insecure:// Vulnerability Analysis of URI Scheme Handling in Android Mobile Browsers

Abdulla Aldoseri
University of Birmingham, UK
University of Bahrain, Bahrain
axa1170@bham.ac.uk

David Oswald
University of Birmingham, UK
d.f.oswald@bham.ac.uk

***Abstract*—Uniform Resource Identifier (URI) schemes instruct browsers to conduct specific actions depending on the requested scheme. Previous research has addressed numerous issues with web URI schemes (e.g., http: and https:) both for desktop and mobile browsers. Less attention has been paid to local schemes (e.g., data: and file:), specifically for mobile browsers. In this work, we examined the implementation of such schemes in Android OS browsers, analysing the top-15 mobile browsers. As a result, we discovered three vulnerability types that affect several major browsers (including Google Chrome, Opera and Samsung Internet). First, we demonstrate an URI sanitisation issue that leads to a cross-site scripting attack via the JavaScript scheme. The problem affects Chromium browsers including Chrome, Opera, Edge, and Vivaldi. Second, we found a display issue in Samsung Internet that allows abusing data URIs to impersonate origins and protocols, posing a threat in the context of phishing attacks. Finally, we discover a privilege escalation issue in Samsung's Android OS, leading to full read and write access to the internal storage without user consent and bypassing the Android storage permission. While this issue was originally discovered in the file scheme of the Samsung browser, utilising a combination of static and dynamic analysis, we traced the problem back to an authorization issue in Knox Sensitive Data Protection SDK. We then show that any app can abuse this SDK to obtain full access to the internal storage without appropriate permission on Samsung devices running Android 10. We responsibly disclosed the vulnerabilities presented in this paper to the affected vendors, leading to four CVEs and security patches in Chrome, Opera and Samsung Internet browser.**

***Keywords*—Android, mobile browsers, XSS, privilege escalation, URI schemes**

I. Introduction

Mobile browsers are among the most widely used apps on mobile devices. They feature various methods, known as URI schemes, to access Internet (e.g., https:) and device resources (e.g., file:) [1]. Due to the complexity and feature-richness of URI schemes, they also expose substantial attack surfaces. A single flaw in their implementation can jeopardize the security of a user's data, profile, and online accounts and thus also compromise the user's privacy.

In general, URI schemes can be classified into two broad types: web schemes and local schemes. Web schemes are protocols that are used to communicate with online endpoints (e.g., https: and http:), whereas local schemes perform certain client-side operations (e.g., JavaScript: and file:). While web schemes have been extensively scrutinised in the literature, particularly with regard to authentication and authorization procedures, local schemes received less attention. Nevertheless, some notable issues in local schemes have been reported for desktop browsers (e.g., phishing with top-level navigation via data URIs [2, 3] and Cross-Site Scripting (XSS) via encoding JavaScript URIs [4]).

In this paper, we set out to analyse local URI schemes on mobile devices. Our initial assumption was that the specific usage context and the different OS characteristics compared to desktop browsers give rise to new vulnerability types specific to the mobile context.

A. Our contribution

In this paper, we perform a systematic analysis of local browser schemes on mobile devices, focusing on JavaScript, data:, and file: schemes. We concentrate on these three schemes, however, also analysed other schemes (e.g., about:) but did not find significant issues. We then describe several vulnerabilities affecting mobile browsers that we found. In summary, our main contribution are:

First, we discovered a self-XSS issue [5, 6, 7] in the JavaScript scheme. The issue allows executing JavaScript in the context of a loaded web page, e.g., leading to session hijacking. The issue is caused by lack of sanitisation of the clipboard of Android's Input Method Editor (IME) keyboards. Most Chromium browsers are affected by this issue, including Google Chrome, Microsoft Edge, Opera, and Brave.

Second, we demonstrate how data URIs in the Samsung Internet browser can be used to impersonate websites, *i.e.*, render content that seems hosted on a genuine origin. This problem is caused by Samsung Internet's behaviour to display the last URI characters in the address bar, making the data: scheme prefix invisible.

Finally, by examining the file scheme, we uncover a privilege escalation issue in Samsung's Android variant that allows an arbitrary app to access the internal storage of the device without user consent, bypassing the Android storage permission. We trace the issue to Sensitive Data Protection (SDP), a Samsung Knox module.

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2022
28 April 2022, San Diego, CA, USA
ISBN 1-891562-78-9
https://dx.doi.org/10.14722/madweb.2022.23003
www.ndss-symposium.org

B. Responsible disclosure

The vulnerabilities described in this paper have been responsibly disclosed through the respective channels. The JavaScript scheme issues described in Section III-A were reported to Google and Opera on December 1, 2020 and November 25, 2020 respectively. They can be tracked via CVE-2020-6159 for Opera and #1154353 in the Chromium bug tracker. The latter issue is currently embargoed until Google has released a patch. The data scheme issue in Samsung Internet browser described in Section III-B was reported on March 19, 2021, and can be tracked via CVE-2021-25419. Finally, the file scheme vulnerability described in Section III-C was reported on October 30, 2020 and resulted in two CVEs as follow: CVE-2021-25348 for bypassing the permission security check in the browser, and CVE-2021-25417 for the underlying privilege escalation issue in Samsung's Android OS variant.

C. Outline

The remainder of this paper is structured as follows: In section II, we provide a background on browser schemes together with related work regarding their security. Based on this, we define the scope of our work and choose specific schemes for our analysis. In section III, we discuss our analysis and present the discovered issues in widely-used browsers. Subsequently, we assess the impact of the discovered issues and propose mitigations in section IV. Finally, we conclude in section V, discussing limitations and opportunities for future work.

II. Background and Related Work

In this section, we provide an overview of browser schemes and review their security aspects. We then discuss related work in terms of security issues and common weaknesses.

A. URIs

User agents (browsers) accept URIs from users and execute an action based on the URI. RFC3986 defines a URI to have the following structure [8]:

```
URI         = scheme ":" hier-part [ "?" query ]
                [ "#" fragment ]

hier-part   = "//" authority path-abempty
                 / path-absolute
                 / path-rootless
                 / path-empty
```

From the perspective of the browser, the scheme refers to the used protocol in a request (e.g., https but also local schemes like file and JavaScript). The "authority" includes host and port for requests to remote servers.

Browsers adopt the Same-Origin Policy (SOP) as a security model. SOP provides isolation between origins (authorities), *i.e.*, prevent different origins from accessing each other unless they are the same origin or explicitly authorised each other for example via Cross-Origin Resource Sharing (CORS) [9]. SOP prevents network adversaries from compromising retrieved responses from one origin or carrying out actions on behalf of the user from compromised origins (e.g., via JavaScript) [10].

B. Browser schemes

Browsers support different schemes for remote communication (e.g., http, https, ftp) as well as local schemes (e.g., JavaScript, data, file). Unlike remote schemes, these local scheme solely operate on the client machine and do not require internet access. As they operate locally, they may expose local computer privileges which can cause security issues. Therefore, in this paper, we focus on understanding and analysing local schemes from a security perspective.

a) ***The JavaScript scheme:*** is implemented in web browsers applications to execute custom JavaScript from a URI [11]. The scheme uses the following URI format:

```
javascript:<script>
```

where `<script>` is JavaScript code. The scheme has two sequential operations: source text retrieval and in-context evaluation. The first operation retrieves the source text that is included in the `<script>` part of the URI and applies necessary decoding and characters replacements operations to it. Then, the in-context evaluation operation evaluates the generated text. A typical JavaScript scheme example for embedding a script as a hyperlink in an HTML document is as follows [11]:

```
<a href="javascript:doSomething()">click</a>
```

In this example, when the user clicks on the hyperlink, the browser executes the `doSometing()` function in the context of the currently loaded origin. The embedded JavaScript inherit the current origin [12]. The JavaScript scheme can also be invoked (like any URI) from the browser address bar.

b) ***The data scheme:*** renders binary data as-is and allow to include external data [13]. An according URI has the following form:

```
data:[<mediatype>][;base64],<data>
```

`<mediatype>` is a media type specification for the represented data (e.g., txt or png). `base64` indicates if the data is Base64-encoded, otherwise, ASCII encoding is assumed. `<data>` is the payload data itself. The following is an example of rendering a PNG image using the data scheme [13]:

```
<img src="data:image/png;base64,aGVsbGl..." />
```

The maximum length of URIs in browsers imposes a limitation on the size of the `<data>` part in data URIs. Similarly to JavaScript URIs, data URIs can be used from the browser's address bar.

c) ***The file scheme:*** allows accessing files and directories on a host machine. Due to the SOP, remote hosts cannot query the scheme to access local files [1]. A file URI is structured as follow:

```
file://<host>/<path>
```

Where `<host>` is the mount point on the host machine (e.g., drive name or `/sdcard` on mobile devices) and `<path>` is the path to the requested file. Each file accessed by a file URI is assigned a unique origin based on the system's Globally Unique IDentifiers (GUID). This prevents files from accessing each others' contents by means of the SOP policy [1].

C. Related work

In this section, we give an overview of related work regarding previously reported attacks on browsers using URI schemes.

*a) **Vulnerabilities in the JavaScript scheme:*** Improper handling of JavaScript URIs enables attacks including Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS). CSRF refers to an attack where the adversary causes the browser to initiate a request to a certain server without user consent, while XSS is an attack that allows an adversary to inject malicious scripts into the visited (legitimate) website [14]. In both attacks, a victim can be affected by the attacks unintentionally by visiting a vulnerable website or using vulnerable services (e.g., plugins, browsers) [15]. XSS is classified into several subtypes that include reflected XSS, stored XSS, DOM-based XSS and self-XSS [6, 15, 16, 17]. Among them, the one that relates to the JavaScript scheme is self-XSS: it is a social engineering attack in which victims are tricked into executing scripts that compromise their web accounts or leak their data [5, 6, 7]. Figure 1 illustrates an example of a self-XSS attack. First, the adversary sets up a server to accept requests and instructs the victim to use a malicious JavaScript URI ①. The victim accesses a benign target websites ②. Then, they copy-paste the malicious URI into the browser address bar ③. Based on the script, the adversary can forward sessions data and cookies to his server ④. Adversaries rely on obfuscation, minification and encoding of JavaScript code to increase the willingness of the victim to execute the scripts [6]. Cao et al. measured the effectiveness of various obfuscation techniques for JavaScript URIs. They found that obfuscated JavaScript URIs achieve 10% more execution by participants (38.4%) compared to regular JavaScript URIs (29.4%) [6].

In Android, Terada demonstrates an issue with the JavaScript scheme in Google Chrome and Android browsers: because the browsers did not sanitize the JavaScript scheme URIs in incoming Android intents, an adversary can perform XSS attacks by sending multiple intents to the browsers to first request a target domain and then to obtain personal data (e.g., session cookies) via the JavaScript scheme [18].

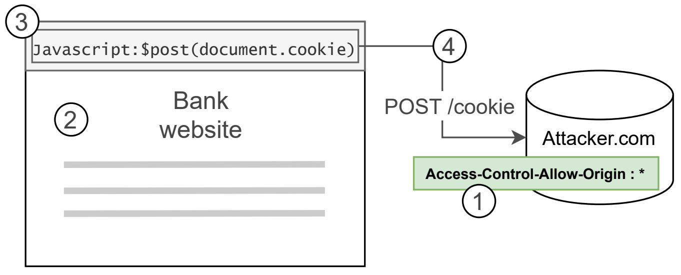

3
Javascript:$post(document.cookie)
4
2
Bank website
POST /cookie
Attacker.com
Access-Control-Allow-Origin : *
1

Figure 1. Scenario showing a self-XSS attack to steal a victim's session cookie. The adversary sets up a server to accepts requests from any origin ①. The victim logs into his bank account ② and copy-pastes a malicious self-XSS URI into the address bar ③. The session cookie is sent to the adversary server via an XML HTTP request ④.

*b) **Phishing with the data scheme:*** In contrast to the JavaScript scheme, the data scheme focuses on data representation. A common issue with it relates to the user clicking e.g., a hyperlink and not realising that this leads to a data URI instead of a web URI [19]. The adversary can then render a phishing page or provide a malicious executable from that data URI. Therefore, most browsers including Firefox and Chromium block top-level navigation to data URIs [2, 3]. In terms of sanitization, Chromium did not sanitise SVG contents in pasted text. This allowed embedding a JavaScript payload in SVG content in a data URI, leading to JavaScript execution in the target document [20].

Address bar spoofing using data URIs refers to displaying data: content that appear to be hosted on a legitimate origin, e.g., showing an attacker-controlled form that seems hosted on `example.com` and thus enabling phishing attacks [21]. Nishimura reported a related bug in Firefox on Android [22]: data URI were persistently shown in the address bar regardless of web navigation. This occurred when it opened from a stored shortcut or from a bookmark intent sent from an Android application. This hides the true origin of the current content. Baloch showed a different attack against Opera, Safari and UC, where data URI content is loaded while the address bar shows a different origin. The issue is due to the browser preserving a target URI in the address bar when requested over an arbitrary port repeatedly [23, 24].

*c) **Privilege escalation with file URIs:*** Mobile browsers expose an interface to accept browsing requests by other apps or web pages. Wu and Chang illustrates a weakness in this feature because of not appropriately sanitising "intents" : if an intent requests a file URI while auto-download is enabled, browsers might download their own private data (e.g., session cookies) to the SD card, allowing malicious apps on the device with storage permission to read these information [25]. Terada reported an issue in Opera with similar consequences: Web pages on Opera can access any private activities in Android Opera browser because of improper filtering. Utilising this issue, allows file URI to access the browser's cookies and render them as an HTML document. A malicious website can set a a malicious JavaScript code in a cookie to execute it when it rendered it as an HTML. Allowing to steal the rest cookies [18].

Improper implementation of SOP in browsers can equally introduce security issues: as demonstrated by Wu and Chang, 63 Android browsers improperly implemented SOP, resulting in leaking sensitive data from local files (e.g., HTML) [25]. In Chrome, Barth et al. showed that remotely hosted XML files can retrieve contents of local files due to the same root issue [26].

*d) **Browsers analysis and detection tools:*** Conventional automated unit tests are often insufficient to detect the aforementioned issues. As a result, the research community has developed according testing and debugging tools. Google developed a remote debugger for troubleshooting Chromium-based mobile browsers and Android Webviews. The debugger provides a developer console for inspecting components and a log monitor [27]. Wu and Chang implemented an automated test tool to check for their file URI issues. The tool interacts with the browser-under-test and the adversary's app using Android Debug Bridge (ADB) and can find certain issues automatically [25].

Browser	JavaScript			Data	File
	Query	Clip-trim	Null origin		
Chrome *C		✓	✓	✓	
Samsung *C	✓			✓	✓
Opera *C		✓	✓	✓	✓
Brave *C		✓	✓	✓	✓
Edge *C		✓	✓	✓	✓
Vivaldi *C		✓	✓	✓	✓
FireFox	✓				✓
FireFox focus	✓				✓
DuckDuckGo	✓			✓	Crash
Mint				✓	✓*2
Mi Browser			✓*1	✓	✓*2
MX6					✓*2
US browser	✓			✓	✓*2
Phoenix browser	✓			✓	✓*2
Dolphin			✓	✓	✓*2

Table I. Summary of how the considered browsers handle JavaScript, file and data schemes. (*C) indicates Chromium browsers. (*1) indicates that JavaScript executed only in the null origin where no website is loaded. (*2) indicate support of the file scheme but without an implementation of an index page that lists the internal storage files.

III. Case studies

We focused our analysis on three browser schemes (JavaScript, data, file), because these have been the most common sources of vulnerabilities in the past and because they expose the largest attack surface compared to functionally limited schemes (e.g., about:). We analysed the implementation of each considered scheme in the top-15 mobile browsers (based on Google Play downloads). As an initial step, we reviewed how each considered browser handles JavaScript, data, and file URIs. The results are shown in Table I and are further elaborated in the following sections.

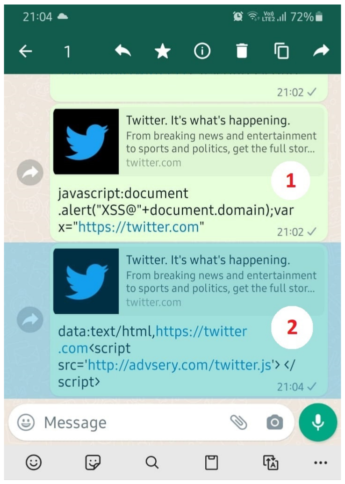
A. Self-XSS using the JavaScript scheme

The JavaScript scheme, as described in Section II, allows JavaScript code to be executed in the context of the currently loaded origin. This enables numerous JavaScript-based threats, in particular self-XSS, which we investigate in the following.

*a) **Threat model**:* We consider the threat model of Facebook, Cao et al., in which an adversary tricks a victim into performing a "self-XSS" by sending them a malicious URI, e.g., through a messenger of social network. The attack does not require the victim to install apps and proceeds as follows:

1) The adversary sends a URI containing the JavaScript scheme with malicious code to the victim and tricks them into copy-pasting the URI.
2) The victim copies the URI using the clipboard and pastes it into a vulnerable web browser.
3) The script is executed, potentially in the context of the current origin.

Obviously, self-XSS requires social engineering: the practice of copy-pasting URIs is common, especially in emails where it is often requested if clicking the URI does not work. The adversary might include the target website's original URI to display its original logo in a social media post or direct message. As can be seen in Figure 2, WhatsApp v2.21.23 (latest version at the time of writing) mis-recognises the URI of Twitter within a malicious JavaScript scheme and displays the website information, potentially misguiding users. Cao et al. have shown that 30% of participants in a self-XSS experiment were deceived by such an attack [6].



21:04
72%
1
21:02
Twitter. It's what's happening.
From breaking news and entertainment to sports and politics, get the full stor...
twitter.com
1
21:02
Twitter. It's what's happening.
From breaking news and entertainment to sports and politics, get the full stor...
twitter.com
2
21:04
Message

```
javascript:document
.alert("XSS@"+document.domain);var
x="https://twitter.com"
```

```
data:text/html,https://twitter
.com<script
src='http://advsery.com/twitter.js'></
script>
```

Figure 2. An example of sharing a JavaScript URI (1) and data URI (2) on WhatsApp. The app recognises only the fake https URI at the end and shows website info, tricking users into believing the URI to be genuine.

*b) **JavaScript scheme implementations in mobile browsers**:* To understand how our selected browsers handle and sanitise JavaScript URIs, we analysed the behaviour by supplying several test URIs to each browser. The results are summarised in the JavaScript column in Table I. Concretely, we found that browsers exhibited one (or a combination) of the following handling practices:

Search query: A JavaScript URI is treated as a search query for the browser's default search engine, but never executed.
Clipboard trimming: The browsers intercepts URI paste events from the clipboard and strips the scheme from the pasted text. This prevents attacks where a victim is lured into copying a URI into the address bar. However, manually typing a JavaScript URI using the keyboard is still allowed.
Only on null origin: JavaScript URI are executed only on a null origin, *i.e.*, when no website loaded in the browser.
Not supported: JavaScript URIs are not not supported at all. This was only the case for the Mint and MX6 browsers.

As can be seen in Table I, Chromium-based browsers adopt the clipboard trimming approach: The browser detects paste events and trims the scheme if required. Nevertheless, we noticed that the Android Clipboard Manager does not provide an event listener for intercepting paste events [28]. Therefore, it appears that Chromium employs a custom solution based on the address bar to sanitise pasted URIs, leading us to a potential security problem:

Potential issue #1: The Android Clipboard Manager does not provide a mechanism to listen to paste events. Chromium-based browsers use a custom solution to sanitise pasted URI, which might contain security issues in the implementation.

*c) **Intercepting clipboard events:*** We focused on Chromium browsers, because they are the only browsers in our set to implement clipboard trimming (apart from Samsung Internet, cf. Table I). Rather than analysing each Chromium browser individually, we examined the Chromium source code as the likely base [29]. From this, we found that the JavaScript scheme is handled as follow:

1) The URI address bar is a custom field that inherits the functionality of `AutoCompleteEditView`.
2) The `onTextContextMenuItem` method is overridden to listen for context menu events (e.g., cut, copy, and paste).
3) The so-called omnibox API sanitises the pasted text, trimming the JavaScript scheme.

We reviewed the omnibox unit test cases and found that the sanitisation properly handles even obfuscated scheme expressions, where special characters or spaces are used, e.g., `java\x0d\x0ascript:alert(0)` and `java script: alert(0)`. However, we noticed that paste interception only works for paste events from the context menu. Therefore, trimming will *not* be applied if the paste events originates from a different source.

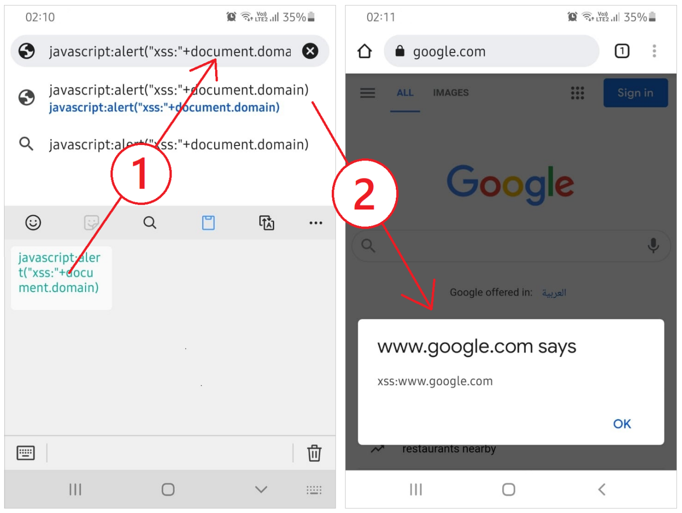
*d) **Bypass clipboard trimming:*** Since Android 7, Android supports Input Method Editor (IME) keyboards [30]. IME keyboards are custom keyboards that extend the functionality of the default keyboard e.g., with emojis and pictures. Several IME keyboards are shipped with a custom clipboard implementation that is not context menu-based. Respective keyboards are pre-installed as the primary keyboard by major manufacturers, e.g., Gboard for Google devices and the Samsung keyboard for Samsung devices.

As suspected, we found that using the paste operation of IME keyboards does not trigger the JavaScript sanitisation of Chromium-based browsers. This issue can thus be exploited to enable self-XSS attacks against Chrome, Opera, Brave, Edge and Vivaldi if the user pastes from the IME keyboard clipboard as shown in Figure 3.

Issue #1: Self-XSS attacks against Chromium browsers are possible if URIs are pasted from IME keyboards

IME keyboards are widely pre-installed by major manufacturers, including Google and Samsung: the Gboard IME keyboard has over 1 billion installs according to Google Play [31]. The issue can be utilized for session hijacking (e.g., stealing session cookies) or general JavaScript code execution in the browser (e.g., to submit forms on behalf of the user without consent and knowledge). In the affected browsers, JavaScript code can access the current session data, including the document components, because the scheme inherits the currently loaded origin. The browser's SOP can be circumvented by configuring the adversary's back-end servers to accept requests from any origin using CORS.

*e) **Disclosure:*** We reported this issue to Opera, and, because it affects all Chromium browsers, subsequently also to the Chromium team. Both vendors confirmed the issue.


javascript:alert("xss:"+document.doma

javascript:alert("xss:"+document.domain)
javascript:alert("xss:"+document.domain)

javascript:alert("xss:"+document.domain)

javascript:alert("xss:"+document.domain)

www.google.com says

xss:www.google.com

Figure 3. An example of self-XSS against Google Chrome with an IME keyboard: pasting a JavaScript URI from the keyboard bypasses sanitisation (1), thus, navigating to the URI causes the JavaScript code to run (2).

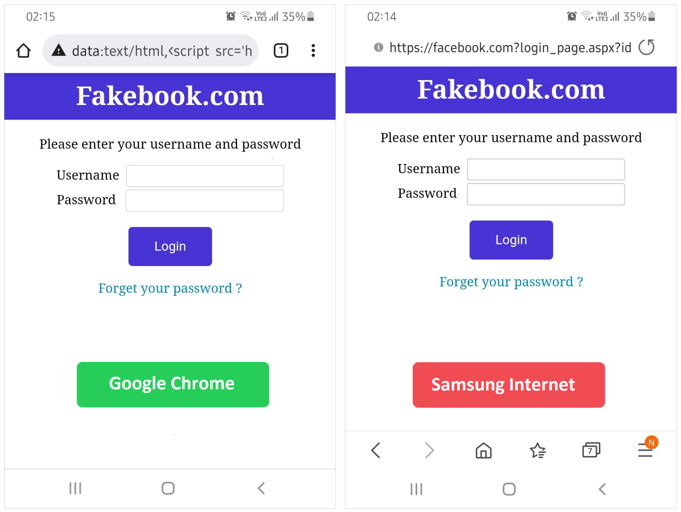
The Opera issue can be tracked via CVE-2020-6159, while the Chromium issue in the patching process. While certain practices from Table I could be used for mitigation (e.g., "Search query" or "Null origin"), Chromium intends to repair the issue while maintaining support for typing JavaScript URIs. This is made difficult because IME keyboards do not provide APIs for listening to clipboard events. Thus, we propose a solution that meets the above criteria in Section IV.

B. Impersonation of website origins using the data scheme

Data URIs allow to render data included in the URI string in the web browser. As a consequence, data URIs have been used in the past to display "fake" origins in browsers as described in Section II. We thus opted to study this problem in the context of our set of mobile browsers. As evident from Table I, only FireFox and FireFox focus lack support for data URIs.

While all browsers restrict top-level navigation for data URIs, we noticed an issue in the behaviour of the address bar in the Samsung browser: it displays the last characters of data URI, which truncates the data scheme and most of the actual data from lengthy URIs. As a consequence, we found that it is possible to craft a data URI that renders a web page that seems to be hosted on a legitimate origin, e.g., a fake login phishing page. As can be seen in Figure 4 (right), such a custom data URI renders a form that appears to be hosted on a legitimate domain (`fakebook.com` in our example) and served via HTTPS. We tested the crafted data URI on two different Samsung devices with different display resolutions, Samsung A75 and Samsung S20 Ultra, and confirmed that the URI produces the same result regardless of the screen size. In contrast, all other browsers in our set correctly displayed the beginning of the URI, including the data scheme, as can be seen for Chrome in Figure 4 (left).

Issue #2: Displaying only the end of a Data URI in the Samsung browser address bar allows an adversary to fake the origin of the rendered data.


02:15

data:text/html,<script src='h

Fakebook.com

Please enter your username and password

Username

Password

Login

Forget your password ?

Google Chrome

02:14

https://facebook.com?login_page.aspx?id

Fakebook.com

Please enter your username and password

Username

Password

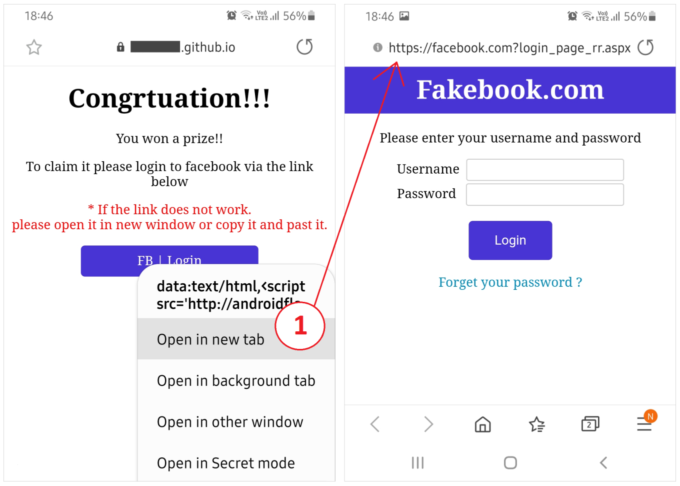
Login

Forget your password ?

Samsung Internet

Figure 4. Loading a crafted data URI in Google Chrome (left) and Samsung browser (right). The data scheme is visible on Google Chrome. In contrast, the Samsung browser displays the last characters of the URI, leading the user to think that the content is hosted on a legitimate origin and served via HTTPS.

*a) **Threat model**:* Two threat models can be considered for this issue. First, since chromium browsers support opening a data URI via new tab option, an adversary can set up a phishing data URI link asking the user to open it with new tab if it is not working resulting of origin spoofing as illustrated in figure 5. Alternatively, the same threat model as for the JavaScript scheme (Section III-A) can be considered. Namely that a victim is ticked into copying a data URI and pasting it into their browser. Our considered browsers do not apply any trimming for data URI, so in this case pasting via the context menu or an IME keyboard leads to identical results. Similar to the JavaScript scheme, the adversary can prepend the URI of a legitimate origin to the crafted data URI to display the target website's logo, as seen in Figure 2.


18:46

[redacted].github.io

Congrtuation!!!

You won a prize!!

To claim it please login to facebook via the link below

* If the link does not work.
please open it in new window or copy it and past it.

FB | Login

data:text/html,<script src='http://androidf

Open in new tab

Open in background tab

Open in other window

Open in Secret mode

18:46

https://facebook.com?login_page_rr.aspx

Fakebook.com

Please enter your username and password

Username

Password

Login

Forget your password ?

Figure 5. An example of opening a data URI via new tab option (1). Samsung browser displays only the final characters leading to origin spoofing.

Conventional phishing defenses, such as showing the origin in the address bar or black-listing known phishing sites [26] are ineffective against this issue because the rendered data is embedded in the URI, rather than being fetched from a remote server.

We reported the issue to Samsung and it can be tracked via CVE-2021-25419. We discuss mitigations in Section IV.

C. Permission system bypass on Samsung Android OS through the file scheme

The file scheme in mobile browsers gives access to the content of the internal storage, thus browser apps should require the Android storage permission. Similar to the previous case studies, we evaluated the implementation of the file scheme in the selected browsers, cf. Table I. Most browsers support the scheme, with the notable exception of Chrome, and give access to mobile internal storage. Many popular browsers, including Samsung browser, Opera, Brave, Edge, Vivaldi, and FireFox (Focus) also display an HTML index page that lists internal storage' files on the device. Accessing local content through the file scheme from a local or an external HTML file is not permitted in all supported browsers because of SOP.

Starting from a fresh browser install with no permissions, we tested the handling of file URIs and the respective permission requests in all our browsers. This revealed an unexpected behaviour in the Samsung browser that was not present in the other browsers:

Navigating to `file:///sdcard` results in a prompt to grant or deny storage permissions. However, if the user selects "deny and don't ask again", *i.e.*, permanently declines storage permission, and then repeats the request to the same URI, the internal file list appears and access to the internal storage is given.

Potential issue #3: The Samsung browser can access the internal storage of the device without storage permission. The app apparently uses a proprietary non-standard mechanism that bypasses the Android permission system.

*a) **Handling of file URIs in the Samsung browser**:* To understand this surprising behaviour and apparent bypass of the Android permission system, we reverse-engineered the handling of file URIs in the Samsung browser. We found that a request to access the file scheme is only *denied* if the following three conditions are all satisfied:

- The app can request storage permissions.
- The app does not have storage permissions.
- The request does not access the Samsung browser's private files located at `/data/data/com.sec.android.app.sbrowser/sbrowser`.

At first glance, this logic (cf. Table II) appears to prohibit requests for local file access without storage permission. The first and second conditions seem identical, *i.e.*, the first condition seems to imply the second and vice versa. More specifically, if storage permission is not granted, it is possible to request it (case I), and once storage permission is granted, it is no longer possible to request it (case II).

However, the relatively new "deny and don't ask again" option in the Android permission system allows to circumvent these rules: it does not grant the app storage permission nor does it allow subsequent requests. Thus, when Samsung browsers does not have storage permission and can no longer request

Conditions	Case I	Case II	Case III (Exploit)
App can request storage permissions	T	F	F
App does not have storage permissions	T	F	T
Request does not access Samsung browser storage	T	T	T
Access to internal storage	Denied	Granted	Granted

Table II. DIFFERENT CASES OF FILE URI RESTRICTIONS IN SAMSUNG BROWSER. CASE I: USER DENIES STORAGE PERMISSION; CASE II: USER GRANTS STORAGE PERMISSION; CASE III: USER DENIES STORAGE PERMISSION AND DISABLES FURTHER PROMPTS.

storage permission, an access to the internal storage will bypass the check and proceed with the access (case III).

*b) **Root cause analysis**:* Still, it remains unclear how the Samsung browser is—even though it theoretically allows the access—is able to gain access to the internal storage without holding the actual permission. Finding the root cause proved to be challenging because many Android components are involved. It was however clear that the issue is caused by the interplay of Samsung-specific modifications, as the problem did not occur with other browsers (and because the Samsung browser only runs on Samsung devices). We therefore examined the involved components step-by-step.

*c) **System-level analysis**:* At the Android system level, we identified two possible mechanisms that could cause the observed permission system bypass:

Privileged permissions: Android can give read/write access to specific apps via so-called allowlists [32].

Signature-based permissions: Apps signed with the vendor key are granted system signature permissions.

We analysed Samsung's allowlist and did not find any rule granting Samsung browser storage permission. This eliminates the first option. Further, we found that the Samsung browser is not a system app and installed on the data partition. We repacked and self-signed the app with a debug key. This invalidates any signature permission that might be granted by Android to the app. We found that the issue is still present in the self-signed browser, thus excluding the possibility of signature-based permissions as root cause.

*d) **App-level analysis**:* As we excluded a system-level root cause, we began to consider possibilities that certain characteristics in the app cause the behaviour. First, we determined whether the issue is related to Chromium: We patched other Chromium browsers and eliminated the storage permission request. We however found that (as expected) access is denied.

We thus a statically analysed the Samsung browser app to find any components that access the internal storage, both within the app and through external components, e.g., third-party apps, services or SDKs. Because the code base of the Samsung browser is large (278 MB after decompression), we concentrated on the file scheme handling and any involved app components (activities, services, content providers and broadcast receivers). From this analysis, we found that file URIs are handled as follows:

1) The requested URI is retrieved from the address bar.
2) It is sanitised against the allowed schemes.
3) The URI is passed to Terrace browser engine to further process and issue the request.

*e) **Inside the Terrace browser engine**:* Terrace is the browser engine used by Samsung. It is a native C library (compiled for ARM) with a large code base of 70 MB for 64-bit and 50 MB for 32-bit processors. We found that Terrace is based on Blink, which is a rendering engine part of the Chromium project [33]. Unfortunately, the size of the Terrace code base and the lack of debug symbols makes a full static analysis infeasible. Nevertheless, we noticed several interesting strings related to the file scheme. At this stage, we decided to switch to dynamic analysis. We used Frida [34] to intercept and override potentially interesting methods within the browser engine. Frida's interception abilities allowed us to inspect parameters and return values of relevant functions calls. Figure 6 illustrates our dynamic analysis setup using Frida. We inject a Frida gadget into the Samsung browser and instruct it to start it when the app starts (1). The browser is self-signed with our own debug key. We used `adb` to connect the smartphone to a PC. From the PC, we use Frida module to communicate with Frida gadget in the app over `adb` (2). The Frida gadget starts in listen mode, ready to accept scripts to inspect components in the target application (3).

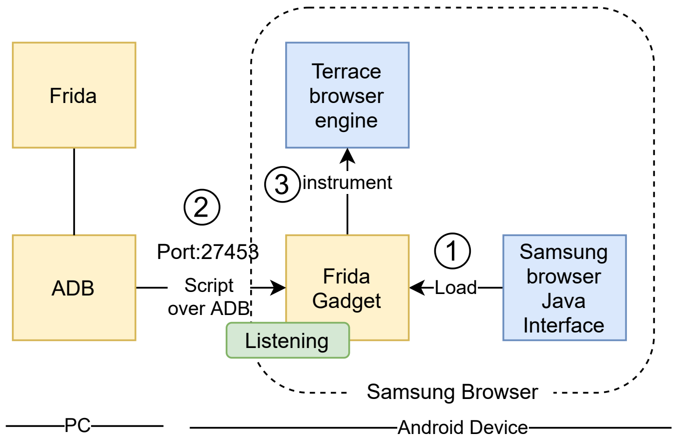


Figure 6. Our dynamic analysis setup for Samsung browser to inspect the Terrace browser engine. Original browser components are blue, and our test components are yellow. The analysis starts by loading a Frida gadget to open a communication port (1). The Frida module communicates with the gadget through adb to transmit inspection scripts (2). Finally, the Frida gadget uses the supplied scripts to perform the analysis (3).

The setup does not require rooting the device, as we noticed that the Samsung browser relies on Samsung Knox. Rooting the device would remove the Knox modules, possibly causing misbehaviour or crashes of the browser [35].

We identified and intercepted the Java method responsible for loading URIs method and witnessed how file URIs are forwarded to the Terrace engine. To overcome Terrace's lack of debug symbols, rather than inspecting potentially relevant Terrace functions, we hooked native file access operations like `read()` and `open()` syscalls. When accessing the internal storage using the file scheme, we indeed observed Terrace invoking `open()` and `opendir()` to open files and directories, respectively. At this stage, we confirmed that the Samsung browser, as a single process, has access to the internal storage and is not reliant on app components or external applications.

*f) **Finding the root cause**:* Having determined that syscalls to access the internal storage always succeed in the Samsung browser (independent of Android permissions), we injected a compiled module into the app to print its effective permission. From this, we found that the Samsung browser is in the group `sdcard_rw`, finally explaining why access to the internal storage is granted. However, it remained unclear *how* the app obtains membership of the `sdcard_rw` group.

To understand that, we repacked the app numerous times, each time removing one component and checking if group membership was removed. We started with removing native C libraries, then app components, and lastly classes and libraries in the app's source code. We found that neither native C libraries, including the Terrace engine, nor app components affect access to internal storage.

However, we identified a specific metadata entry called "SDP" in the Android manifest of the application that grants the `sdcard_rw` permission to the Samsung browser.

```
<meta-data android:name="sdp"
 android:value="enabled"/>
```

We found that Secure Data Protection (SDP) is a Samsung Knox SDK component that can provide protection of sensitive data within an app [36, 37].

*g) **Exploiting SDP to bypass storage permissions**:* It appears that any app can request the inclusion of SDP and its SDK through a simple metadata setting, both are used in the Samsung browser. To verify that, as a proof-of-concept we cloned the code base of the Samsung browser app, but changed its package name and self-signed the binary. This process is complex for an app with a large code base like Samsung browser, because it requires changes in several places (classes, directories and metadata) and because the authority names of app components must be altered as well. Taking this into account, we successfully created an independent app with a distinct name that uses SDP and has access to the internal storage. The app does not interfere with and can be installed alongside the original Samsung browser. Because it has a distinct package name, the cloned app can also be published to Google Play.

Alternatively, one can also include the SDP configuration in any other app. Thus, an adversary who can trick the user into installing a malicious app (e.g., disguised as a game or similar) can exploit this issue to exfiltrate the internal storage even though the user did not grant the Android storage permission. We reported these issues to Samsung on October 30, 2020 and February 9, 2021. They assigned two CVEs as follow: CVE-2021-25348 for bypassing the security check in the browser, and CVE-2021-25417 for the underlying privilege escalation issue in Samsung's Android variant. At the time of reporting, both issues affected Android 10. Samsung has patched the issues on December 1, 2020 and June 8, 2021, respectively, with the Samsung browser version v13.0.1.64.

Issue #3: Regardless of Android permission, the Samsung Knox SDP gives any app access to the internal storage without user approval.

IV. Discussion and Mitigations

Our case studies show that, even though the general topic has been extensively studied, handling of local URI schemes in mobile browsers still suffers from substantial oversights. Thus, we conclude that currently used test methods are insufficient to detect complex issues, especially those that are caused in interaction with external components, e.g., the Knox SDK or IME keyboards. Similarly, more principal mitigations appear to be required for the issues pointed out in this paper, especially those related to JavaScript and data URIs.

*a) **Mitigating self-XSS via the JavaScript scheme**:* As shown in Section III-A, hooking paste events is not sufficient to capture all sources of input into the address bar (other than the user typing): IME keyboards are third-party components and do not expose APIs for event listening (e.g., paste events), nor does Android provide a generic interface for this. The issue cannot be resolved through changes to the IME keyboard code, as the Android guidelines state that IME keyboards are not responsible for sanitising their output [38].

Apart from that, new input methods might be introduced in the future that could also bypass the existing address bar sanitisation logic. Google thus sought a solution that fundamentally prevents future similar issues but still allows manual typing of URIs with local schemes (like JavaScript).

We thus propose a generic multi-character handler for Android that solves the issue. Our solution is based on the fact that user-typed text appears character-by-character, whereas pasted text (be it from normal or IME keyboard clipboard) is inserted as a block. Android keyboard uses the `commitText()` method to send text to a designated input field. If the sent text is a single character, it is sent as it. Otherwise, the keyboard sends a special key event along with the text in a block. Therefore, to intercept paste events, we can override the address bar's `onTextChange()` method and inspect the number of inserted characters: if a multi-character insertion is detected, this indicates a paste event (from any source) that can be blocked.

We implemented this approach as a proof-of-concept and confirmed that it was able to intercept paste events from both the context menu and the IME keyboards. Alternately, instead of overriding the address bar's `onTextChange()` method. It its possible to attach a `TextWatcher` to the address bar and override its `onTextChange()` method to achieve the same results. Google's security team considered and our proposed solution. They adopted and deployed a fix using `TextWatcher` [39].

*b) **Standard approach to avoid phishing with data URIs**:* While most browsers adopt the correct approach to display the beginning of an URI, rather than its end, we found that Samsung browsers did not follow this behaviour: the prefix data URI scheme for long URIs is hidden as shown in Figure 4, enabling an adversary to create phishing URIs that appear to be hosted on legitimate origins. To resolve this issue for data URIs (and also other schemes), we propose that the community defines a standard approach to correctly and securely display URIs in browser. For example, such a standard could mandate to always show the start of the URI as implemented in most browsers (and also how Samsung patched the issue after our report).

*c) **Preventing permission system bypass on Samsung Android**:* The issue presented in Section III-C cannot be fully prevented at the browser level, as the underlying reason is rooted in Samsung's modifications to Android. Because SDP does not authenticate apps that utilise it, any app may publicly subscribe to it and obtain access to internal storage. Therefore, an OS update or an update to the Knox SDK are required to patch the vulnerability. While Samsung informed us that the issue has been resolved for Android 10, we did not receive information on their mitigation strategy. However, recently, we noticed that SDP is deprecated in the latest Knox SDK patch v3.7.

*d) **Limitations**:* Our work is based on manual inspection of the most common local schemes in the most popular mobile browsers. Automating (parts of) our analysis and extending it to other browsers and schemes is an interesting research area that we leave for future work. We note that such automation is challenging: the found issues are related to complex interactions between UI components (e.g., text fields) and data entry methods (e.g., IME keyboards) or OS-specific configuration options.

V. Conclusions

In this paper, we demonstrated several security issues in local URI schemes, affecting major mobile browsers including Google Chrome, Edge, Opera, and the Samsung browser. We show that a lack of proper sanitisation of JavaScript URIs can lead to self-XSS attacks, while data URIs can be abused for spoofing origins in phishing attacks. Finally, an issue in file URIs led us to discover a much deeper design flaw in Samsung's Android, giving an arbitrary app access to the internal storage without user consent and bypassing the dedicated Android storage permission. Our results highlight that, even though the overall attack surface is well-understood, testing methods and tools to (semi)automatically detect URI handling issues in mobile browsers are still lacking and motivate future work in this direction.

Acknowledgments

We thank Ali Darwish, ElMuthana Mohamed and Hamad Salmeen for participating in browsers testing.

This research was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grant EP/R012598/1. Abdulla Aldoseri is supported by a stipend from the University of Bahrain.

References

[1] T. Berners-Lee et al., "IETF RFC 1738 - Uniform Resource Locators (URL)," https://www.w3.org/Addressing/rfc1738.txt, Jan 2021, (Accessed on 01/27/2021).

[2] C. Kerschbaumer, "Blocking Top-Level Navigations to data URLs for Firefox 59 - Mozilla Security Blog," https://blog.mozilla.org/security/2017/11/27/blocking-top-level-navigations-data-urls-firefox-59/, Nov 2017, (Accessed on 10/17/2021).

[3] Chromium, "684011 - remove: Top frame navigations to data urls - chromium," https://bugs.chromium.org/p/chromium/issues/detail?id=684011, Jan 2017, (Accessed on 10/17/2021).

[4] KirstenS, "Cross Site Scripting (XSS) Software Attack — OWASP Foundation," https://owasp.org/www-community/attacks/xss/, (Accessed on 12/03/2021).

[5] Facebook, "What is a self-XSS scam on Facebook? — Facebook Help Centre," https://www.facebook.com/help/246962205475854, May 2021, (Accessed on 10/05/2021).

[6] Y. Cao, C. Yang, V. Rastogi, Y. Chen, and G. Gu, "Abusing browser address bar for fun and profit—an empirical investigation of add-on cross site scripting attacks," in *International Conference on Security and Privacy in Communication Networks*. Springer, 2014, pp. 582–601.

[7] C. Joe-Uzuegbu, U. Iwuchukwu, and L. Ezema, "Application virtualization techniques for malware forensics in social engineering," in *2015 International Conference on Cyberspace (CYBER-Abuja)*. IEEE, 2015, pp. 45–56.

[8] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986, Jan. 2005. [Online]. Available: https://rfc-editor.org/rfc/rfc3986.txt

[9] Mozilla, "Access-Control-Allow-Origin — HTTP — MDN," https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin, Jan 2021, (Accessed on 01/26/2021).

[10] A. Barth, "RFC 6454 — The Web Origin Concept," https://tools.ietf.org/html/rfc6454, December 2011, (Accessed on 01/26/2021).

[11] B. Hoehrmann, "The 'javascript' resource identifier scheme," Internet Engineering Task Force, Tech. Rep., Sep. 2010, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-hoehrmann-javascript-scheme-03

[12] Mozilla, "Same-origin policy — Web security — MDN," https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, Jan 2021, (Accessed on 01/26/2021).

[13] L. M. Masinter, "The "data" URL scheme," RFC 2397, Aug. 1998. [Online]. Available: https://rfc-editor.org/rfc/rfc2397.txt

[14] Mozilla, "Types of attacks — web security — mdn," https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks, May 2021, (Accessed on 10/05/2021).

[15] S. Gupta and B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 1, pp. 512–530, 2017.

[16] OWASP, "A7:2017-cross-site scripting (xss) — owasp," https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS), (Accessed on 12/01/2021).

[17] M. D. Ambedkar, N. S. Ambedkar, and R. S. Raw, "A comprehensive inspection of cross site scripting attack," in *2016 international conference on computing, communication and automation (ICCCA)*. IEEE, 2016, pp. 497–502.

[18] A. Terada, "Attacking Android browsers via intent scheme URLs," https://www.mbsd.jp/Whitepaper/IntentScheme.pdf, Mar 2014, (Accessed on 10/16/2021).

[19] Mozilla, "Data URLs — HTTP — MDN," https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs, May 2021, (Accessed on 10/05/2021).

[20] Chromium, "1040755 - Security: Another "univer-

sal" XSS via copy&paste," https://bugs.chromium.org/p/chromium/issues/detail?id=1040755, Jan 2020, (Accessed on 10/17/2021).

[21] CWE, "CWE-451: User Interface (UI) Misrepresentation of Critical Information (4.5)," https://cwe.mitre.org/data/definitions/451.html, Oct 2021, (Accessed on 10/17/2021).

[22] M. Nishimura, "Addressbar spoofing through stored data URL shortcuts on Firefox for Android — Mozilla," https://www.mozilla.org/en-US/security/advisories/mfsa2016-05/, Jan 2015, (Accessed on 10/05/2021).

[23] R. Baloch, "Multiple Address Bar Spoofing Vulnerabilities In Mobile Browsers—Miscellaneous Ramblings of An Ethical Hacker," https://www.rafaybaloch.com/2020/10/multiple-address-bar-spoofing-vulnerabilities.html, May 2021, (Accessed on 10/05/2021).

[24] Apple Support, "About the security content of Safari 14.0," https://support.apple.com/en-bh/HT211845, Sep 2020, (Accessed on 10/17/2021).

[25] D. Wu and R. K. Chang, "Analyzing android browser apps for file:// vulnerabilities," in *International Conference on Information Security*. Springer, 2014, pp. 345–363.

[26] A. Barth, C. Jackson, C. Reis *et al.*, "The security architecture of the Chromium browser," in *Technical report*. Stanford University, 2008.

[27] Google, "Remote debug android devices," https://developer.chrome.com/docs/devtools/remote-debugging/, April 2015, (Accessed on 10/14/2021).

[28] Android, "Copy and Paste — Android Developers," https://developer.android.com/guide/topics/text/copy-paste, Sep 2021, (Accessed on 09/29/2021).

[29] Chromium, "AutocompleteEditText.java — Chromium Code Search," https://source.chromium.org/chromium/chromium/src/+/main:chrome/browser/ui/android/omnibox/java/src/org/chromium/chrome/browser/omnibox/AutocompleteEditText.java;l=189, (Accessed on 12/01/2021).

[30] Android, "Image keyboard support — Android Developers," https://developer.android.com/guide/topics/text/image-keyboard, Sep 2021, (Accessed on 09/29/2021).

[31] Google, "Gboard — the Google Keyboard," https://play.google.com/store/apps/details?id=com.google.android.inputmethod.latin&hl=en&gl=US, Oct 2021, (Accessed on 10/06/2021).

[32] Android, "Privileged Permission Allowlisting — Android Open Source Project," https://source.android.com/devices/tech/config/perms-allowlist, Aug 2021, (Accessed on 10/08/2021).

[33] Chromium, "Chromium Blog: Blink: A rendering engine for the Chromium project," https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html, Sep 2021, (Accessed on 10/09/2021).

[34] Frida, "Frida — A world-class dynamic instrumentation framework," https://frida.re/docs/examples/android/, Sep 2021, (Accessed on 10/09/2021).

[35] Samsung, "Root of Trust — Knox Platform for Enterprise White Paper," https://docs.samsungknox.com/admin/whitepaper/kpe/hardware-backed-root-of-trust.htm, Sep 2021, (Accessed on 10/09/2021).

[36] ——, "Sensitive Data Protection (SDP)," https://docs.samsungknox.com/dev/knox-sdk/sensitive-data-protection.htm, Sep 2021, (Accessed on 10/09/2021).

[37] ——, "Sensitive Data Protection — Knox Platform for Enterprise White Paper," https://docs.samsungknox.com/admin/whitepaper/kpe/sensitive-data-protection.htm, Sep 2021, (Accessed on 10/09/2021).

[38] Android, "Create an input method — Android Developers," https://developer.android.com/guide/topics/text/creating-input-method, (Accessed on 09/29/2021).

[39] Chromium, "81e2250b60d4a6d863864249cc2f1da97669f03d - chromium/src - git at google," https://chromium.googlesource.com/chromium/src/+/81e2250b60d4a6d863864249cc2f1da97669f03d, March 2022, (Accessed on 03/23/2022).