Reading Between the Lines: An Extensive Evaluation of the Security and Privacy Implications of EPUB Reading Systems

Gertjan Franken
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

Tom Van Goethem
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

Wouter Joosen
imec-DistriNet, KU Leuven
3001 Leuven, Belgium

***Abstract*—In recent years, e-books have proven to be a very appealing alternative to physical books; nowadays, almost every written book is published in an electronic format next to its physical copy. In an attempt to promote consensus and to offer an alternative to emerging proprietary e-book formats, the Open eBook format was introduced, now known as the EPUB format. Building on existing web functionalities, this open format relies primarily on XHTML and CSS to construct e-books. As such, browser engines are often employed to render the contents of EPUBs. However, this implies that reading systems may face similar vulnerabilities as web browsers.**

In this paper, we report on a semi-automated evaluation of the security and privacy aspects of EPUB reading systems. This evaluation, which was performed on 97 EPUB reading systems covering seven platforms and five physical reading devices, revealed that almost none of the JavaScript-supporting reading systems sufficiently adhere to the EPUB specification's security recommendations. Furthermore, our results indicate that 16 reading systems even allow an EPUB to leak information about the user's file system, and in eight cases extract file contents. In addition to the semi-automated evaluation, we demonstrate that an attacker can launch even more potent attacks that may lead to a full compromise of a user's system, by exploiting aspects specific to the implementation of reading systems used by millions of users. Finally, we investigate the root cause of the identified security and privacy issues, uncovering several flaws in both the implementation of EPUB reading system, as well as shortcomings of the EPUB specification.

I. Introduction

In the last decade, digital books have gained significantly in popularity, and experts argue they are here to stay [52], [62]. Today, almost every newly published book or magazine is made available in a digital format, in addition to their physical copy. EPUB e-book creation is considered fairly straightforward, which combined with the possibility to publish without any vendor interposition, explains in particular its popularity among self-publishing authors and within open license communities such as Project Gutenberg[1]. EPUBs primarily consist of XHTML documents and CSS stylesheets, bearing close resemblance to web pages. Consequently, browser engines are often employed by reading systems to render EPUB content. Next to static content, the EPUB standard also allows for media such as audio or video, and dynamic content leveraging JavaScript.

Because EPUB reading systems are closely related to web browsers, they are prone to similar security and privacy issues. A malicious EPUB could leverage the reading system's capabilities to mount attacks against the user, much like the dynamics between a malicious website and a browser. For instance, since the introduction of EPUB3, several blogs and articles have voiced concern about the capabilities associated with supporting JavaScript in EPUB reading systems [18], [19], [30], [35], [48]. The EPUB specification acknowledges these concerns by dedicating one of its sections to security recommendations for EPUB reading system developers [2]. However, these recommendations are very lenient and lack strict enforcement.

The EPUB specification defines a large number of (optional) capabilities, several of which are questionable from a security perspective, such as access to the local filesystem, especially when considered in combination with access to remote end-points. Consequently, since most EPUB reading systems are not well-documented, this makes it difficult for the user to assess what privileges the reading system provides to an opened EPUB. Moreover, there is no indication whether a reading system is compliant to the EPUB specification, nor are we aware of any study evaluating their capabilities and compliance.

In this paper, we present the first extensive evaluation of EPUB reading systems, addressing their capabilities, compliance to the EPUB specification, and security and privacy implications. To this end, we crafted an extensive testbed that covers a wide range of the threat surface, consisting of EPUBs which are loaded in EPUB reading systems to perform a semi-automated evaluation. As soon as such an EPUB is opened by a reading system, the embedded experiments are executed, after which the resulting data is rendered or sent to a server. This way, we evaluated 97 of the most popular EPUB reading systems, covering seven different platforms and five physical e-reader devices. Our evaluation uncovered that almost all reading systems that execute embedded JavaScript do not fully respect the specification's security recommendations, of which 16 can be abused by malicious EPUBs to leak information about the local filesystem. We reached out to all vendors in order to report the identified issues.

Moreover, to complement our semi-automated evaluation, we manually inspected three widely used EPUB reading systems for implementational flaws, revealing several severe security vulnerabilities. We discovered a universal XSS affecting two browser extensions (≈300,000 browser installations), and the ability to leak documents residing in the user's library as soon

[1] https://www.gutenberg.org/

as a malicious e-book is opened on a Kindle (the most widely used physical e-reader [27]). These findings indicate that the results of our semi-automated evaluation should be considered as a lower bound, and that in fact EPUB reading systems may face even more security issues that are implementation-specific. We argue that by limiting the capabilities of the reader application and the employed rendering engine to a minimum, the threat surface can be significantly reduced. For instance, iOS reading systems can only access files within the application by design, and thus none could be abused to leak sensitive files. Furthermore, we explored the presence of real-world abuse by analyzing more than 9,000 EPUBs obtained from five online e-book stores and two file-sharing platforms. We did not find any evidence of ongoing abuse, allowing EPUB reading system developers to adopt adequate security measures before users are actively being exploited. Finally, we show that four out of six evaluated self-publishing EPUB services do not adequately vet submitted manuscripts, which could lead to the distribution of malicious EPUBs through legitimate channels.

We make the following contributions:

- We developed a testbed of numerous EPUBs to assess the security and privacy impact of various aspects of EPUB reading systems and the rendering engine they employ.
- By applying this testbed in a semi-automated analysis, we evaluated a total of 97 EPUB reading systems, of which 92 were freely available as applications on desktop (Windows, macOS and Ubuntu) and mobile (iOS and Android), or as a browser extension (Chrome and Firefox), and of which five were physical e-reader devices.
- The result of this analysis shows that many reading applications can be abused, either by leaking file contents, or by violating the user's privacy expectations.
- To explore both ongoing and potential abuse in the EPUB ecosystem, we downloaded over 9,000 EPUBs from two torrent sites and five online e-book stores, and assessed the vetting process of six popular self-publishing services.
- Lastly, based on the identified issues and their root cause, we propose to make the EPUB specification more strict. Moreover, to encourage consumers and developers to measure the security and privacy impact of their reading systems, we have released the EPUBs used in our evaluation along with the source code to craft them.

II. Background

In May 2019, W3C issued EPUB 3.2, the most recent version of the EPUB standard at the time of writing [1]. In the remainder of this paper, we refer to this particular version when discussing the EPUB standard, unless specified otherwise.

A. EPUB Technical Standard

The EPUB standard consists of five sub-specifications, each defining complementary core features and functionality. For reasons of brevity, we will not describe each sub-specification, instead we will discuss the standard as a whole.

The EPUB format and the internal structure of a compliant EPUB file, or a so-called EPUB Container, are visualized in Figure 1. An EPUB is a single-file container with the `.epub` extension, of which the included content is compressed in a ZIP archive. The `mimetype` file indicates the EPUB Open Container format (OCF) media-type, with `application/epub+zip` as the two-part identifier [17].

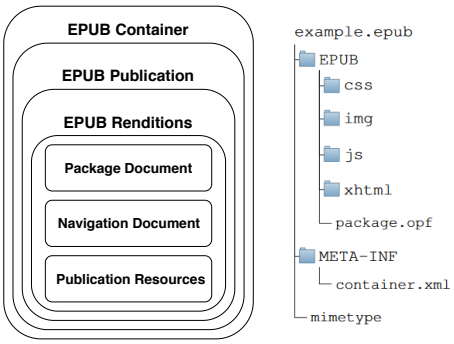

Fig. 1. On the left a visual representation of the EPUB format, and on the right the internal file structure of a compliant EPUB archive.

An EPUB Publication, included in the Container, must consist of at least one Rendition, the specific rendering of the EPUB's content. A Rendition is represented by an EPUB Package, which consists of the Package Document (`package.opf`), the Navigation Document and all Publication Resources used to build the Rendition. The Package Document conveys various types of information such as the title and authors of the EPUB Publication. Moreover, it also defines the sequence in which the Content Documents are rendered, called the spine.

Publication Resources characterize the actual content and layout of the EPUB Rendition. The standard makes a distinction between Core Media Type Resources and Foreign Resources. The former are resources of media types that are deemed supported by all Reading Systems. This set of media types consists of Content Documents (XHTML or SVG media type files), CSS stylesheets, and various image and audio formats. The latter are resources of which support is not mandatory and therefore require fallback mechanisms to Core Media Type Resources, in case the Reading System does not offer support.

The Package Document is also used to assign special properties to particular Content Documents. An interesting example is the `scripted` property, which indicates that the referred Content Document contains executable JavaScript. EPUB Containers are allowed to contain such Scripted Content Documents, however, EPUB Reading Systems are not obligated to support script execution. As such, the EPUB standard states that Scripted Content Documents should retain their integrity when scripting support is disabled, without any loss of information or legibility.

B. EPUB Reading Systems

EPUB Reading Systems are applications that interpret and render EPUB files according to the EPUB specification. They come pre-installed on physical e-book reading devices (e.g. e-ink readers), but are also available on smartphones,

tablets, desktop computers, and even in the form of browser extensions. Nearly all applications are free (some with in-app advertisements) and often provide support for various other e-book formats.

The EPUB specification dictates the minimal requirements that should be met by an EPUB Reading System. These requirements are mostly related to rendering and presenting the content that is the EPUB Publication. Only a few paragraphs of the standard are dedicated to security considerations, with special attention to providing support for scripting [2]. Here, Reading System developers are provided with several attack vectors that should be considered, and with recommendations on how to deal with security-related issues concerning scripted content execution.

One of the recommendations states that the Reading System should behave as if a unique domain were allocated to each Content Document, consequently isolating documents from each other. This isolation is enforced by the Same-Origin Policy, which dictates that one origin cannot access resources from another. Reading Systems that allow scripting and network access should also notify the user whether any network activity is occurring, and ideally provide functionality for the user to disable it.

According to its specification, EPUB Reading Systems may allow an EPUB to store persistent data (e.g. LocalStorage). This data is recommended to be considered sensitive, and therefore this data should not be accessible by other documents.

C. Same-Origin Policy

Modern browsers employ a wide range of policies to protect users against malicious websites, among which the highly essential Same-Origin Policy (SOP) [46]. This policy in particular is used to isolate documents and scripts located on different origins, to prevent one website to perform undesirable actions on, or steal sensitive user information from another website. Two URLs share the same origin if their scheme, host and port are identical. For example, when the origin `https://attacker.com` tries to extract information from `https://bank.com`, this would be prevented by the SOP. Not only documents and scripts are protected by the SOP, but also any information stored by the LocalStorage API [33]. Although cookies are not subjected to the SOP, they can only be accessed by associated domains. As an extension to the SOP, websites cannot instruct the browser to render or access files located on the user's file system. For example, when a website leverages an iframe to render a file by referring to `file:///etc/passwd` or employs the XMLHttpRequest or Fetch API to access its content, the browser will refuse.

III. Motivation

While e-books have grown to be a multi-billion dollar industry [52], [62] and countless EPUB reading systems are available on essentially every platform, the reading system ecosystem has never been subjected to a comprehensive security or privacy assessment. In the following subsections, we argue why such an assessment is imperative.

A. Intransparency

Most EPUB reading systems rely on browser engines to render e-book content. Over the last few years, there has been an extensive growth in the number of features that these browser engines support, significantly increasing their threat surface [57]. Scripting in e-books, which was already suggested as an optional functionality in the 1999 OEBPS specification, could be used to launch a variety of attacks to circumvent the same-origin policy, or even to attack the underlying operating system. As such, opening an e-book could introduce the user to a plethora of attacks, which have not been extensively explored to this date. In face of these threats, the more recent editions of the EPUB standard now include a section on security considerations for EPUB reading system developers.

However, we argue that these considerations lack binding requirements and are insufficiently concrete. In that regard, even if an EPUB reading system is compliant with the official specification, users do not have any guarantee that their security and privacy will be safeguarded. For instance, a compliant reading system might allow an EPUB to freely access the user's file system and send a copy of it to a remote server. Moreover, countless curated lists only recommend reading systems based on usability features and supported e-book formats, making it nearly impossible for users to verify whether an application is sufficiently secure.

We argue that the more features are being added to the specification, the more it will cripple the transparency of security and privacy factors in EPUB reading systems as long as no clear compulsory considerations are included. However, even when these are included, still, there is no straightforward way to verify compliance of a reading system. This uncertainty is one of the reasons why we deem a comprehensive evaluation of EPUB reading systems imperative. We aim to improve this much-needed transparency by evaluating the most popular EPUB reading systems, leveraging a semi-automated analysis. In the following subsections, we discuss two attacker models which aim to abuse EPUB reading system capabilities, impacting the user's security and privacy.

B. Malicious EPUBs

Nowadays, tens of thousands of EPUBs are made available online for free, either legally or illegally. Whereas EPUB submissions to the Gutenberg Project are subjected to examination by volunteers [53], various other channels omit third-party validation and share the EPUB as-is (e.g. torrent sites, social media). A study of the UK government's Intellectual Property Office finds that approximately 17% of e-books are illegally consumed online, accounting for around four million e-books [34]. As such, users may face various threats when accessing an e-book obtained either from a publisher who does not sufficiently sanitize or verify the published books, from a malicious website directly, or from a file-sharing platform.

The attacker could configure the EPUB such that upon opening, a malicious JavaScript payload is executed. Depending on the capabilities and vulnerabilities of the reading system, the attacker could try to either extract sensitive system files, such

as the browser's cookie store, and then send the contents of these files to an online web server. Furthermore, if the browser engine used by the reading system is outdated, it might contain publicly known vulnerabilities that can then be exploited by the malicious e-book in order to compromise the system.

In an explorative experiment using Chrome and Firefox, we assessed whether a website could automatically cause a malicious EPUB to be loaded in an installed reading system. In both browsers (on desktop and mobile), this requires at least one user interaction. Although a website can instruct the browser to download an EPUB (e.g. clicking a URL through JavaScript), still one user click is required to actually open it. However, an EPUB reading system installed as a browser extension could intercept the download and automatically render the EPUB.

C. Tracking EPUBs

E-books in a proprietary format are usually distributed through the associated vendor's online bookstore, which is often embedded within the vendor's own reading system (e.g. physical e-reader devices or applications). Leveraging their own proprietary formats and reading systems, vendors are known to harvest user data based on interactions with their reading system [5], [31], [36].

Although EPUB is an open format and not affiliated to any specific vendor, distributors of EPUBs might still be able to track users. To supplement their recommendation system, the distributor might try to figure what other books make up the user's library. This can be accomplished if the user's reading system allows EPUBs to render local files located within the directory where the unpacked EPUBs are stored. Then, to scan the contents of the library, each distributed EPUB could include a list of popular EPUBs, and code to test their presence on the system. Even when the targeted EPUB reading system does not allow rendering local files, timing attacks can prove as a suitable alternative. The distributor could be even more intrusive by scanning for other information, such as installed applications and browsing history, in the same way. Moreover, a tracking-enabled EPUB might try to associate the user with an online browsing session, e.g. by fingerprinting the installed fonts. The EPUB might even try to obtain an even more intrusive device fingerprint, e.g. by detecting the presence of specific files on the system, which in most cases can also reveal the username.

IV. Methodology

To evaluate the potential threats posed by opening an EPUB file, we conduct a series of experiments. In this section we describe our experimental setup through which we test a wide variety of EPUB reading systems for different primitives that could be used to launch attacks.

A. Experimental design

Our experiments aim to document the capabilities entrusted to EPUBs by the reading systems, and to detect related security and privacy issues. Because most reading systems are closed-source, we opt for a black-box approach, developing a testbed of various EPUBs which, upon loading, instruct the EPUB reading system to run embedded experiments. Because of the high variety of reading systems, both in terms of the platform they are run on as well as the functionality they provide and their user-interface, we deemed it infeasible to perform a fully automated evaluation while maintaining completeness. Instead, we opted for a semi-automated approach, where we use JavaScript code to render the results of our experiments in the reader, or, if possible, send these to a remote web server. As such, the manual effort is limited to copying this output from the EPUB reader into a file that can be further evaluated by our analysis framework. An overview of our experimental design can be found in Figure 2.

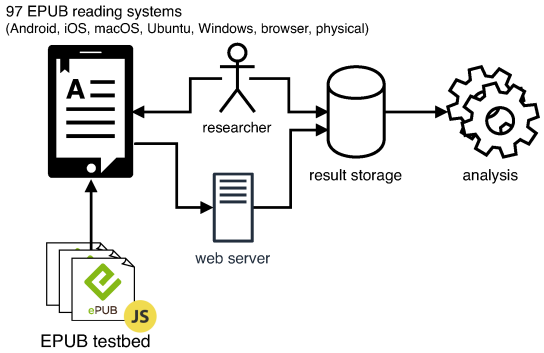

Fig. 2. Overview of our experimental design. The various EPUB files that make up our testbed are manually loaded in the tested reading system. If remote communication is available, the results are automatically submitted to a web server, which will store it in the database. Alternatively, these are manually copied from the e-book.

Supported by this setup, we aim to evaluate the presence of certain "primitives" that are required to launch attacks. For instance, in order to leak the contents of a file on the local file system, an attacker requires the ability to render content from local files, execute JavaScript code, and finally send remote requests. For every primitive functionality, our testbed uses a separate EPUB file that tests its presence. The reason for this is that EPUB reading systems label certain EPUBs as corrupt when these try to execute unsupported functionality. Several experiments rely on specific functionality such as JavaScript execution; when this functionality is not present, the associated experiments can be omitted. The decision on which experiment to perform next is each time imposed by our testbed protocol.

We used the official EPUB Validation Tool [59] provided by W3C to validate conformance with the standard. To accommodate all EPUB reading systems, the embedded JavaScript uses ECMAScript 5 functionality because the more recent ECMAScript 6 is not widely supported among reading systems. We have publicly released all code required to construct this testbed of EPUBs.[2]

In the rest of this section we discuss all features that were evaluated, an overview is depicted in Figure 3.

1) JavaScript execution: Because most reading systems do not disclose whether JavaScript is supported, which is indeed an optional feature in the EPUB specification, this information

[2]https://github.com/DistriNet/evil-epubs

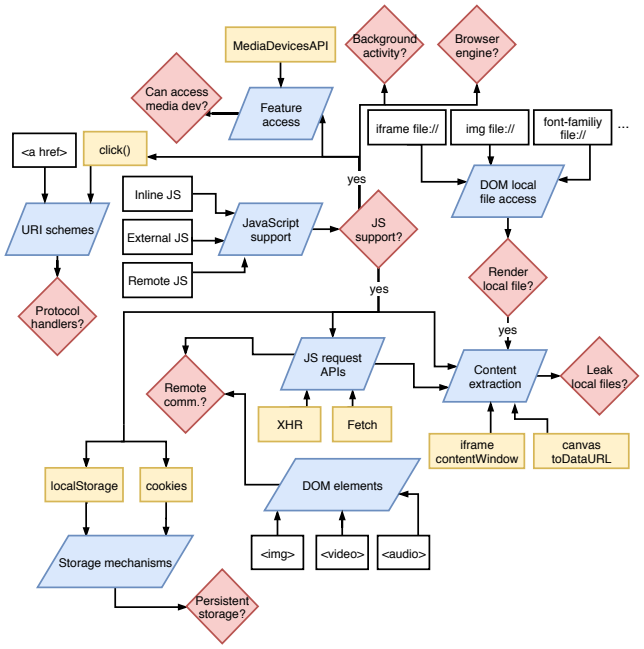

Fig. 3. Overview of the different EPUB experiments. In order to assess certain features (red) of the reading system, we used several experiments (rectangular), both with (yellow) and without (white) JavaScript; these experiments are grouped by category (blue).

needs to be obtained empirically. JavaScript support might be an important trait to the user, e.g. to support interactive EPUBs, but even more so to a potential attacker, considering the substantially increased threat surface. That is, through JavaScript a multitude of different APIs become available, which could be used to request local or remote resources, or even access user media devices (MediaDevices API [45]).

We test three different ways of how JavaScript can be included in an EPUB: (i) directly embedding code with a `<script>` tag in an XHTML file (inline), (ii) reference a separate JavaScript file within the EPUB by setting the `src` attribute of `<script>` tags (external), and (iii) reference a JavaScript file hosted on an external web server (remote). All three approaches were evaluated by dynamically changing the content of a visible HTML element through inline, external or remote JavaScript code. When the content of such an element assumed the dynamically assigned value over the original value, we could safely assume that JavaScript was executed.

2) Local file system access: The EPUB specification allows reading systems to support references to certain types of resources on the local file system, explicitly mentioning audio, video and fonts, but also any resource retrievable by a script [1]. JavaScript-supporting reading systems that implement this optional feature may implicitly grant every EPUB the ability to retrieve files from the user's operating system. Even when the SOP is enforced to prevent content leaking, as is recommended by the specification [2], a malicious EPUB might still be able to gather sensitive information such as the presence of certain files, or even the user's account name.

For this evaluation, we performed three sets of experiments in which the EPUB attempts to access five types of resources: textual files (.html, .txt, .log, .bogus), images (.png, .jpg), audio (.mp3), video (.mp4) and fonts (.ttf). In an attempt to bypass the potentially restrictive direct access to the local file system, we do not only refer to the resource through its absolute path (`file://` protocol), but also leverage relative symbolic links. For UNIX systems, we were able to enclose correctly functioning symbolic links pointing to a file and folder in the ZIP file, which is essential for embedding them in an EPUB. We did not find a way to reproduce this on Windows. These experiments were considered successful only when the obtained information could be leaked to a remote server (see Experiment IV-A3).

In the first set of experiments, the EPUB attempts to render the local user files by simply including them by means of an `iframe`, `img`, `audio` or `video` element, or by assigning a CSS `font-family` to include a font. When such a resource is rendered, it is trivial to confirm its existence on the local file system. For iframes and images, an observable `load` event is fired when the subresource was successfully loaded. Similarly, on audio and video elements the `canplaythrough` event is fired. Although no such event exists for fonts, existence of font files can be inferred by leveraging `canvas` elements to check whether the referred font has been applied to a text box.

The second set of experiments aims to determine whether the content of local resources can be accessed through the XMLHttpRequest and Fetch API [43], [47] or by leveraging content-specific methods. For textual resources, the EPUB tries to access the rendered content within an iframe through its `contentWindow` attribute. Images, on the other hand, can be encoded in the base64 format through the `toDataURL` functionality of the `canvas` element. However, when reading systems use a unique domain to host the EPUB's content, as is recommended by the specification [2], the SOP disallows access to the content of the referred resource. Again, here we can leverage symbolic linking to make it appear as if the referred content is hosted on the same domain.

Well-secured EPUB reading systems will prevent the EPUB from rendering local files and leaking their content, however, we might still be able to leak the existence of a particular file by leveraging timing attacks. This was evaluated in an additional experiment, by measuring the time between setting the `src` attribute of an image element and the firing of the `onerror` event, for both a URL of an existing file and a non-existing file. In every experiment, this measurement was performed 20,000 times, alternating the sequence order of the existing and non-existing file to reduce potential noise. When in a significant portion of these cases the measured time was larger for the existing file than for the non-existing file, or vice-versa, we consider a timing attack to be viable. For all such labeled reading systems, this calculated accuracy was at least 95%, except for two reading systems where we measured an accuracy of about 75%. However, in all cases the accuracy can be increased by performing multiple measurements. On MacOS and Linux, we used a filesystem in user space (FUSE) [50] in advance to determine whether the local filesystem is accessed in an attempt to read out the file.

3) Remote communication: Similar to the local resource access discussed in the previous section, the EPUB specification allows reading systems to support references to online resources for certain resource types [1], implying that remote communication with a server is possible. However, the standard also acknowledges the security implications produced by this trait and advises reading system developers to explicitly notify users of network traffic, and ideally, even request user consent in advance [2]. Indeed, this capability is essential for relaying sensitive information to a tracker, or for receiving instructions from an attacker.

In this experiment, we investigate whether an EPUB is able to communicate with remote servers while it is opened by the reading system, and whether the user is notified of the occurring network traffic. Various HTML tags can be used to initiate HTTP requests, and in an attempt to be exhaustive we leveraged the comprehensive collection on the HTTPLeaks GitHub repository [25], in combination with the XMLHttpRequest and Fetch API [43], [47]. When any of these requests reaches the remote server, we label the EPUB reading system as supporting remote communication. As we manually load the crafted EPUBs in the readers, we also take note of any request for consent that was presented to the user.

4) Persistent storage: In modern browsers, websites have access to various mechanisms to store data locally, such as cookies and the LocalStorage API [33]. Again, EPUB reading systems might inherit this functionality to provide storage capabilities to EPUBs. The EPUB specification rightfully recommends reading system developers to treat all stored data as sensitive, preventing other documents from accessing.

In these experiments, we first determine whether the EPUB reading system supports persistent storage through one of the two mechanisms. Since reading systems might merely provide the API, neglecting the persistence trait, we evaluate whether the stored information persists after closing the EPUB reading system. To adequately validate inter-session persistence, we start an initial session by opening the crafted EPUB. After rendering is complete, we close the reading system, thereby ending the first session. Finally, by reopening the same EPUB file and starting a second session, we inquire whether any cookies or LocalStorage entries have remained.

In an additional experiment, we check compliance with the recommendation to isolate this data from other documents. For this, we use different EPUBs in subsequent sessions, validating whether a modification by the first EPUB is detectable by the second.

5) Feature access: Modern browsers allow websites to request access to features, such as the user's geolocation, microphone and webcam [20], [51]. When such access is requested, the browser will ask the user for consent to allow the website to access the indicated resource. We did not find any occurrence of these mechanisms in the EPUB specification, however, since most EPUB reading systems rely on browser engines, it is possible that this functionality is inherited. Because access to these media devices could allow an EPUB to record the user's surroundings or determine the user's location, it proves a tempting target for a potential attacker.

In this experiment, we evaluate whether the GeoLocation and MediaDevices API are made available in EPUB reading systems, and if so, whether user consent is required.

6) URI schemes: On the Internet, resources are referenced through Uniform Resource Identifiers (URI), of which most rely on the `http:` or `https:` protocol. However, by using custom URI schemes, websites can also instruct the browser to open applications upon activation of the URI (e.g. by clicking a hyperlink), even passing on arguments in the URI. For instance, the `mailto:` scheme is often employed to refer to an e-mail address, and when activated, will open the operating system's default mail application [29]. Whereas the `mailto:` scheme is one of the official URI schemes issued by the Internet Assigned Numbers Authority (IANA) [16], there are also many non-registered schemes used in practice.

To prevent misuse, modern browsers generally request confirmation from the user to initiate another application. This precaution is considered critical as URI links can be activated without any user interaction, e.g. through the `click()` function in JavaScript. Depending on the security considerations of a referred application, leveraging the arguments of such an activation could initiate a phone call, send a mail or download a file, facilitating various attacks by respectively exposing a user's phone number or e-mail address, or downloading a malicious payload.

In this experiment, we investigate whether EPUB reading systems support initiation of applications through URI schemes, and if so, whether the reader requested permission from the user for this action.

7) Browser engine evaluation: Considering that browser engines require regular patching to fix security bugs, disclosed vulnerabilities could be abused to target reading systems with an outdated browser engine.

In this experiment, we explore browser engine use in EPUB reading systems by evaluating whether the embedded browser engine is outdated and insecure. While at first sight consulting the user-agent string poses a straightforward solution, this information might not correctly represent the underlying browser engine. For instance, reading systems could have modified it, and WebKit has stopped updating the user-agent string altogether [61]. Therefore, we identify the embedded browser engine version by fingerprinting browser engines based on supported features, leveraging MDN's browser compatibility dataset [44]. Such a fingerprint is constructed by evaluating support for each HTML element, attribute and JavaScript API present in the MDN dataset. This way, we collected almost 100 distinct fingerprints from applications whose embedded browser engine is known, and subsequently used those to determine the embedded browser engine of the reading systems. A browser engine is marked insecure if its age has surpassed at least three years, and if any vulnerabilities are publicly disclosed.

8) Background activity: To facilitate multi-tasking, mobile applications retain operation for a short time after focus is lost (e.g. when the user switches to another app), depending on the application's configuration. However, to improve battery life

TABLE I
EVALUATION RESULTS FOR EPUB READING SYSTEMS ON WINDOWS.

Application	JavaScript Local	JavaScript Remote	Local Resources Existence	Local Resources Render	Local Resources Leak	Remote communication	Persistent storage Cookies	Persistent storage LocalStorage	Features	URI handles	Insecure engine
Adobe Digital Editions (4.5.10)	●	●	🖼	🖼	‡	-	●	○	●	-	○	○
Bibliovore (2.0.2.0)	○	○	-	-	-	-	○	○	○	-	○	-
BookReader (1.6.0.0)	○	○	-	-	-	-	○	○	○	-	○	-
Bookviser Reader (6.8.1.0)	○	○	-	-	-	-	○	○	○	-	○	-
Calibre (3.40.1)	●	●	</> 🖼 🗋 🗎 ■ 🔊 📼 𝑥	</> 🖼 🗋 🗎 🔊 𝑥	-	●	○	○	-	●	●
(4.3.0)*	●	○	-	-	-	●	○	○	-	○§	○
CoolReader (n/a)	○	○	-	-	-	-	○	○	○	-	○	-
EPUB File Reader (1.5)	●	●	</> 🖼 🗋 🔊 📼	</> 🖼 🗋 🔊 📼	</> 🖼 🗋	●	○	○	○	-	○	○
FBReader (0.12.10)	○	○	-	-	-	○	○	○	○	-	○	-
Freda (4.21)	○	○	-	-	-	○	○	○	○	-	○	-
Icecream Ebook Reader (5.19)*	●	○	</> 🖼 🗋 🗎 𝑥	</> 🖼 🗋 🗎 𝑥	</> 🖼 🗋 🗎	●	○	○	○	-	○	●
Liberty (1.0.0.13)	○	○	-	-	-	●	○	○	○	-	○	-
MS Edge (44.17763.1.0)	●	●	🔊 📼	🔊 📼	-	●	○	○	○	-	●†	○
Nook (1.10.1.15)	○	○	-	-	-	○	○	○	○	-	○	-
Overdrive (3.8.0)	○	○	-	-	-	●	○	○	○	-	○	-
SumatraPDF (3.1.2)	○	○	-	-	-	○	○	○	○	-	○	-

</>: .html file 🖼: .png/.jpg file 🗋: .txt file 🗎: .log file ■: .bogus file 🔊: .mp3 file 📼: .mp4 file 𝑥: .ttf

* Only executes inline JavaScript.
† Requires user consent.
‡ Additionally renders textual files (.html, .txt), images (.png, .jpg), audio and video residing on a connected network share.
§ Allows EPUB to open URL in default browser.

and memory consumption, mobile platforms impose restrictions on background tasks. We did not find any official documentation on how much time an application is allowed to run in the background on iOS, yet an Apple staff member has stated on the official Apple Developer Forums that this is around three minutes after losing focus [10]. Similarly for Android, this exact time limit is undocumented, but is said to be around ten minutes before the application is forced into idle mode [7].

Likewise, EPUB reading systems can invoke this functionality to remain running when switched to the background, increasing the time window of a potential attack. By embedding a counter inside the EPUB, we can detect whether a switch to the background paused the embedded JavaScript execution.

B. Evaluated EPUB reading systems

This testbed was used to evaluate a set of 92 free EPUB reading systems, available for desktop platforms (Windows 10, macOS 10.14.6 and Linux Ubuntu 18.04), mobile platforms (iOS 12 and Android 9) or as browser extensions (Chrome 78 and Firefox 70). We used the iOS App Store, Google Play Store, Chrome Web Store and Firefox Add-on Store for selecting and installing reading systems on iOS, Android, Chrome and Firefox. The selection was based on the store's search functionality, using the terms "epub reader" and "ebook reader", scanning the first 100 results each time. For Android, we limited our selection to applications that were downloaded by at least 5.000 users. For the desktop platforms, we used a web search engine to make up the selection of EPUB reading systems, also leveraging curated lists. Here, we installed all encountered applications by downloading them from a website or installing them using the respective application store of the platform. By converting the evaluation EPUBs to AZW e-books, we also evaluated Kindle applications if available on the platform. For a complete overview of all evaluated EPUB reading systems, we refer to Appendix A.

Additionally, we evaluated the five most popular physical e-reader devices (Kindle Paperwhite 4, PocketBook Touch HD 3, Kobo Clara HD, Onyx Nova Pro, Tolino Shine 3). Their pre-installed EPUB rendering applications were tested out-of-the-box.

V. RESULTS

This section will cover the results obtained by performing the semi-automated evaluation described in the previous section. Some reading systems were not able to render a perfectly compliant EPUB 3.2 e-book and were therefore excluded from our evaluation (see Appendix A).

A. Desktop

For desktop-based reading systems, experiments were run on Windows 10 (17763), macOS (10.14.6) and Ubuntu (18.04).

1) Windows: Table I shows the results of our evaluation on the Windows platform, which consisted of 15 reading systems. Of these reading systems, five execute embedded JavaScript, which can be escalated to leak at least the existence of certain files, and two of them can even be abused to leak file contents. Calibre 3 and MS Edge grant EPUBs the ability to open third-party applications installed on the user's operating system. Interestingly, only the latter asks for the user's consent.

Interestingly, Adobe Digital Editions's rendering behavior differs between files residing on the local file system and files residing on a network share. Although the means of access are identical; through an absolute file path, it only allows an EPUB to render images located on the former, whereas textual files (.html and .txt), images, audio and video can be rendered if located on the latter. This can be exploited to enumerate both local files and files residing on a network share. This vulnerability was assigned CVE-2020-3798, and has been resolved since Adobe's 4.5.11.187303 release of the application [3].

Our tests identified WebKit 538.1 as the underlying browser for both Calibre 3 and Icecream Ebook Reader, which was released in 2014. This engine is considered insecure, since several vulnerabilities are publicly disclosed. For instance, by leveraging such a vulnerability [38], we were able to leak arbitrary file contents in Calibre 3. Fortunately, Calibre started using an updated engine since its major update to version 4, effectively mitigating this vulnerability.

2) macOS: As shown in Table II, except for FBReader and Amazon's Kindle application, all reading systems evaluated on macOS support JavaScript execution. All reading systems

TABLE II
EVALUATION RESULTS FOR EPUB READING SYSTEMS ON MACOS.

Application	JavaScript Local	JavaScript Remote	Local Resources Existence	Local Resources Render	Local Resources Leak	Remote communication	Persistent storage Cookies	Persistent storage LocalStorage	Features	URI handles	Insecure engine
Adobe Digital Editions (4.5.10)	●	○	-	-	-	●	○	○	-	●	○
Apple Books (1.17)	●	●	-	-	-	●	○	●	-	○	○
Azardi (43.1)	●	○	</> 🖼 📄 📄 ■ 🔊 🎞 𝑥	</> 🖼 📄 📄 ■ 🔊 🎞 𝑥	</> 🖼 📄 📄 ■	●	○	●	-	○	●
BookReader (5.14)	●	○	</> 🖼 📄 📄 🔊 🎞 𝑥	</> 🖼 📄 📄 🔊 🎞 𝑥	</> 🖼 📄 📄 ■	●	●	●†	-	●	○
Calibre (3.40.1)	●	●	</> 🖼 📄 📄 ■ 🔊 🎞 𝑥	</> 🖼 📄 📄 🔊 🎞 𝑥	-	●	○	○	-	●	●
(4.3.0)	●	○	-	-	-	●	○	○	-	○*	○
FBReader (0.9.0)	○	○	-	-	-	○	○	○	-	○	-
Kindle (1.25.2)	○	○	-	-	-	○	○	○	-	○*	-
Kitabu (1.2)	●	●	𝑥	𝑥‡	-	●	○	○	-	●	○
Murasaki (1.0.2)	●	●	</> 🖼 📄 📄 ■ 🔊 🎞 𝑥	</> 🖼 📄 📄 🔊 🎞 𝑥	</> 🖼 📄 📄 ■	●	○	○	-	○	○

</>: .html file 🖼: .png/.jpg file 📄: .txt file 📄: .log file ■: .bogus file 🔊: .mp3 file 🎞: .mp4 file 𝑥: .ttf

* Allows EPUB to open URL in default browser.
† Allows access to LocalStorage of other EPUBs.
‡ Attempts to load all resources in the default application without consent (except fonts).

TABLE III
EVALUATION RESULTS FOR EPUB READING SYSTEMS ON LINUX UBUNTU.

Application	JavaScript Local	JavaScript Remote	Local Resources Existence	Local Resources Render	Local Resources Leak	Remote communication	Persistent storage Cookies	Persistent storage LocalStorage	Features	URI handles	Insecure engine
Calibre (3.46)	●	●	</> 🖼 📄 📄 ■ 🔊 🎞 𝑥	</> 🖼 📄 📄	𝑥	●	○	○	-	●	●
(4.3.0)	●	○	-	-	-	●	○	○	-	○*	○
FBReader (0.12.10)	○	○	-	-	-	○	○	○	-	○	-
Okular (1.7.2)	○	○	-	-	-	○	○	○	-	○	-

</>: .html file 🖼: .png/.jpg file 📄: .txt file 📄: .log file ■: .bogus file 🔊: .mp3 file 🎞: .mp4 file 𝑥: .ttf

* Allows EPUB to open URL in default browser.

TABLE IV
EVALUATION RESULTS FOR EPUB READING SYSTEMS ON IOS.

Application	JavaScript Local	JavaScript Remote	Local Resources Existence	Local Resources Render	Local Resources Leak	Remote communication	Persistent storage Cookies	Persistent storage LocalStorage	Features	URI handles	Runs in background
Aldiko Book Reader (1.1.6)	○	○	-	-	-	○	○	○	-	○	○
Apple Books (4.2.3)	●	●	-	-	-	●*	○	●	-	●†	●
Bluefire Reader (2.9)	○	○	-	-	-	○	○	○	-	○	○
CHMate (6.9.1)	●	●	-	-	-	●	○	●	-	○	●
Ebook Reader (4.0.8)	●	●	-	-	-	●	●	●	-	○	○
Eboox (1.60.1)	○	○	-	-	-	○	○	○	-	○	○
Epub Reader (1.1)	○	○	-	-	-	○	○	○	-	○	○
EPUB Reader (5.1.55)	○	○	-	-	-	○	○	○	-	○	○
FBReader (1.0.10)	○	○	-	-	-	○	○	○	-	○	○
Gerty (1.1.5)	●	●	-	-	-	●	○	●§	-	○	○
Kobo Books (9.14)	●	○	-	-	-	●	○	○	-	○	●
Kybook 3 (0.7.8)	○	○	-	-	-	○	○	○	-	○	○
Marvin (3.1.2)	●	○	-	-	-	●	○	●	Location†	●‡	○
Play Books (5.3.0)	○	○	-	-	-	○	○	○	-	○	○
PocketBook (3.2)	●	●	-	-	-	●	○	○	-	○	○
Power Reader (6.10)	●	●	-	-	-	●	●	●	-	○	○
R2 Reader (2.0.1)	●	○	-	-	-	●	○	○	-	○	○
TotalReader (5.1.61)	●	●	-	-	-	●	○	●	-	●	○
YiBook (1.8.6)	○	○	-	-	-	○	○	○	-	○	○
Yomu (2.3.0)	●	●	-	-	-	●	○	●§	Location†	○	○

* Requires user interaction, only the first time.
† Requires user consent.
‡ Requires user constent, except for Mail app.
§ Allows access to LocalStorage of other EPUBs.

that support JavaScript can communicate with a remote server without informing the user. Moreover, half of the ten tested readers can leak the presence of certain resources on the local file system by rendering them. Three of those even allow an attacker to leak arbitrary files to a remote server.

Furthermore, four reading systems allow EPUBs to open installed applications on macOS, leveraging specific URI schemes, without requiring user consent. It is considered good practice for these referred applications to require user interaction before irreversible actions are undertaken, however, this is not always the case. For instance, when Skype for Business is configured as the default app to handle `tel:` scheme URIs, activation of such a URI will immediately lead to calling the included phone number. Although the results of this action are very noticeable, since both a visual and auditory cue are given when the call is initiated, it does not require any user interaction. Correspondingly, an attacker could make the user initiate calls to their premium-rate telephone number, e.g. when there has been no user activity for a certain time.

3) Linux Ubuntu: For the evaluation on the Linux platform, we only found three functioning reading systems, as shown in Table III. Here, Calibre (version 3 and 4) is the only reading system that provides scripting support. Similar to the installations on the other desktop platforms, Calibre 3 uses an outdated browser engine for which a disclosed vulnerability can be exploited to leak arbitrary file contents.

B. Mobile

In this section, we discuss the results for EPUB reading systems on iOS 12 and Android 9. Note that, to improve legibility, we omitted the browser identification column in the mobile platform tables, since all relied on the engine framework provided by the OS. Consequently, these browser engines are implicitly updated with every system update, thus are considered up-to-date.

TABLE V
EVALUATION RESULTS FOR EPUB READING SYSTEMS ON ANDROID.

Application	JavaScript Local	JavaScript Remote	Local Resources Existence	Local Resources Render	Local Resources Leak	Remote communication	Persistent storage Cookies	Persistent storage LocalStorage	Features	URI handles	Runs in background
4shared Reader	○	○	-	-	-	○	○	○	-	○	○
AlReader (1.911805270)	○	○	-	-	-	○	○	○	-	○	○
Aldiko Book Reader (3.1)	○	○	-	-	-	○	○	○	-	○	○
Aldiko Classic (3.1.3)	○	○	-	-	-	○	○	○	-	○	○
Bookari Free (4.2.5)	○	○	-	-	-	○	○	○	-	○	○
Book Reader (1.12.12)	○	○	-	-	-	○	○	○	-	○	○
Cool Reader (3.2.32)	○	○	-	-	-	○	○	○	-	○	○
Ebook Reader (1.0)	○	○	-	-	-	○	○	○	-	○	○
Ebook Reader (5.0.8.2)*	●	○	-	-	-	●	○	●§	-	●	●
EBook Reader (3.5.0)	○	○	-	-	-	○	○	○	-	○	○
eBoox (2.22)	○	○	-	-	-	○	○	○	-	○	○
ePub Reader (2.1.2)†	●	●	</> .png .txt .log .bogus .mp3 .mp4 .ttf	</> .png .txt .log .bogus .mp3 .mp4 .ttf	-	●	○	○	-	●	●
Epub reader (4.0)	○	○	-	-	-	○	○	○	-	○	○
Epub Reader (librera) (8.0.39)	○	○	-	-	-	○	○	○	-	○	○
EPUBReader (1.0.32)	●	○	</> .png .txt .log .bogus .mp3 .mp4 .ttf	.mp3 .mp4 .ttf	-	○	○	●	-	●	○
eReader Prestigio (6.0.0.9)	○	○	-	-	-	○	○	○	-	○	○
FBReader (3.0.15)	○	○	-	-	-	○	○	○	-	○	○
Freda (4.31)	○	○	-	-	-	○	○	○	-	○	○
FullReader (4.1.4)	○	○	-	-	-	○	○	○	-	○	○
Gitden Reader (4.5.3)	●	●	</> .png .txt .log .bogus .mp3 .mp4 .ttf	</> .png .txt .log .bogus .mp3 .ttf	</> .png .txt .log .bogus .mp3 .mp4 .ttf	●	○	●§	-	○	●
Google Play Books (5.2.7)	○	○	-	-	-	○	○	○	-	○	○
Infinity Reader (1.7.57)	●	●	-	-	-	●	○	○	-	○	●
iReader (1.1.4)	●	○	</> .png .txt .log .bogus .mp3 .mp4 .ttf	.png .mp3 .mp4 .ttf	-	●	○	●§	-	●	●
Kindle (3.2.0.35)	○	○	-	-	-	○	○	○	-	○	○
Librera (8.1.242)	○	○	-	-	-	○	○	○	-	○	○
Lit Pub (3.5.3)	●	○	</> .png .txt .log .bogus .mp3 .mp4	</> .png .txt .log .bogus .mp3 .mp4	</> .png .txt .log .bogus .mp3 .mp4	●	○	●§	-	○	○
Lithium (0.21.1)	●	●	-	-	-	●	○	○	-	●‖	●
Moon+ Reader (5.1)	○	○	-	-	-	○	○	○	-	○	○
PocketBook (3.21)‡	●	○	-	-	-	○	○	○	-	○	○
Reader FB2 (1.20)	○	○	-	-	-	○	○	○	-	○	○
ReadEra (19.07.28)	○	○	-	-	-	○	○	○	-	○	○
Reasily (1907d)	●	○	</> .png .txt .log .bogus .mp3 .mp4 .ttf	</> .png .txt .log .bogus .mp3 .mp4 .ttf	</> .png .txt .log .bogus .mp3 .mp4	●	○	●§	-	●	●
Solati Reader (2.5.1)	○	○	-	-	-	○	○	○	-	○	○
Supreader (3.2.30)	●	○	-	-	-	●	○	○	-	●	●
Tolino (4.10.2)	○	○	-	-	-	○	○	○	-	○	○

</>: .html file .png: .png/.jpg file .txt: .txt file .log: .log file .bogus: .bogus file .mp3: .mp3 file .mp4: .mp4 file .ttf: .ttf

* Crashes when attempting to render resource on local file system.
† Opens all referenced resources in a new frame.
‡ Only executes inline JavaScript.
§ Allows access to LocalStorage of other EPUBs.
‖ Requests user consent only when browser is launched.

TABLE VI
EVALUATION RESULTS FOR EPUB READING EXTENSIONS FOR CHROME

Application	JavaScript Local	JavaScript Remote	Local Resources Existence	Local Resources Render	Local Resources Leak	Remote communication	Persistent storage Cookies	Persistent storage LocalStorage	Features	URI handles
Chrome extensions (3)	●*	○	-	-	-	●	●†	●†	Microphone, Camera, Location‡	●‡
Chrome extensions (2)	○	○	-	-	-	●	○	○	-	●‖
Firefox extensions (5)	○	○	-	-	-	●	○	○	-	●‖

* Only external JavaScript.
† Can access cookies and LocalStorage of other EPUBs.
‡ Requires user consent.
‖ Requires user click, and only works for `mailto:`.

1) iOS: Out of the 20 evaluated iOS reading systems, 11 support JavaScript and allow EPUBs to communicate with servers over the Internet without user consent, as shown in Table IV. Only Apple Books requires explicit user interaction to permit the EPUB to communicate remotely, which is then remembered between sessions. Furthermore, two reading systems allow EPUBs to share LocalStorage data, while four reading systems allow it to access the GeoLocation API or enable it to open other applications, which in most cases requires user consent.

As a result of the iOS application platform design, the EPUB is isolated from the rest of the file system: the user has to select the EPUB that will be loaded, and the application can only access this particular file. Consequently, access to local resources is blocked by design as these are not available within the application.

2) Android: In contrast to the iOS reading systems, almost every reading system requests the permission "Photos, media and files on your device", either upon installation or when attempting to import an EPUB. Interestingly, most also make use of Android's Storage Access Framework (SAF) [8], an API to access user-selected files, which is more constrained but subsequently does not need explicit permissions to facilitate importing EPUBs. However, when SAF is used in combination with the file permissions, which was the case for all but two applications, this does not prevent attacks that leak arbitrary file contents. In fact, for three Android applications we could successfully leak arbitrary file contents to a remote server, as shown in Table V.

Again, the results show that JavaScript support provides additional capabilities that often can be abused. Seven reading systems grant the ability to open other applications, of which only one asks for user permission when this referred application is the browser (i.e. by using the `http` or `https` scheme). Furthermore, out of six applications that support access to the LocalStorage API, five do not provide sufficient isolation, thus allowing access to content saved by other EPUBs.

C. Browser extensions

For both Chrome and Firefox, we evaluated five extensions that are advertised as EPUB reading systems. As shown in Table VI, three Chrome extensions allow JavaScript execution, giving them access to persistent storage, and even to

the microphone, camera or location (provided that the user consents). Because all EPUBs opened by these applications shared the same origin (chrome-extension://[extension_id]), EPUBs can access the persistent storage of e-books that were opened previously.

The remaining seven extensions do not allow JavaScript execution as a result of the imposed Content Security Policy (CSP), prohibiting all inline JavaScript and only allowing resources local to the extension [42]. Although remote resources are blocked by the CSP, the extensions still provide functionality to fetch these (hence the ability to perform remote communication). In Section VI we show how this functionality lead to a universal XSS in EPUBReader (available on both Chrome and Firefox). This was also the only extension that automatically rendered an EPUB when a referring link is clicked (achievable through JavaScript) in Chrome.

D. Physical e-reader devices

Only for the Kobo e-reader, our testbed confirmed limited JavaScript support, as is documented by Kobo [37]. Note that the e-reader only executes embedded scripts for KEPUB files (Kobo's custom e-book format), these are created by simply changing the `.epub` file extension to `.kepub.epub`. Furthermore, we could use the e-reader's internet connection to contact remote servers without user consent. Finally, the embedded browser engine framework was identified as QT 5.2.1 (released in 2014) for which several vulnerabilities have been reported [26].

Amazon's publishing guidelines affirm that scripting is not supported, and that all scripts are stripped from the source during conversion [6]. However, as part of a manual evaluation of the Kindle, we found this to be inaccurate: the browser engine supports JavaScript execution, although it is disabled by default (in Section VI-C we show how this can be circumvented).

VI. Case studies

To complement our semi-automated evaluation, we manually analyzed a select number of applications for implementational flaws. This selection was based on different characteristics: Apple Books on macOS (popular pre-installed application that supports JavaScript but prevents rendering local files), EPUBReader (the most widely used browser extension on both Chrome and Firefox), and Kindle (the most widely used physical e-reader, with an 83.6% market share in the US [27]).

A. Apple Books

As we discussed in Section V-A2, the capabilities detected by our semi-automated evaluation did not lead to direct local file system access in Apple Books for macOS, even when leveraging symbolic links. However, through manual evaluation, we identified a user information disclosure vulnerability and persistent denial of service vulnerability.

The *user information disclosure* vulnerability allows an attacker to infer whether a specific EPUB is present in the user's library. When an EPUB is opened by Apple Books, it is unpacked and stored in a folder (`Books`) alongside other previously unpacked EPUBs. The contents of each EPUB are stored in a separate folder named after the EPUB's deterministically assigned 32-character serial ID. While we could not infer how this ID is generated exactly, we have verified that an EPUB is assigned the same ID across multiple devices or accounts. Although embedded symbolic links referring outside of this directory are denied, these links would still remain functional when pointing to a location within the EPUB folder, or even within the `Books` directory. As a result, to gain information about the contents of the user library, an EPUB could include a series of symbolic links, referring to potential locations of unpacked EPUBs. By verifying whether an arbitrary file in such a folder can be rendered, the EPUB can disclose whether any of the selected EPUBs is present in the user's library. We found that the iOS applications Gerty and Marvin could be exploited in a similar way.

The *persistent denial of service* attack is achieved by simply including a symbolic link that refers to the `Books` folder in which the EPUBs are unpacked. This will cause Apple Books to crash, reporting that it cannot access the user's library, for every subsequent reboot. Because Apple Books is an integral part of the operating system, it cannot be reinstalled without reinstalling macOS.

In response, Apple issued a CVE for both vulnerabilities (CVE-2019-8789 and CVE-2019-8774, respectively), and distributed a fix through operating system updates [11]–[14].

B. EPUBReader extension

The results of our semi-automated evaluation in Section V-C show that all Firefox extensions and two Chrome extensions block JavaScript execution, due to the imposed Content Security Policy (CSP). Even more interesting; upon installation, three extensions request permission to read and change all data of visited websites, using the `<all_urls>` permission indicator [23]. This allows the extensions to send HTTP requests to any visited website and read out the response. Moreover, if the user is logged in on such a website, the request will implicitly include session cookies, and thus authenticating the user. By bypassing the CSP restrictions for embedded JavaScript in EPUBReader for both Chrome and Firefox, we were able to abuse this permission to steal user account information of any website on which the user is logged in, effectively leading to a universal XSS.

Although including remote resources directly is prevented by EPUBReader's CSP, it still tries to provide this functionality by first fetching the included images and referred media files, and then making their content available through a `blob://` URL (which is allowed by the default CSP). We leveraged this functionality to trick EPUBReader to make a JavaScript file available as a `blob://` URL (by simply including this file as an image). Because these URLs contain an unguessable UUID, we first used a CSS-based data exfiltration technique to leak this to the attacker server. Finally, the adversary can dynamically generate another EPUB that refers to this `blob://` URL, and then tricks the reader to open this generated EPUB

(EPUBReader will automatically try to read EPUBs based on the URL pattern). Finally, the JavaScript payload will be executed, giving the attacker the same privileges as the browser extension (access to all authenticated content on all websites). A proof-of-concept implementation of our attack requires a single user interaction, such as a click, from the victim on Chrome in order to open a new window; on Firefox the attack can be performed without any user interaction, and is unnoticeable. Collectively, this affects almost 300,000 users.

C. Kindle

Our semi-automated evaluation indicated that Kindle does not support JavaScript execution, as confirmed by the publishing guidelines [6]. By reverse engineering the application that renders the converted AZW3 files (`webreader`), we found that WebKit version 1.4.2 is used to render e-books, but that in the browser engine's settings, the `enable-scripts` property is set to `false`. However, the application itself uses JavaScript to extract information from the DOM or to change styles.

Before each JS execution, the `enable-scripts` property is set to `true`, and immediately after it is set back to `false`. Consequently, JavaScript code contained in the EPUB will never be executed. Nevertheless, we found that in various instances dynamic input, which could potentially be under the control of an attacker, is not properly escaped or sanitized. For instance, a script that is executed on every page refresh includes the font that is used. This value can be controlled by the attacker by changing the font, which is done by sending a GET request to the webreader's SOAP server. Another example is the image viewer used to zoom in to images in e-books. Here the reference to the image is not sanitized, allowing an attacker to inject HTML code, including `<script>` elements. This could also be initiated by sending a request to the SOAP server. Although the rendering engine blocks web requests, this can be circumvented by leveraging SVGs. Presumably these are rendered outside of the browser engine, and thus an `<image>` element with the `xlink:href` attribute can still be used to issue requests. Consequently, it is possible to set the font to any value and run arbitrary JavaScript code by escaping the string context with a single quote[3], or inject HTML in the image viewer.

Once the attacker is able to execute JavaScript, WebSockets can be leveraged to send arbitrary requests, as these are not blocked either. Furthermore, in the image viewer both the restriction that JavaScript code is only temporarily executed and the restriction of sending remote requests are lifted. As various applications on the Kindle are controlled via HTTP requests, this means these now fall under the attacker's control. For instance, the ccat service, which provides an HTTP interface on port 9101, is used to manage the user's library. In a proof-of-concept exploit we leveraged CVE-2011-3243, a UXSS vulnerability that has been publicly known for over nine years and only requires a few lines of code to mitigate[4], to read out the entire library of a victim, along with metadata. We also showed that it is possible to leak the contents of documents by interacting with the built-in KFX reader: through an HTTP request the reader is instructed to open the document, after which it is possible to obtain a rendered image showing the contents of the document. These can then be extracted via a remote request to an attacker-controlled server. We did not explore the other, undocumented, closed-source services that are controlled through HTTP requests. To defend against these issues, most Kindle services now require a verification token that indicate the authenticity of requests, preventing an attacker to arbitrarily interact with the services.

VII. Real-world Analysis

In this section, we analyze the EPUB ecosystem by assessing the presence of malicious and tracking EPUBs in the wild, and the feasibility of distributing them through a self-publishing service.

A. Malicious and tracking EPUBs in the wild

In order to investigate whether any of the discussed techniques are currently being used in the wild to either attack or track users, we performed an additional analysis of EPUBs available in a real-world setting. To this end, we downloaded several free EPUBs from five popular online e-book stores (eBooks.com, Google Play Books, Project Gutenberg, Kobo, Amazon). After a manual inspection of the e-books, we did not find any indication of abuse. It should be noted however that this evaluation is limited, and only considers abuse by the EPUB stores; abuse by individual publishers would be infeasible to evaluate from an external perspective, as this would require purchasing a very large number of e-books.

To further evaluate other types of abuse, we obtained a large number of EPUBs from file sharing platforms. More precisely, we downloaded the 1,000 most popular and most recent EPUB torrents from The Pirate Bay and the same amount from 4shared (these are marked as the most widely used sources to illegally obtain an e-book according to a study by Digimarc [28]). In total, we obtained 7,238 EPUB files from torrents (in several cases, a single torrent contained multiple EPUBs), and 1,807 from 4shared. Next, we unpacked all EPUBs and parsed all documents, looking for possible types of abuse (references to files on the local file system, symbolic links, connections to a remote server, and JavaScript inclusions). We did not find any evidence of abuse, either in terms of tracking or attempts to compromise the EPUB reading system. Interestingly, we found that only 65 e-books, less than one percent of all 7,238 considered EPUBs, made use of JavaScript. In most cases, the code was minimal, and was used to change the background color or font size. All e-books were completely functional without executing the JavaScript code.

[3]Because the font name length is limited, the malicious payload has to use `eval()` on the `textContent` of a (hidden) DOM element.

[4]The mitigation only adds one additional if-statement: https://trac.webkit.org/changeset/88071/webkit

B. Malicious EPUB distribution through self-publishing

To explore the feasibility of publishing malicious EPUBs through official e-book vendors, we submitted manuscripts to the six most popular free self-publishing services. For each service, we bought the published version of our manuscript to check whether any of the embedded scripts were still present. The following is a list of these services, along with their associated vendor and its e-book market share according to the 2017 AuthorEarnings report [15]: Kindle Direct Publishing (Amazon, 80%), iBooks Author (Apple Books, 10%), Barnes & Noble Press (Barnes & Noble, 3%), Kobo Writing Life (Kobo, 2%), and Google Books Partner Centre (Google Play Books, 1.4%). No exact figures were available for the sixth tested service, Smashwords, however they also distribute self-published titles through Apple Books, Barnes & Noble, and Kobo in addition to their own website [55].

Of the six vendors, only Google Play Books rejected our submitted manuscript. Although Amazon succeeded in removing most scripts, we were still able to publish the exploit discussed in Section VI-C, which could target millions of Kindle devices. The remaining four vendors appeared to take no vetting measures at all; embedding scripts in a published EPUB was trivial. Of these, only Smashwords provides downloadable EPUB files upon purchase, hence, any EPUB reading system can be used to open them. The other three vendors deliver e-books directly to their associated reading systems (however, this can be circumvented). For Apple Books (application) and Kobo (application and physical e-reader), the embedded scripts were executed, and even allowed remote communication. However, Barnes & Noble's application crashed upon rendering embedded scripts, curbing potential abuse.

In conclusion, our experiment shows that four out of six evaluated self-publishing services can be abused to distribute malicious EPUBs through official vendors. These vendors account for approximately 94% of all EPUB sales, of which at least 33% is attributed to self-published EPUBs according to the 2017 AuthorEarnings report [15]. We notified all five self-publishing services of which the vetting process was deemed inadequate.

VIII. Discussion

Our semi-automated evaluation shows that many of the JavaScript-supporting EPUB reading systems do not correctly enforce the specification's security recommendations, and thus can be abused in several ways. Furthermore, a significant part of these reading systems does not prevent EPUBs from accessing the local file system and even provide JavaScript APIs that are not included in the EPUB specification. In this section, we elaborate on the underlying issues and make suggestions on what can be improved to remedy the various issues.

A. EPUB reading system implementations

In contrast to mobile reading systems, we identified a high variety of rendering engines for desktop reading systems. Moreover, we find that five of the evaluated desktop applications employ an outdated engine, and consequently, a publicly disclosed vulnerability could be leveraged to exploit the application. Even applications that employ a more up-to-date engine may still be affected by so-called n-day vulnerabilities [24], [60], security issues that have been patched in the upstream component (and thus known publicly), but that still affect software that did not yet update this vulnerable component. As it may take days, or even years (e.g. in the case of Calibre 3) to update a known-vulnerable browser engine, we believe this forms a significant threat for EPUB reading systems.

For both mobile platforms, we found that applications relied on the built-in renderer, and thus all share the same version. Another key difference with desktop applications is that mobile reading systems operate from a more sandboxed environment by default. For instance, on iOS, none of the applications requested permission to access other files on the system, and consequently could not be abused to render or leak files on the local system. Although a similar functionality is available on Android, through the Storage Access Framework [8], most applications still required file permissions, and as a result, we managed to detect the existence of files in six applications, and leak their contents in half of those. This highlights that developers should try to use the minimal amount of privileges to reduce the potential consequences of an attack. By further analyzing the Android applications, we found that for two the file-leak vulnerability was caused by configuring the WebView component to allow access to the local file system, using `setAllowFileAccessFromFileURLs` [9].

Although several applications would render files on the local filesystem, not all of them lead to extraction of their contents. Our manual analysis of these cases showed a direct relation to SOP enforcement: the EPUB content was served from a custom, non-existent domain, preventing access to `file://` resources. Yet, not all reading systems implementing this practice were able to achieve complete isolation of the local file system. For instance, Adobe Digital Editions on Windows employs a dedicated domain, but EPUBs are still allowed to render local images or even HTML files on network shares. The latter is especially dangerous, as it gives access to the `file://` from where the local filesystem can be accessed.

B. EPUB specification

Although a valued effort has been made to include effective security recommendations, we argue that the EPUB specification does not impose sufficiently strict requirements for EPUB reading systems. Of course, the responsibility to actually conform their reading system to the specification's security requirements remains that of the developers, however, hardened requirements could eventually be consolidated into a quantified compliance checker application.

Probably even more effective would be attenuating the capabilities that are to be granted according to the EPUB specification. For instance, an EPUB is allowed to only refer to audio, video and fonts through static XHTML and CSS, but any resource is allowed to be retrieved by embedded scripts [2]. This can be useful for keeping the size of an EPUB small, since the more sizable audio and video files can be fetched from an

online service. However, access to resources from the local filesystem, which in the current version of the specification is allowed, introduces a significant threat, which does not outweigh its limited benefits. Furthermore, the ability to render local resources implies the ability to determine their existence, information that can be gained for various purposes among which file system fingerprinting. For this reason, we argue to completely prohibit EPUBs from referring to resources that reside on the user's operating system. Moreover, as reference to remote resources is very rare in EPUBs, we strongly believe that this should require consent from the user, in order to prevent any form of tracking.

Interestingly, our semi-automated evaluation revealed that more than half of the JavaScript supporting reading systems also support GeoLocation and UserMedia APIs, or opening applications through URI handles, functionalities that are not mentioned in the EPUB specification. These functionalities originate from the underlying browser engine, and are likely not considered by the developer. Assuming the EPUB specification does not aspire to incorporate such browser functionalities, we argue that the specification should include a whitelist of APIs that can be enabled.

Based on our real-world analysis of 9,000 EPUBs, we argue that the discussed restrictions for the EPUB specification would have a minimal impact; none of the analyzed EPUBs required local or remote resources to render correctly, and even the few that embedded JavaScript remained functional when execution was prevented. In that regard, we also propose to reconsider the capability of unrestricted JavaScript execution in EPUB reading systems, perhaps requiring user consent when a script is about to be executed.

C. Responsible disclosure

All vulnerabilities, either identified through our semi-automated testbed or our case-studies, were responsibly disclosed to the involved parties. In addition, we sent out an early warning to all vendors whose reading system did not satisfy the specification's security recommendations. In total, we reached out to 33 vendors, responsible for 37 reading systems, each time using the most appropriate private channel that was available. Although we received a generic or no response from the majority, vendors of very popular reading systems such as Apple and Adobe were eager to solve the reported issues, for which three CVEs were issued.

IX. Related work

We did not encounter any prior studies evaluating the implications of web technology use in non-browser applications. However, our work shares several similarities with the following research.

A. Portable Document Format

Today, the Portable Document Format (PDF) is one of the most popular file formats used for operating system independent document exchange. Its capabilities bear close resemblance to those of the EPUB format, including support for scripting and network connectivity. Unfortunately, previous research has demonstrated that these traits may to lead to security, privacy and content integrity vulnerabilities [21], [22], [41]. In that regard, we hope that by expressing our concerns about EPUB capabilities at an early stage, the specification can be adapted to avoid similar consequences.

Various research efforts focus on the use of machine learning to distinguish between benign and malicious PDF files. Research by Smutz et al. and Srndic et al. argue that PDF file metadata and structure are valuable features that can be used by a static, machine learning based detection system [56], [58]. Maiorca et al. demonstrated a new evasion technique for PDF file analysis based on logical structure; they also present a framework to solve this problem [40]. Nissim et al. performed an extensive study reviewing and comparing state-of-the-art techniques for detecting malicious PDF files [49].

B. Comprehensive policy evaluations

Various studies have exposed vulnerabilities and inconsistencies in browser policy implementations through comprehensive evaluations. By combining manual and automated analysis in four popular browsers, Aggarwal et al. uncovered several implementational weaknesses for private browsing modes [4]. Furthermore, they show that some of these weaknesses can be exploited by an attacker to bypass the imposed privacy policy. Schwenk at al., on the other hand, performed a comprehensive evaluation of the same-origin policy in 10 browsers, leveraging an extensive set of 544 different test cases [54]. Their results exposed various vulnerabilities and inconsistencies among browsers, pleading for a formal definition of the same-origin policy. In another study, Franken et al. performed an evaluation of third-party cookie policy implementations in 7 browsers and 46 browser extensions, leveraging their automated framework [32]. According to their results, all imposed third-party cookie policies of all major browsers as well as evaluated extensions can be bypassed. Finally, a longitudinal study by Luo et al., comprising of 20 different mobile browser families, analyzed support for eight different security mechanisms over the course of seven years [39]. Their findings expose various issues such as lacking support and multi-year vulnerability issues, even for several popular mobile browsers.

X. Conclusion

In this paper we report on a semi-automated evaluation to measure the security and privacy practices of 92 free EPUB reading systems and five physical reading devices. Our results show that almost none of the systems that support JavaScript execution adequately adhere to the security considerations of the EPUB specification. For eight reading systems, a malicious EPUB can even extract arbitrary files from the local system.

We are the first to comprehensively evaluate the security and privacy practices of EPUB reading systems, and hope to increase awareness of the associated threat surface among users and developers. Furthermore, we propose that the current security recommendations of the EPUB standard should be refined into binding requirements. To further assist developers,

additional documentation could be provided in more specific terms how existing browser engine frameworks can be correctly incorporated, pointing out critical configuration elements.

In addition to this large-scale evaluation, we also performed a more elaborate manual analysis of a select number of EPUB reading systems. This manual analysis exposed two severe security issues: first, as soon as a malicious e-book would be opened in Kindle, it could leak documents from the user's library; second, the entire browsing session of users with the EPUBReader browser extension can be compromised upon visiting a malicious website. The results also highlight that the outcome of our semi-automated evaluation should be considered a lower bound, and that several security and privacy issues rely on application-specific behavior.

As part of our assessment of the EPUB ecosystem, we performed an analysis of more than 9,000 EPUBs, obtained "in the wild" from five online e-book stores and two popular file sharing platforms, and evaluated the vetting process of six popular self-publishing services. We did not find evidence of any ongoing abuse, indicating that the issues identified through our evaluations are indeed novel. However, this and the fact that four of the evaluated self-publishing services allowed JavaScript inclusion which could lead to publication of malicious EPUBs, make this study timely: we urge developers to further mitigate the identified issues and adopt additional security measures before their users are exploited.

Finally, we demonstrated that the consolidation of established web technologies in non-browser applications does not necessarily imply a proper translation of the web security and privacy primitives. With this study, we hope to have motivated the need for more comprehensive and in-depth evaluations in this largely unexplored research domain.

Acknowledgment

We would like to thank our shepherd Adam Doupé and the anonymous reviewers for their insightful comments. We would also like to extend our gratitude to Lieven Desmet, Victor Le Pochat and Yana Dimova for their helpful feedback. This research is partially funded by the Research Fund KU Leuven.

References

[1] EPUB 3.2. Standard, W3C, May 2019. https://www.w3.org/publishing/epub3/epub-spec.html.

[2] EPUB content documents 3.2. Standard, W3C, May 2019. https://www.w3.org/publishing/epub3/epub-contentdocs.html.

[3] Adobe. Security Updates Available for Adobe Digital Editions — APSB20-23. https://helpx.adobe.com/security/products/Digital-Editions/apsb20-23.html, April 2020.

[4] Gaurav Aggarwal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.

[5] Alexandra Alter. Your e-book is reading you. *The Wall Street Journal*. https://www.wsj.com/articles/SB10001424052702304870304577490950051438304.

[6] Amazon. *Amazon Kindle Publishing Guidelines*, 2019. http://kindlegen.s3.amazonaws.com/AmazonKindlePublishingGuidelines.pdf.

[7] Android. Background execution limits. https://developer.android.com/about/versions/oreo/background.

[8] Android. Open files using storage access framework. https://developer.android.com/guide/topics/providers/document-provider.

[9] Android. WebSettings. https://developer.android.com/reference/android/webkit/WebSettings.html.

[10] Apple. Uiapplication background task notes. https://forums.developer.apple.com/thread/85066.

[11] Apple. About the security content of iOS 13.1 and iPadOS 13.1. https://support.apple.com/en-us/HT210603, February 2020.

[12] Apple. About the security content of iOS 13.2 and iPadOS 13.2. https://support.apple.com/en-gb/HT210721, April 2020.

[13] Apple. About the security content of macOS Catalina 10.15. https://support.apple.com/en-us/HT210634, February 2020.

[14] Apple. About the security content of macOS Catalina 10.15.1, Security Update 2019-001, and Security Update 2019-006. https://support.apple.com/en-us/HT210722, April 2020.

[15] AuthorEarnings. February 2017 Big, Bad, Wide & International Report: covering Amazon, Apple, B&N, and Kobo ebook sales in the US, UK, Canada, Australia, and New Zealand. https://web.archive.org/web/20190218084936/http:/authorearnings.com/report/february-2017/, 2017.

[16] Internet Assigned Numbers Authority. Uniform resource identifier (uri) schemes. https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml.

[17] Internet Assigned Numbers Authority. Media type assignment: epub+zip. https://www.iana.org/assignments/media-types/application/epub+zip, 2014.

[18] Baldur Bjarnason. EPUB javascript security. https://www.baldurbjarnason.com/notes/epub-javascript-security/, July 2012.

[19] Baldur Bjarnason. Javascript in ebooks. https://www.baldurbjarnason.com/notes/javascript-in-ebooks/, February 2012.

[20] Henrik Boström, Cullen Jennings, Anant Narayanan, Jan-Ivar Bruaroey, Daniel Burnett, Adam Bergkvist, and Bernard Aboba. Media capture and streams. Candidate recommendation, W3C, July 2019. https://www.w3.org/TR/2019/CR-mediacapture-streams-20190702/.

[21] Ron Brandis and Luke Steller. Threat Modelling Adobe PDF. Technical report, Defence Science and Technology Organisation, 2012. https://www.dst.defence.gov.au/sites/default/files/publications/documents/DSTO-TR-2730.pdf.

[22] Aniello Castiglione, Alfredo De Santis, and Claudio Soriente. Security and privacy issues in the portable document format. *Journal of Systems and Software*, 83(10):1813 – 1822, 2010.

[23] Chrome. Declare Permissions and Warn Users. https://developer.chrome.com/extensions/permission_warnings.

[24] Ang Cui. The overlooked problem of 'n-day' vulnerabilities. https://www.darkreading.com/vulnerabilities---threats/the-overlooked-problem-of-n-day-vulnerabilities/a/d-id/1331348, March 2018.

[25] Cure53. HTTPLeaks. https://github.com/cure53/HTTPLeaks, 2019.

[26] CVE Details. QT 5.2.1 Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-12593/product_id-24410/version_id-164958/Digia-QT-5.2.1.html.

[27] Matt Day and Jackie Gu. The enormous numbers behind Amazon's market reach. https://www.bloomberg.com/graphics/2019-amazon-reach-across-markets/, March 2019.

[28] Digimarc. Inside the mind of a book pirate. https://www.digimarc.com/docs/default-source/default-document-library/inside-the-mind-of-a-book-pirate, 2017.

[29] M. Duerst, L. Masinter, and J. Zawinski. The 'mailto' uri scheme. RFC 6068, RFC Editor, October 2010.

[30] Eric Hellman. Publishing Hackathon Pretty Much Ignores eBooks. https://go-to-hellman.blogspot.com/2013/05/publishing-hackathon-pretty-much.html, 2013.

[31] Alison Flood. Ebooks can tell which novels you didn't finish. *The Guardian*. https://www.theguardian.com/books/2014/dec/10/kobo-survey-books-readers-finish-donna-tartt.

[32] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. Who left open the cookie jar? a comprehensive evaluation of third-party cookie policies. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 151–168, Baltimore, MD, August 2018. USENIX Association.

[33] Ian Hickson. Web storage (second edition). W3C recommendation, W3C, April 2016. http://www.w3.org/TR/2016/REC-webstorage-20160419/.

[34] Intellectual Property Office. Online copyright infringement tracker: Latest wave of research. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/628704/OCI_-tracker-7th-wave.pdf, 2017.

[35] Jun Kokatsu. Is your ePub reader secure enough? https://shhnjk.blogspot.com/2017/05/is-your-epub-reader-secure-enough.html, May 2017.

[36] Martin Kaste. Is your e-book reading up on you? https://www.npr.org/2010/12/15/132058735/is-your-e-book-reading-up-on-you.
[37] Kobo Labs. Kobo EPUB guidelines. https://github.com/kobolabs/epub-spec/blob/master/README.md.
[38] Jung Hoon Lee. Issue 1134: WebKit: UXSS via ContainerNode::parserRemoveChild (2). https://bugs.chromium.org/p/project-zero/issues/detail?id=1134, 2017.
[39] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. Time does not heal all wounds: A longitudinal analysis of security-mechanism support in mobile browsers. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*, Jan 2019.
[40] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 119–130, New York, NY, USA, 2013. ACM.
[41] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. Vulnerability Report: Attacks bypassing the signature validation in PDF. Technical report, Ruhr-Universität Bochum, 2018. https://www.nds.ruhr-uni-bochum.de/media/ei/veroeffentlichungen/2019/02/12/report.pdf.
[42] Mozilla Developer Network. Content Security Policy. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_Security_Policy.
[43] Mozilla Developer Network. Fetch API. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.
[44] Mozilla Developer Network. mdn-browser-compat-data. https://github.com/mdn/browser-compat-data.
[45] Mozilla Developer Network. MediaDevices. https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices.
[46] Mozilla Developer Network. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.
[47] Mozilla Developer Network. XMLHttpRequest. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.
[48] Nate Hoffelder. An epub3 ebook could be used to hack your tablet, steal your identity, and cause the downfall of western civilization. 2013. https://the-digital-reader.com/2013/06/09/eric-hellmans-publishing-hackathon-entry-could-be-used-to-hack-your-tablet-steal-your-identity-and-cause-the-downfall-of-western-civilization/.
[49] Nir Nissim, Aviad Cohen, Chanan Glezer, and Yuval Elovici. Detection of malicious pdf files and directions for enhancements: A state-of-the art survey. *Computers & Security*, 48:246 – 266, 2015.
[50] nrclark. Pyfuse: A tool for simple FUSE Filesystems. https://github.com/nrclark/pyfuse, 2019.
[51] Andrei Popescu. Geolocation API specification 2nd edition. W3C recommendation, W3C, November 2016. https://www.w3.org/TR/2016/REC-geolocation-API-20161108/.
[52] PricewaterhouseCoopers. Turning the page: The future of ebooks. 2010. https://www.pwc.co.uk/assets/pdf/ebooks-trends-and-developments.pdf.
[53] Project Gutenberg. Project Gutenberg Submission Guidelines. https://web.archive.org/web/20181108181052/https://upload.pglaf.org/.
[54] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 713–727, Vancouver, BC, August 2017. USENIX Association.
[55] SmashWords. Smashwords Distribution Network. https://www.smashwords.com/distribution, 2019.
[56] Charles Smutz and Angelos Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 239–248, New York, NY, USA, 2012. ACM.
[57] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 179–194. ACM, 2017.
[58] Nedim Srndic and Pavel Laskov. Detection of malicious pdf files based on hierarchical document structure. In *NDSS*, 2013.
[59] W3C. EPUBCheck. https://github.com/w3c/epubcheck, 2019.
[60] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. Detecting "0-day" vulnerability: An empirical study of secret security patch in oss. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 485–492. IEEE, 2019.
[61] WebKit Bugzilla. Limit user agent versioning to an upper bound. https://bugs.webkit.org/show_bug.cgi?id=180365.
[62] Rüdiger Wischenbart. *The global eBook market: current conditions & future projections*. O'Reilly Media, Inc., 2013.

APPENDIX A
ADDITIONAL READING SYSTEM INFORMATION

For measures of completeness and transparency, we provide an overview of all EPUB reading systems that were considered during our evaluation. We also included the number of reported users. Unfortunately, this metric is not available for iOS, so instead the number of ratings can be used as an estimator of the relative reach of an application. Additionally, we reported on the deduced embedded browser engine for all reading systems supporting JavaScript. Here, "OS" indicates that the reading system relies on the engine framework provided by the operating system, and thus is considered up-to-date. Finally, we also include the readers that were excluded from our analysis along with the reason. In total, we considered 92 reading applications on seven platforms (Windows, Ubuntu, macOS, iOS, Android, Firefox & Chrome extensions) and five stand-alone physical e-readers.

TABLE VII
EVALUATED EPUB READING SYSTEMS FOR WINDOWS

Reading system	Version	Rendering engine	Release date
Adobe Digital Editions	4.5.10	OS Trident	N/A
Bibliovore	2.0.2.0	-	-
BookReader	1.6.0.0	-	-
Bookviser Reader	6.8.1.0	-	-
Calibre	3.40.1	WebKit 538.1	Oct 2014
Calibre	4.3.0	Blink 77	Sep 2019
CoolReader	N/A	-	-
EPUB File Reader	1.5	OS Trident	N/A
FBReader	0.12.10	-	-
Freda	4.21	-	-
Icecream Ebook Reader	5.19	WebKit 538.1	Oct 2014
Liberty	1.0.0.13	-	-
MS Edge	44.17763.1.0	EdgeHTML 18.17763	Oct 2018
Nook	1.10.1.15	-	-
Overdrive	3.8.0	-	-
SumatraPDF	3.1.2	-	-

TABLE VIII
OMITTED EPUB READING SYSTEMS FOR WINDOWS

Reading system	Reason
Cover	Unable to open fully compliant EPUB file.
Epub3 Reader	Unable to correctly render fully compliant EPUB file.
FlyReader	Unable to open fully compliant EPUB file.
Perfect PDF Reader	Unable to correctly render fully compliant EPUB file.

TABLE IX
EVALUATED EPUB READING SYSTEMS FOR MACOS

Reading system	Version	Rendering engine	Release date
Adobe Digital Editions	4.5.10	OS WebKit	N/A
Apple Books	1.17	OS WebKit	N/A
Azardi	43.1	Gecko 38	May 2015
BookReader	5.14	OS WebKit	N/A
Calibre	3.40.1	WebKit 538.1	Oct 2014
Calibre	4.3.0	Blink 77	Sep 2019
FBReader	0.9.0	-	-
Kindle	1.25.2	-	-
Kitabu	1.2	OS WebKit	N/A
Murasaki	1.0.2	OS WebKit	N/A

TABLE X
OMITTED EPUB READING SYSTEMS FOR MACOS

Reading system	Reason
Kobo for Desktop	Unable to side-load EPUBs.

TABLE XI
EVALUATED EPUB READING SYSTEMS FOR LINUX UBUNTU

Reading system	Version	Rendering engine	Release date
Calibre	3.46	WebKit 538.1	Oct 2014
Calibre	4.3.0	Blink 77	Sep 2019
FBReader	0.12.10	-	-
Okular	1.7.2	-	-

TABLE XII
OMITTED EPUB READING SYSTEMS FOR LINUX UBUNTU

Reading system	Reason
Bookworm	Unable to install.
Buka	Unable to start.
Cool Reader	Unable to start.
Easy eBook Viewer	Unable to open fully compliant EPUB.
GNOME Books	Unable to open fully compliant EPUB.
Lucidor	Unable to start.

TABLE XIII
EVALUATED EPUB READING SYSTEMS FOR IOS

Reading system	Version	Last updated	Number of ratings	Rendering engine	Release date
Aldiko Book Reader	1.1.6	Aug 10, 2017	51	-	-
Apple Books	4.2.3	Jun 3, 2019	N/A	OS WebKit	N/A
Bluefire Reader	2.9	Jan 6, 2018	2.5K	-	-
CHMate	6.9.1	May 22, 2018	13	OS Webkit	N/A
Ebook Reader	4.0.9	Dec 22, 2017	345	OS Webkit	N/A
eBoox	1.60.1	Jul 18, 2019	114	-	-
EPUB Reader	5.1.55	Mar 14, 2017	647	-	-
FBReader	1.0.10	Aug 31, 2019	12	-	-
Gerty	1.1.5	Aug 8, 2015	65	OS Webkit	N/A
Kobo Books	9.14	Jun 11, 2019	8.4K	OS Webkit	N/A
Kybook 3	0.7.8	Feb 23, 2019	579	-	-
Marvin	3.1.2	Oct 11, 2017	239	OS Webkit	N/A
Play Books	5.3.0	Aug 26, 2019	8.3K	-	-
PocketBook	3.2	Aug 12, 2019	637	OS Webkit	N/A
Power Reader	6.10	Aug 28, 2017	4	OS Webkit	N/A
R2 Reader	2.0.1	Jun 21, 2019	5	OS Webkit	N/A
TotalReader	5.1.61	Jul 4, 2017	329	OS Webkit	N/A
YiBook	1.8.5	May 13, 2018	4	-	-
Yomu	2.3.0	Jul 17, 2019	59	OS Webkit	N/A

TABLE XIV
OMITTED EPUB READING SYSTEMS FOR IOS

Reading system	Version	Reason
Kindle	6.24	Unable to side-load EPUBs.
PureReader	1.4.2	Unable to open fully compliant EPUB.

TABLE XV
EVALUATED EPUB READING SYSTEMS FOR ANDROID

Reading system	Version	Number of downloads	Last updated	Rendering engine	Release date
4shared Reader	1.20.0	1M+	May 2019	-	-
Aldiko Book Reader	3.1.3	10M+	Oct 2018	-	-
Aldiko Classic	3.1.3	500K+	Oct 2018	-	-
AlReader	1.911805270	5M+	May 2018	-	-
Bookari Free	4.2.5	1M+	Feb 2018	-	-
Book Reader	1.12.12	1M+	Jun 2019	-	-
Cool Reader	3.2.32	10M+	Aug 2019	-	-
Ebook Reader	1.0	10K+	Aug 2019	-	-
Ebook Reader	5.0.8.2	5M+	Jun 2019	OS Blink	N/A
EBook Reader	3.5.0	5M+	Jul 2019	-	-
eBoox	2.22	1M+	Aug 2019	-	-
ePub Reader	2.1.2	1M+	May 2015	OS Blink	N/A
Epub reader	4.0	10K+	Apr 2019	-	-
Epub Reader	8.0.39	100K+	Feb 2019	-	-
EPUBReader	1.0.32	100K+	Nov 2014	OS Blink	N/A
eReader Prestigio	6.0.0.9	10M+	May 2019	-	-
FBReader	3.0.15	10M+	Jul 2019	-	-
Freda	4.31	10K+	Mar 2019	-	-
FullReader	4.1.4	1M+	Aug 2019	-	-
Gitden Reader	4.5.3	100K+	Jan 2018	OS Blink	N/A
Google Play Books	5.2.7	1B+	Aug 2019	-	-
Infinity Reader	1.7.57	5K+	May 2017	OS Blink	N/A
iReader	1.1.4	5K+	Aug 2019	OS Blink	N/A
Kindle	3.2.0.35	100M+	Jul 2019	-	-
Librera	8.1.242	10M+	Aug 2019	-	-
Lit Pub	3.5.3	100K+	Jun 2017	OS Blink	N/A
Lithium	0.21.1	1M+	Jan 2019	OS Blink	N/A
Moon+ Reader	5.1	10M+	Aug 2019	-	-
PocketBook	3.21	1M+	Aug 2019	OS Blink	N/A
Reader FB2	1.20	50K+	Jun 2019	-	-
ReadEra	19.07.28	5M+	Jun 2019	-	-
Reasily	1907d	100K+	Jun 2019	OS Blink	N/A
Solati Reader	2.5.1	10K+	Jun 2015	-	-
Supreader	3.2.30	1M+	Dec 2018	OS Blink	N/A
Tolino	4.10.2	100K+	Feb 2019	-	-

TABLE XVI
OMITTED EPUB READING SYSTEMS FOR ANDROID

Reading system	Reason
Adobe Digital Editions	Unable to open local EPUBs.
Cloudshelf	Unable to open local EPUBs.
Kobo	Unable to side-load EPUBs.
SKY Reader	Unable to open local EPUBs.

TABLE XVII
EVALUATED EPUB READING SYSTEMS FOR CHROME AND FIREFOX

Reading system name	Version	Store ID	Number of users
Chrome			
Ebook Reader for Google Drive	1.0.7	mfpbhmcmakfaeajfpehaoijecamlehpl	1,513
EPUBReader	2.0.8	jhhclmfgfllimlhabjkgkeebkbiadflb	137,917
EPUB READER	1.0.1	mbcgbbpomkkndfbpiepjimakkbocjgkh	816
ePUB Reader	0.1.1	dgkibcakfnhcnijakeobjifghganmojn	10,574
ePUB Reader	1.1.0	fnplkbhndemgbopkkpmpnfklkhphpneg	1,893
Firefox			
EPUBReader	2.0.9	-	158,480
ePUB Reader	0.1.1	-	5,097
ePUB Reader	0.0.1	-	36
myBook	0.4.0	-	434
QiuReader	0.1.5	-	1,512