for for the `uid`. Additionally, the domain makes three secure web socket connection attempts to the domain `<uid>.wss.myapp.com` (also resolving to localhost), with yet again a different `<uid>`. This `uid` is found in JSON data on the main page of the `sj.qq.com` domain and is also used as headers in other HTTP requests, where it is referred to as `ual-access-signature-token`. These six new requests are initiated by a different script, one found between HTML `script` tags in a file called `core.html`.

| Domain | Ports | Initiator | uid origin |
|---|---|---|---|
| `https://<uid>.sdk.myapp.com` | 29000–29009 | `user-portrait-sdk.iife.js` | retrieved from `https://yybadaccess.3g.qq.com/pcyybopen/getsdkhost` |
| `https://<uid>.sdk.myapp.com` | 29000–92002 | Script in `core.html` | retrieved from `https://yybadaccess.3g.qq.com/pcyybopen/getsdkhost` |
| `wss://<uid>.wss.myapp.com` | 25120–25122 | Script in `core.html` | Included in JSON data on the `sj.qq.com` main page with 6 seemingly random characters appended to the end |

Table 6: Targets and initiators of localhost requests on `sj.qq.com`

These requests display very similar behavior to the Local Network Access bypass described in Section 3.3 and can be used to share unique tracking IDs between web and mobile contexts. We see HTTPS traffic to a unique subdomain that resolves, via a DNS wildcard, to localhost. The secure WebSocket requests that the domain creates also work for our bypass, since these start out as HTTPS requests. Under normal circumstances such an HTTPS request gets upgraded by an 'HTTP 101 switching protocols' response [33]. However, when using WebSockets for the SNI bypass, the connection gets terminated before that happens. Still, this initial HTTPS message requires a TLS handshake, which allows the bypass to succeed.

Like Meta and Yandex, Tencent has its own analytics tool that users can add to their website called Real User Metrics (RUM) [56]. However, this SDK is not responsible for these requests, therefore this is not a Local Mess scenario. The requests are only made on a first-party basis and not by this third-party SDK on any other websites. We speculate that the requests may still be made for tracking purposes, but without knowing the receiving end of these requests, it is hard to say anything concrete.

Tencent (partially) owns vast numbers of applications, among which the hugely popular QQ browser and the "super app" WeChat [55]. While performing an in-depth analysis on a multitude of applications is out of scope for this work, a limited investigation into the QQ store Android application is possible. We speculate that this native version of the QQ website has the highest probability of being the recipient of the observed requests. The application cannot be downloaded from the Google Play store, but is instead available on the `sj.qq.com` website.[5] Upon running the app, we used `netstat` via ADB to determine that it does not listen on ports 29000–29009 or 25120–25122. It does however listen on ports 51023 and 19714 (Listing 5), but it is possible that these port numbers have been randomly assigned. The app may not listen on our expected ports, because the right conditions for enabling the local server haven't been met yet. Perhaps we did not interact with the application yet in the right way, or it hadn't been installed long enough. It could also be that the listening end is a different application entirely.

```
$ adb shell su netstat -untap | grep -wE '2900[0-9]' # no results
$ adb shell su netstat -untap | grep tencent
tcp6  0  0 ::ffff:127.0.0.1:51023  [::]:*     LISTEN  28300/com.tencent.android.qqdownloader:daemon
udp   0  0 0.0.0.0:19714           0.0.0.0:*          28345/com.tencent.android.qqdownloader
```

Listing 5: Commands used to determine on which ports the QQ application listens

[5] At https://sj.qq.com/appdetail/com.tencent.android.qqdownloader.

### 5.2.3 ipcheckhost.com

Table 5 shows that ipcheckhost.com also send requests with an identifier in the subdomain. This website is used to display several properties of the device that visits it, such as its location, public IP address and whether a proxy is detected. The local requests it sends are seemingly used to detect DNS leakage. First, the site sends a request to a custom API endpoint at `iprify.com/api/v2/dnsleak`. In the query parameters of this request, it includes 8 randomly generated subdomains of `*.t.iprify.com`, which resolve to localhost. Presumably this service then checks how these domains were resolved. This type of traffic again falls outside of any of the four categories of local traffic, proposed by Kuchal et al. [25].

### 5.2.4 DNS amplification attack

During our analysis one domain's DNS records stood out. The domain `bumm.live` does not resolve to any website on the internet. Instead, this domain has 1080 DNS A records associated to it, all pointing to unique addresses in the `192.168.0.0/16` range. This domain is likely used for an attack called DNS amplification. The idea is that an attacker sends a relatively small request to a DNS server, with a spoofed source IP address. The small request then generates a lot of data, that could cause a denial of service to a target. An attacker can also leverage a botnet to vastly increase the amount of traffic the target receives. This traffic is hard to filter out, because it all originates from legitimate DNS servers [57]. It does appear that the domain can be used for this purpose. Asking a server for all the A records does not require much traffic, while getting 1080 IP addresses in the DNS answer is rather substantial. The domain has recently also been added to a popular DNS blocklist [58]. Note that this domain was directly included in the TRANCO 100k list, as opposed to being requested by one of the websites on the list. This popularity may be an indicator that the domain has already been used for attacks.

### 5.2.5 pixel.advertising.com

During our analysis of domains resolving to local addresses, we found that the domain `pixel.advertising.com` resolves to `192.168.18.7`. Requests to this domain were made by five different websites, including large sites such as `eurosport.de`. The path of each request is `/ups/28/sync`, followed by a unique identifier in the query parameters. These requests were initiated by a script loaded in from the third-party domain `discovery.demdex.net`, which is part of Adobe's ad network [59].

All of these facts are indicative of cookie syncing, a technique used in online advertising that allows different ad companies to share and synchronize user data stored in cookies across multiple websites. However, it remains puzzling as to why the domain would then resolve to this particular private IP address. The domain was also mentioned in an online forum, where it is suspected to be an instance of DNS rebinding [60]. We believe this is unlikely, given the high monetary value of the `advertising.com` domain, the origin of the request that we mentioned in the previous paragraph and the fact that the DNS record has been unchanged since December 2024.[6] Instead, we believe that this local request is a result of a misconfiguration.

### 5.2.6 WebRTC

To determine whether any of the websites used WebRTC to bypass LNA, we first extracted all STUN traffic from the packet capture files. To do this, we ran the command in Listing 4. This gave us a large number of individual files. On these, we again used `rg` to filter out only the interesting

[6] According to https://dnshistory.org/dns-records/pixel.advertising.com

addresses. In this case, these are the local IP address ranges `127.0.0.0/8` and `::1`. Additionally, to check if the IPv6 GUA bypass is used, we filter for communications from the nine unique addresses back to themselves. All three commands give no output, which indicates that these bypasses are no longer used in-the-wild.

```
tshark -r example.pcap -T fields -E header=y -E separator=, \
  -Y stun -e ip.addr -e _ws.col.info > example-webrtc.pcap
```

Listing 6: Command ran on every PCAP file to extract WebRTC traffic

### 5.2.7 mDNS

To further analyze any mDNS traffic, we used the `tshark` command in Listing 7, which. First, we searched these files for the IPv6 mDNS address `ff02::fb`. This gave exactly one result, which is the website www.jamesclear.com, that we had previously also found in the analysis of the Tracker Radar Collector JSON files. Next we searched for the `224.0.0.251` IPv4 address. Surprisingly, this returned a large number of results.

```
tshark -r example.pcap -q -z conv,ip -z conv,ipv6 > example.ip-conv.txt
```

Listing 7: Command ran on every packet capture file that extracts the source and destination IP addresses of any communication that occurred while visiting a website

In total, there are 607 sites whose packet captures contain mDNS IPv4 traffic. None of these websites directly query `.local` domains, according to the Tracker Radar Collector JSON files. After investigating a handful of websites manually, while running Wireshark to inspect the packets being sent, it seems that these websites consistently cause the mDNS traffic. Visiting douglas.de for example, caused an mDNS request `156090b9-ee41-47dd-9851-1a13677888e2.local`. As discussed in Section 3.2.3, this request can be used to bypass Local Network Access. We believe it may also be likely that the requests are used for fingerprinting, based on the smart devices in the local network. However, due to time limitations and code obfuscation, we could not identify the exact purposes of these mDNS lookups.

# Chapter 6

# Conclusion

In this work, we assessed the security of the new Local Network Access permission. To make this assessment, we started by providing an overview of currently known bypasses. We also introduced a new bypass, which uses the Server Name Indication field in a failed TLS handshake. Subsequently, we scraped 45,000 websites to see if any of the bypasses occur in-the-wild. These crawl results also gave insights in why websites make local requests and showed us that earlier classification efforts of local requests are still largely relevant.

We believe that requiring permission for Local Network Access strengthens privacy and security. Its current implementation provides an effective countermeasure against the traditional DNS rebinding attacks described in Chapter 2. However, at the moment there are some flaws. If an attacker coordinates both the web and native applications, then they can abuse protocols to exchange information. The Local Mess research has shown that this data only needs to be a few bytes in size in order to uniquely identify users [4]. In this thesis, we have demonstrated that several such bypasses exist. To prevent the Server Name Indication bypass, we recommend the authors change when the LNA permission is requested. Currently, this happens after the TLS handshake successfully completes (and not at all if it fails). But it should happen after a successful three-way TCP handshake, *before* the TLS handshake is even attempted. That way, the TLS *Client Hello* message, which contains the SNI field, is only sent when the user gives permission.

In general, when introducing a new user permission, browser vendors should be wary of consent fatigue. If a user gets pop-ups too frequently, they may habitually give permission in all cases. Paradoxically, that can lead to worse security. This is especially true for a technical permission such as Local Network Access. Browser vendors and authors of the specification need to strike a balance between tightening security and not allowing for too many false positives. Admittedly, this is a difficult problem.

## 6.1 Discussion

With our analysis of the crawled data, we had three goals in mind:
1. Determine if any known bypasses were used in-the-wild;
2. Determine if the are classifications of local requests by Kuchal et al. are still relevant [25];
3. Investigate new bypasses in the recorded traffic.

For the first goal, we largely focused on local IP addresses, IPv6 Global Unicast addresses and mDNS addresses. The latter two were found in the crawls, but not in a way that constituted an LNA bypass. We saw lots of traffic to local addresses, which mostly consisted of HTTP(S). In many cases, this traffic is used for either bot detection or communicating with a native application.

Traffic from the Chinese app store myapp.com sends identifiers in a subdomain which resolves to localhost. We conclude that this could be used to bypass LNA via the Server Name Indication. However, we can't be certain of this, without knowing the receiving end of the communication.

The localhost traffic we found is overall very similar to the earlier classification [25]. In certain cases, even the same sites or bot detection provider was found. Our crawl contains no instances of the Fraud Detection category, but this is actually in line with the earlier findings. In their work, crawls were performed on Windows, Linux and Mac devices. Only the Windows machine showed Fraud Detection, which likely explains why we do not observe this kind of traffic in our Linux crawl. Interestingly, we do observe several instances of Bot Detection, a category which Kuchal et al. also observed exclusively on Windows. Additionally, we see some of these bot detection requests are going to port 9222, which is used for the Chrome DevTools protocol. This port was not seen at all in the previous work, indicating that this type of bot detection may be a recent development. In our crawl, we observed two websites making requests that do not fit into any categories. We have labeled these *Detection Evasion* and *Utility*.

Our packet capture files proved to be a bit of a mixed bag. On the one hand they proved to be very useful in analyzing the DNS requests in order to extract domains that resolve to local IP addresses. This is one of the techniques that Yandex used to cover their tracks in the Local Mess tracking [4]. Unfortunately, there were also some concerns with the data organization of the packet captures. Each crawled site results in a PCAP file from `tcpdump` and a JSON file from Tracker Radar Collector. In several instances, the packet captures contained DNS requests for a domain that was not requested in the corresponding JSON file. Therefore, extracting these DNS queries required manual cross-referencing to verify the validity of the DNS request. We also think think our initial scope for finding bypasses was too wide. It was unfeasible to find a new bypass by inspecting these packet capture files, because the amount of data was simply too large. We think this approach makes more sense in a targeted search for e.g. one specific protocol. This would then have the added benefit of allowing more `tcpdump` filters to be defined, thereby reducing the output file sizes. For research not focused on local traffic, `nsntrace` may prove very useful as isolating the process should help tremendously in reducing the noise in packet captures.

## 6.2 Future work

The investigation into `sj.qq.com` (`myapp.com`), was left open-ended. This is primarily due to study limitations and excessive number of possible applications to test. Performing in-depth analysis on Android applications is out of scope for this work. Still, the website is doing something out of the ordinary and therefore we believe it would be an interesting direction for future work. In the research by Vlummens et al., a pool of instrumented smartphones is used; as applications are executed, synthetic user interactions are performed using Android Monkey [4, 61]. A similar setup could be used here in order to trigger potential listening behavior of candidate applications.

In this research, the focus was on desktop browsers. These are often the first to receive new implementations of the (not yet finalized) specification. We therefore decided to conduct our crawls on Chrome's desktop Linux version. In the context of the Local Mess tracking, it would have been nice to give the crawler an Android User-Agent header. This makes visited websites think that the requests are coming from an Android device, possibly leading to more attempted communication to native applications. Future work could rerun the crawls, to see if this is indeed the case.

A number of websites were found to cause mDNS traffic, without directly sending HTTPS or WebSocket requests to `.local` addresses. Due to time limitations, we were unable to investigate this traffic in detail. Future work could study this novel form of (suspected) tracking.

### 6.2.1 LNA's protection against DNS rebinding attacks

Earlier, we mentioned that Local Network Access prevents against *traditional* DNS rebinding attacks. This phrasing was actually chosen carefully, because near the end of this thesis a new DNS rebinding attack was discovered that works, even under LNA restrictions [62]. The attack cleverly manipulates a main frame navigation, which is currently out of scope of the LNA specification [5]. In this case, the main frame navigation results in a new window being opened, so the attack is easier to detect for a user. We believe that it is unlikely for such a sophisticated attack to show up in the top 45,000 websites, especially given how new it is. For future research, it would be interesting to verify that the attack works, investigate whether the attack can be improved and see if it appears in any malicious website datasets such as PhishTank or abuse.ch [63, 64].

### 6.2.2 Implementing stricter defenses for loopback requests

In Section 3.4, we describe an attack where the target IP address in the loopback range is used to transmit a unique identifier. One possible defense against this side-channel is to require user permission before even initiating the TCP handshake. However, a consequence of this is that any attempted request would show an LNA pop-up, even if no listening server is present. In our crawls, this amounts to 73 local network access attempts made by high-traffic websites. Considering that there is already a risk of consent fatigue, we believe that introducing more false positives is undesirable. Another possible countermeasure is for browsers to restrict loopback requests, such that only requests to `127.0.0.1` are allowed. By convention, this address is the most commonly used and this is reflected in our crawl results. We see no requests to any other addresses in the `127.0.0.0/8` range. This suggests that, at least for these high-traffic websites, such a change would not have any major ramifications. More work should be done to investigate whether this change impacts any other parts of the web.

# Bibliography


[1] A. Girish, G. Acar, N. Vallina-Rodriguez, N. Weerasekara, and T. Vlummens, "Covert Web-to-App Tracking via Localhost on Android." Accessed: Sep. 08, 2025. [Online]. Available: https://localmess.github.io/#contact

[2] Meta, "Meta pixel: Measure, optimise and retarget ads on Facebook and Instagram." Accessed: Dec. 04, 2025. [Online]. Available: https://en-gb.facebook.com/business/tools/meta-pixel

[3] Yandex, "Яндекс Метрика – система веб-аналитики." Accessed: Dec. 04, 2025. [Online]. Available: https://metrika.yandex.ru/promo/product

[4] T. Vlummens, A. Girish, N. Weerasekara, F. Z. Borgesius, G. Acar, and N. Vallina-Rodriguez, "Bridges to Self: Silent Web-to-App Tracking on Mobile via Localhost."

[5] W3C Web Incubator Community Group, "Local Network Access." Accessed: Jan. 08, 2026. [Online]. Available: https://wicg.github.io/local-network-access/

[6] C. Thompson, "New permission prompt for Local Network Access | Blog." Accessed: Feb. 10, 2026. [Online]. Available: https://developer.chrome.com/blog/local-network-access

[7] Mozilla, "Control personal device and local network permissions in Firefox." Accessed: Feb. 15, 2026. [Online]. Available: https://support.mozilla.org/en-US/kb/control-personal-device-local-network-permissions-firefox

[8] letitz, "Local Network Access." Accessed: Feb. 15, 2026. [Online]. Available: https://github.com/WebKit/standards-positions/issues/163

[9] "Private Network Access on hold." Accessed: Feb. 13, 2026. [Online]. Available: https://developer.chrome.com/blog/pna-on-hold

[10] W3C Web Incubator Community Group, "Private Network Access." Accessed: Jan. 08, 2026. [Online]. Available: https://wicg.github.io/private-network-access/

[11] tvanc, "CORS error on request to localhost dev server from remote site." [Online]. Available: https://stackoverflow.com/questions/66534759/cors-error-on-request-to-localhost-dev-server-from-remote-site

[12] R. Luttino, "PNA (Private Network Access) in Chrome restrictions nightmare - Google Chrome Community." Accessed: Feb. 13, 2026. [Online]. Available: https://support.google.com/chrome/thread/387728146/pna-private-network-access-in-chrome-restrictions-nightmare

[13] Okta, "Configure Chrome to Suppress the Local Network Access Prompt for Okta FastPass." Accessed: Feb. 04, 2026. [Online]. Available: https://support.okta.com/help/s/article/configure-chrome-to-suppress-the-local-network-access-prompt-for-okta-fastpass

[14] Beyond Identity, "Managing the Chrome v141 local network access prompt." Accessed: Feb. 15, 2026. [Online]. Available: https://support.beyondidentity.com/docs/managing-the-chrome-v141-local-network-access-prompt

[15] eduVPN, "Portal Configuration - Documentation." Accessed: Feb. 13, 2026. [Online]. Available: https://docs.eduvpn.org/server/v3/portal-config.html

[16] PlexInfo, "Important note about the Plex Web app - Local Network Access." Accessed: Feb. 15, 2026. [Online]. Available: https://forums.plex.tv/t/important-note-about-the-plex-web-app-local-network-access/933264

[17] thehashimwarren, "I've never seen this before... What does it mean?." Accessed: Feb. 15, 2026. [Online]. Available: https://www.reddit.com/r/webdev/comments/1pu40e8/ive_never_seen_this_before_what_does_it_mean/

[18] Y. Beugin, S. Dutton, Y. Dimova, R. Merewood, and B. Pollard, "Cookies," *The 2024 Web Almanac*. HTTP Archive, 2024. doi: 10.5281/zenodo.14065903.

[19] "Same-origin policy - Security | MDN." Accessed: Feb. 15, 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Defenses/Same-origin_policy

[20] T. Ormandy, "transmission: rpc session-id mechanism design flaw." Accessed: Feb. 22, 2026. [Online]. Available: https://project-zero.issues.chromium.org/issues/42450491

[21] modelcontextprotocol, "MCP Inspector." Accessed: Feb. 22, 2026. [Online]. Available: https://github.com/modelcontextprotocol/inspector

[22] National Institute of Standards and Technology, "CVE-2025-49596." Accessed: Feb. 22, 2026. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2025-49596

[23] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from DNS rebinding attacks," *ACM Transactions on the Web*, vol. 3, no. 1, pp. 1–26, Jan. 2009, doi: 10.1145/1462148.1462150.

[24] mrBruh, "One-Click RCE in ASUS's Preinstalled Driver Software." Accessed: Oct. 13, 2025. [Online]. Available: https://mrbruh.com/asusdriverhub/

[25] D. Kuchhal and F. Li, "Knock and talk: investigating local network communications on websites," in *Proceedings of the 21st ACM Internet Measurement Conference*, Virtual Event: ACM, Nov. 2021, pp. 550–568. doi: 10.1145/3487552.3487857.

[26] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation," in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2019. doi: 10.14722/ndss.2019.23386.

[27] D. Nemec, "eBay is port scanning visitors to their website - and they aren't the only ones - nem.ec." Accessed: Oct. 15, 2025. [Online]. Available: https://blog.nem.ec/2020/05/24/ebay-port-scanning/

[28] "ThreatMetrix: Automated Risk Management & Fraud Detection." Accessed: Oct. 16, 2025. [Online]. Available: https://risk.lexisnexis.com/products/threatmetrix

[29] "web-platform-tests documentation — web-platform-tests documentation." Accessed: Jan. 08, 2026. [Online]. Available: https://web-platform-tests.org/

[30] WHATWG, "HTML Standard." Accessed: Jan. 08, 2026. [Online]. Available: https://html.spec.whatwg.org/

[31] Marten Richter, “About about webtransport.” Accessed: Feb. 15, 2026. [Online]. Available: https://github.com/WICG/local-network-access/issues/10#issuecomment-2940249502

[32] “WebRTC.” Accessed: Feb. 15, 2026. [Online]. Available: https://webrtc.org/

[33] I. Fette, Google, A. Melnikov, and Isode, “The WebSocket Protocol.” [Online]. Available: https://www.rfc-editor.org/rfc/rfc6455.txt

[34] E. Kinnear and V. Vasiliev, “The WebTransport Protocol Framework,” technical report, Oct. 2025. Accessed: Feb. 15, 2026. [Online]. Available: https://ietf-wg-webtrans.github.io/draft-ietf-webtrans-overview/draft-ietf-webtrans-overview.html

[35] A. Lumelsky, “0.0.0.0 Day: Exploiting Localhost APIs From the Browser.” Accessed: Oct. 09, 2025. [Online]. Available: https://www.oligo.security/blog/0-0-0-0-day-exploiting-localhost-apis-from-the-browser

[36] G. Acar, “Consider some reserved IPv6 ranges as local.” [Online]. Available: https://github.com/WICG/local-network-access/issues/15#issuecomment-3225316474

[37] S. Cheshire and M. Krochmal, “Multicast DNS.” [Online]. Available: https://www.rfc-editor.org/info/rfc6762

[38] Gunes Acar, “mDNS access for WebRTC.” [Online]. Available: https://github.com/WICG/local-network-access/issues/22#issuecomment-3223162671

[39] P. Mockapetris, “DOMAIN NAMES - CONCEPTS AND FACILITIES.” [Online]. Available: https://www.rfc-editor.org/rfc/rfc1034.txt

[40] D. Eastlake, “Domain Name System Security Extensions.” [Online]. Available: https://www.rfc-editor.org/rfc/rfc2535.txt

[41] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol.” [Online]. Available: https://www.rfc-editor.org/rfc/rfc5246.txt

[42] H. Chao, “Programmatically trigger the LNA permission prompt.” Accessed: Feb. 11, 2026. [Online]. Available: https://github.com/WICG/local-network-access/issues/44#issuecomment-3314219109

[43] “duckduckgo/tracker-radar-collector.” Accessed: Dec. 17, 2025. [Online]. Available: https://github.com/duckduckgo/tracker-radar-collector

[44] Dax, “DuckDuckGo Tracker Radar Exposes Hidden Tracking.” Accessed: Dec. 24, 2025. [Online]. Available: https://spreadprivacy.com/duckduckgo-tracker-radar/

[45] Google, “Chrome DevTools Protocol.” Accessed: Feb. 11, 2026. [Online]. Available: https://chromedevtools.github.io/devtools-protocol/

[46] V. L. Pochat, T. V. Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, “A research-oriented top sites ranking hardened against manipulation - Tranco.” Accessed: Jan. 07, 2026. [Online]. Available: https://tranco-list.eu/

[47] P. Serban, “Using navigator.permissions.query for Local Network Access in Chrome.” Accessed: Jan. 27, 2026. [Online]. Available: https://paulserban.eu/blog/post/using-navigatorpermissionsquery-for-local-network-access-in-chrome/

[48] Microsoft, “Adapting your website for new Local Network Access restrictions in Microsoft Edge.” Accessed: Jan. 27, 2026. [Online]. Available: https://learn.microsoft.com/en-us/deployedge/ms-edge-local-network-access

[49] A. Gallant, “ripgrep.” Accessed: Feb. 22, 2026. [Online]. Available: https://github.com/BurntSushi/ripgrep

[50] F5 Inc., “Mitigate Bots and Abuse.” Accessed: Feb. 21, 2026. [Online]. Available: https://www.f5.com/solutions/use-cases/bot-management

[51] SafePoint, “Web Application Firewall.” Accessed: Feb. 21, 2026. [Online]. Available: https://safepoint.cloud/landing/safeline

[52] D. Weston, “Microsoft's guidance to help mitigate Kerberoasting.” Accessed: Feb. 11, 2026. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2024/10/11/microsofts-guidance-to-help-mitigate-kerberoasting/

[53] Ofcom, “Site Blocking” to reduce online copyright infringement,” May 2011, [Online]. Available: https://assets.publishing.service.gov.uk/media/5a79583740f0b642860d748f/Ofcom_Site-Blocking-_report_with_redactions_vs2.pdf

[54] “What is kaspersky-labs.com (ff.kis.v2.scr.kaspersky-labs.com).” Accessed: Feb. 10, 2026. [Online]. Available: https://forum.kaspersky.com/topic/what-is-kaspersky-labscom-ffkisv2scrkaspersky-labscom-21484/

[55] L. Jia, D. B. Nieborg, and T. Poell, “On super apps and app stores: digital media logics in China’s app economy,” *Media, Culture & Society*, vol. 44, no. 8, pp. 1437–1453, Nov. 2022, doi: 10.1177/01634437221128937.

[56] Tencent Cloud, “Real User Monitoring.” Accessed: Feb. 10, 2026. [Online]. Available: https://www.tencentcloud.com/

[57] Cybersecurity and Infrastructure Security Agency, “DNS Amplification Attacks | CISA.” Accessed: Jan. 21, 2026. [Online]. Available: https://www.cisa.gov/news-events/alerts/2013/03/29/dns-amplification-attacks

[58] DNSDebunker, “Amplification Attack Domain · Issue #8686 · hagezi/dns-blocklists.” Accessed: Jan. 21, 2026. [Online]. Available: https://github.com/hagezi/dns-blocklists/issues/8686

[59] Adobe, “Understanding Calls to the Demdex Domain.” Accessed: Feb. 20, 2026. [Online]. Available: https://experienceleague.adobe.com/en/docs/audience-manager/user-guide/reference/demdex-calls

[60] “Possible DNS-rebind attack detected - Hardware Hub / Networking.” Accessed: Feb. 02, 2026. [Online]. Available: https://forum.level1techs.com/t/possible-dns-rebind-attack-detected/226711

[61] Android, “UI/Application Exerciser Monkey.” Accessed: Feb. 14, 2026. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[62] B. Gupta, “LNA Blocking bypassable by opening a new window with DNS rebinding.” Accessed: Feb. 14, 2026. [Online]. Available: https://issues.chromium.org/issues/475274565

[63] PhishTank, “PhishTank | Join the fight against phishing.” Accessed: Feb. 14, 2026. [Online]. Available: https://phishtank.org/

[64] abuse.ch and SPAMHAUS, “URLhaus.” Accessed: Feb. 14, 2026. [Online]. Available: https://urlhaus.abuse.ch/

[65] A. Senol and M. Humbert, “Leaky Forms: A Study of Email and Password Exfiltration Before Form Submission.”

# Appendix A

# LNA Query Collector

The below code contains the modified Collector for the Tracker Radar Collector project from DuckDuckGo, inspired by prior work by Senol et al. [43, 65]. Listing 9 gathers information about which parts of a web page request the status of the Local Network Access Permission. It gets injected into every page by the LNA Query Collector in Listing 8.

```
const BaseCollector = require('./BaseCollector');
const fs = require('fs');
const queryInject = fs.readFileSync('helpers/
localNetworkAccessQueryIntercept.js', 'utf8');

class LNAQueryCollector extends BaseCollector {
    /**
     * @param {import('./BaseCollector').CollectorInitOptions} options
     */
    init({ log }) {
        this._log = log;
    }

    id() {
        return 'lna-queries';
    }

    /**
     * @param {import('puppeteer-core').CDPSession} session
     * @param {import('devtools-protocol/types/
protocol').Protocol.Target.TargetInfo} targetInfo
     */
    async addTarget(session, targetInfo) {
        if (targetInfo.type !== 'page' && targetInfo.type !== 'iframe') {
            return;
        }

        this._cdpClient = session;

        await session.send('Page.enable');
        await session.send("Page.addScriptToEvaluateOnNewDocument", {
          source: queryInject,
        });
    }

    async getData(options) {
        const evaluationResult = await this._cdpClient.send(
          "Runtime.evaluate",
          {
            // Can't directly send more complex objects, so `JSON.stringify`
            // them. `Object.fromEntries` is necesarry because the JSON
```

```
            // representation of a Map is it properties and not
            // its key/value pairs
            expression:
              "JSON.stringify(Object.fromEntries(window.lna_queries))",
            returnByValue: true,
          },
        );

        const jsonString = evaluationResult.result.value;

        if (jsonString) {
            const lnaQueries = JSON.parse(jsonString)
            return lnaQueries
        } else {
            return null
        }
    }
}

module.exports = LNAQueryCollector;
```

Listing 8: Source code for the `LNAQueryCollector`

```
(() => {
    // Create a Map that will track which origin requested the LNA permission
    // status. The variable will be exported by `LNAQueryCollector`.
    window.lna_queries = new Map();

    if (navigator.permissions && navigator.permissions.query) {
        const originalQuery = navigator.permissions.query.bind(
          navigator.permissions,
        );
        Reflect.defineProperty(navigator.permissions, 'query', {
            value(descriptor) {
                // Only report the status of the LNA permission
                if (descriptor?.name === 'local-network-access') {
                    const source = getSourceFromStack();

                    // Update origin statistics
                    if (window.lna_queries.has(source)) {
                        window.lna_queries.set(
                          source,
                          window.lna_queries.get(source) + 1,
                        );
                    } else {
                        window.lna_queries.set(source, 1);
                    }
                    // Return the spoofed permission
                    return Promise.resolve({ state: 'granted' });
                }

                // If it is a different permission, return the original
                try {
                    return originalQuery(descriptor);
                } catch (error) {
                    return Promise.reject(error);
                }
            },
            writable: false,
            configurable: false,
        });
    } else {
        console.warn('Permissions API not supported in this environment.');
```

```
    }

    console.log('Added LNA permission detection');

    // Instrumentation code for retrieving the origin of the
    // permission access
    const STACK_LINE_REGEXP = /(\()?(http[^)]+):[0-9]+:[0-9]+(\))?/;
    const getSourceFromStack = function () {
        const stack = new Error().stack.split('\n');
        // remove our own intercepting functions from the stack
        stack.shift();
        stack.shift();
        const res = stack[1].match(STACK_LINE_REGEXP);
        return res ? res[2] : 'UNKNOWN_SOURCE';
    };
})();
```

Listing 9: Script that overrides the Local Network Access query. Gets injected into all webpages by `LNAQueryCollector`

# Appendix B

# Configuring and managing the Virtual Private Servers

```bash
#!/usr/bin/env bash

SESSION="crawler"

commands=(
    "ssh spider1"
    "ssh spider2"
    "ssh spider3"
    "ssh spider4"
    "ssh spider5"
    "ssh spider6"
    "ssh spider7"
    "ssh spider8"
    "ssh spider9"
)

# Check if the session already exists
tmux has-session -t $SESSION 2>/dev/null

if [ $? != 0 ]; then
    tmux new-session -s $SESSION -d

    # 9 windows, so 8 splits
    for i in {1..8}
    do
        tmux split-window -t $SESSION
        tmux select-layout -t $SESSION tiled
    done

    # Connect each window with the corresponding server
    for i in {0..8}
    do
        tmux send-keys -t $SESSION:0.$i "${commands[$i]}" C-m
    done

    # Turn on synchronize-panes so typing in one types in all
    tmux set-window-option -t $SESSION:0 synchronize-panes on
fi

tmux attach -t $SESSION
```

Listing 10: Script used to connect and send commands to all nine virtual private servers

```bash
#!/usr/bin/env bash

set -e -o pipefail

if [ -z "$1" ] || [ -z "$2" ]
  then echo "Please provide the URL_LIST and OUTPUT_DIR parameters"
  exit
fi

URL_LIST=$1
OUTPUT_DIR=$2

mkdir -p ./data/$OUTPUT_DIR
for url in $(cat $URL_LIST); do
    echo crawling $url >> ./data/$OUTPUT_DIR/crawled-urls.txt;

    # Run tcpdump in background with increased buffer and filtering out SSH and CDP noise
    sudo tcpdump -B 8192 -i any "port not 9222 and not port 22" -w "./data/$OUTPUT_DIR/$url.pcap" &

    # Run the crawler on the current url
    npm run crawl -- -u "$url" -o ./data/$OUTPUT_DIR/ --config config.json -v

    # End the tcpdump process
    sudo pkill tcpdump
done
```

Listing 11: Script used to run Tracker Radar Collector and `tcpdump` simultaneously

```bash
apt-get update
apt-get upgrade
apt-get install nodejs npm git rsync tmux

useradd -m -p "" user
usermod -aG sudo user

su user
cd /home/user

# Before this, we copied our version of TRC
# onto the server using rsync
cd tracker-radar-collector
npm install
```

Listing 12: Installation commands ran on each server

# Appendix C

# Domains directly resolving to 127.0.0.1

The following domains resolve to 127.0.0.1 and were present in the 45,000 highest ranked domains from the Tranco 100k list [26].

- 127.net
- amediateka.tech
- ampfeed.com
- clusterdns.net
- coupadev.com
- cvrg7h3nwgyffrwu9iww6.com
- cvrutteyynwfffgyui2w4.com
- da.direct
- datto.io
- dft-cdn42.net
- dmdsdp.com
- domaincontrol.com
- dzcdn.net
- e1c-ops.com
- erinsha.com
- facecast.xyz
- findlaw.cn
- foreverpirates.co
- gfpnet.com
- globaldomainserver.net
- habradns.net
- hoztnode.net
- iothings.site
- iptv2021.com
- ipvanish-stg.com
- istole.it
- kaspersky-labs.com
- kraftonde.com
- livemediahost.com
- mailchannels.net
- maindnstech.com
- musical.ly
- myradns.net
- myworkdayjobs.com
- nbcuas.com
- osnova.io
- others-cdn.com
- oui-0x00199d.com
- pega.st
- photoccino.com
- pknic.pk
- plat.fi
- pshbz.com
- qqqmedia.com
- qrserver.com
- swypeconnect.com
- tencentmusic.com
- trex.media
- turnkeydns.com
- ugfdbijnjhf02jbhdfg9u.com
- uihds73nwgydfhbhuifdf.com
- upsaffiliate.net
- volia.net
- vultrusercontent.com
- wt-eu02.net
- wt-safetag.com
- xnxx-cdn.com
- xvideos-cdn.com