A Security Study about Electron Applications and a Programming Methodology to Tame DOM Functionalities

Zihao Jin*†‡, Shuo Chen*§, Yang Chen*§, Haixin Duan†¶††, Jianjun Chen†¶** and Jianping Wu†‖
*Microsoft Research
†Tsinghua University
**Zhongguancun Laboratory
††Quancheng Laboratory
‡jinzihao1996@gmail.com
§{shuochen, yachen}@microsoft.com
¶{duanhx, jianjun}@tsinghua.edu.cn
‖jianping@cernet.edu.cn

***Abstract*—The Electron platform represents a paradigm to develop modern desktop apps using HTML and JavaScript. Microsoft Teams, Visual Studio Code and other flagship products are examples of Electron apps. This new paradigm inherits the security challenges in web programming into the desktop-app realm, thus opens a new way for local-machine exploitation. We conducted a security study about real-world Electron apps, and discovered many vulnerabilities that are now confirmed by the app vendors. The conventional wisdom is to view these bugs as *sanitization errors*. Accordingly, secure programming requires programmers to explicitly enumerate all kinds of unexpected inputs to sanitize. We believe that secure programming should focus on specifying programmers' intentions as opposed to their non-intentions. We introduce a concept called *DOM-tree type*, which expresses the set of DOM trees that an app expects to see during execution, so an exploit will be caught as a type violation. With insights into the HTML standard and the Chromium engine, we build the DOM-tree type mechanism into the Electron platform. The evaluations show that the methodology is practical, and it secures all vulnerable apps that we found in the study.**

I. Introduction

The Electron platform [1] represents a paradigm to develop modern desktop apps using HTML and JavaScript (HTML+JS) running on Chromium. It is becoming an industrial trend, as many companies' flagship applications have been written or re-written as Electron apps, such as Microsoft Teams, Visual Studio Code, WhatsApp, and Slack. Compared with desktop apps written in traditional languages, Electron apps have the advantage to utilize the power of Chromium to build rich features. In addition, an Electron-app codebase is very easy to be adapted to web and mobile platforms, saving significant development and maintenance costs when the app is expected to run across platforms.

However, from the security perspective, vulnerabilities due to the nature of HTML+JS have been a risky pitfall for decades. They manifest in different app categories, spanning websites, web apps (SaaS), mobile apps, browser extensions, etc. There is a rich body of literature about vulnerability studies focusing on these app categories. Compared with them, desktop apps are usually more sophisticated in their functionalities, targeting more substantial scenarios, such as enterprise communication, programmer productivity, business planning, etc., and impose different security and privacy requirements. Moreover, unlike in a website, a web app or a browser extension, security boundaries based on web origins (according to the same-origin-policy) are often not applicable to most code and data in a desktop app, which reads from local-input sources and exercises the local-machine privilege. Because of all these differences, a dedicated study about Electron apps is needed to understand the vulnerability sources and to develop an effective methodology for secure programming. We will show that our observations from the study are complementary to those obtained from the previous studies about other app categories (e.g., [32] [31]).

Our study and key insight. To get valuable insights into the characteristics of real-world Electron apps and investigate their potential vulnerabilities, it is important that our study has both depth and breadth, so we conducted the study in two rounds. The first round focused on 12 apps of which we could get source code access. By inspecting and testing the code, we found that 6 apps could be exploited to cause security and privacy consequences. For example, an attacker could fake a group conversation in *Microsoft Teams*; the Antares SQL client could get malicious script executed on the local machine when it operated a remote MySQL database; a user of the SSH client *GraSSHopper* who was enticed to log into an attacker's SSH server could result in an arbitrary JavaScript execution. Our second-round study was scaled out to 70 more apps using a semi-automatic approach. Despite the relatively light-weight investigation, we still found 13 vulnerable apps.

The essence of all these vulnerabilities can be described as follows. An app has its intended functionalities, which means that there is a (usually infinite) set of DOM trees within which all the intended mutations happen as the app runs. An attack happens when a DOM tree in the set mutates into one out of the set, and the latter has extra functionalities that none of the DOM trees in the set has. The goal of secure programming is to defend against such gain-of-function DOM-tree mutations.

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24305
www.ndss-symposium.org

Our proposed methodology. With this understanding, the question for secure programming becomes how to tame the DOM functionalities. Today, the intended functionalities (i.e., the set of intended DOM trees) only vaguely exist in the programmer's mind. Prevention of unintended mutations is done implicitly through writing code about user-input sanitization logic and other restriction logics, because the conventional wisdom is to view these vulnerabilities as "sanitization errors". This approach has been known to be error-prone. We emphasize that secure programming should focus on specifying programmers' intentions as opposed to enumerating their non-intentions. The methodology we propose in this paper is called *DOM-tree type-checking*. A new concept, namely *DOM-tree type*, is introduced to safeguard the program execution. It is inspired by the concept "types-from-data" [21] in the programming language community. The basic idea is to use sample values of an object to construct a type that captures the programmer's intention about the object, so that if the object mutates out of the intention during an actual execution, a type violation is raised. DOM-tree type is essentially the embodiment of the types-from-data concept for DOM-trees (i.e., the "object" being typed is a DOM-tree).

To enable this methodology, we build two modules in Electron. One is a tool named *TypeBuilder* to assist the programmer to build a DOM-tree type based on concrete test-runs of an app. The other is the type-checking mechanism built into the platform, which we call *TypeEnforcer*. The design requires a deep understanding about the internals of the Chromium engine. Specifically, we identify and intercept all the chokepoints in Chromium to ensure the type enforcement for all DOM-tree-mutating code paths. For all DOM elements not yet connected to the DOM tree, TypeEnforcer has the mechanism to defer their persistent effects until they are connected and checked against the DOM-tree type. This important mechanism was missed in the previous DOM-tree inspection techniques, which we could bypass.

Evaluations. We evaluated the practicality and effectiveness of the methodology using the vulnerable Electron apps we discovered. In every case, the programmer only needed to make a small adjustment (or no adjustment) to the type generated by *TypeBuilder*, which showed that the methodology was practical for normal programmers. In every case, *TypeEnforcer* caught the exploit successfully. Speedometer 2.0 benchmark [20] was used for performance evaluation, which showed negligible completion time increase under realistic user-action loads. Meanwhile, we reported the 19 vulnerable apps to their vendors, who have confirmed or fixed 13 of them.

II. Background

In this section, we give the necessary context for our study and proposed methodology. Section VIII will provide additional discussions about related research work.

A. Cross-site scripting and scriptless attacks

Cross-site scripting (XSS) is a terminology to describe many types of vulnerabilities due to the nature of HTML+JS. Originally, XSS referred to the consequence that an attacker's script runs in the context of a target site not owned by the attacker. Throughout the years, the notion of "cross-site" has become a misnomer because XSS does not necessarily involve two sites. For example, the attack payload can come from a database, a local text string, or any other means of input. Nowadays, "XSS" means broadly all types of input validation bugs that execute the attacker's data as scripts. There is another type of input validation bugs, usually called "scriptless attacks" [7]. In these exploits, the attacker's payloads become harmful non-script elements in the DOM trees to compromise the user's security and privacy. In this paper, we refer to both XSS and scriptless attacks as *payload injection bugs*.

B. Security studies about different app categories

Payload injection bugs have been known for decades. They are due to the inherent vagueness of the distinction between "data" and "code" in HTML+JS. Specifically, HTML by design allows elements to have dynamic behaviors, and JavaScript by design allows texts to be executed as code. It is unlikely that these bugs can be eradicated in an app-agnostic manner, unless HTML and JavaScript are fundamentally redefined. Therefore, deeper insights about characteristics of different app categories become crucial in the search for effective solutions.

For example, Jin et al. [32] conducted a study about Android apps running in the PhoneGap framework. The key insight from the study was that mobile apps had many more input channels than web apps, such as barcode, SMS, file system, contacts, NFC, etc. Accordingly, the authors proposed a static code analysis technique. For each input channel, the technique constructed a JavaScript program slice, and performed taint analysis to determine whether the input could flow into a sensitive API. Staicu et al. conducted a study about Node.js apps [31]. The key observations were that many injection attacks were due to `eval` and `exec` in the code, and that Node.js modules constituted a significant portion of code in the ecosystem. The authors developed a code-analysis technique to statically compute all possible strings that could flow to an `eval` or `exec` call in a module. The result was expressed in a *string template*, containing user-controlled portions as "holes". The technique then instrumented the call with a runtime check, which checks its argument value against the template.

Characteristics of Electron apps. The observations we made about Electron apps are complementary to the ones about other app categories. The differences exist in several aspects.

- It is very rare for Electron-app programmers to use `eval` to evaluate a non-constant string, as we will explain in Section VI. It suggests that this programmer community is very aware of the danger and there are good alternatives to avoid `eval` in Electron apps. Therefore, unlike in Staicu et al.'s work, we do not focus on analyzing script's content, but on the DOM tree. Specifically, our primary question regarding a `<script>` element is whether it is expected in a particular DOM-tree position. The question is agnostic to its script content.
- Although not a web app, an Electron app resembles Single-Page App (SPA) [33], in which the execution after initialization only updates certain UI elements on the initial HTML page with new data, but does not navigate to a whole new page. We will discuss more in Section VI. For this reason, reflected XSS, which is an XSS

category significant for websites and web apps, is unlikely for Electron apps. Also, the expected stable shape of the DOM tree underpins our idea of DOM-tree type.
- Taint analysis about unconventional input channels is effective for PhoneGap apps, but Electron apps take inputs from both unconventional and conventional channels, such as network messages, SQL queries, disk-file contents, etc, so it is unclear how to define taint rules with an appropriate granularity for a precise static analysis. This is compounded by the path explosion challenge because Electron apps are often more complex than mobile apps. Therefore, our methodology switches the focus from all kinds of input to their common outcome – the DOM tree, and from static bug-finding to runtime type enforcement.

III. Our investigation and discoveries

In this section, we first give an overview of Electron apps' security situation by showing that payload injection bugs are a significant challenge. We will show 6 vulnerable apps in our first-round case studies, followed by a summary of 13 more apps found vulnerable in the second round. The discoveries inspire our defense methodology discussed in Section IV.

A. Sanitization challenges

The security of Electron apps relies heavily on sanitization. However, sanitization turns out to be error-prone in the apps, for two reasons:

Difficulty of parsing custom sub-grammars. Parsing is a major source of complexity within a sanitizer. If parsing is flawed, a dangerous payload can potentially go undetected by getting itself misinterpreted. An app needs to take user input from many places to mutate DOM elements. In each place, the programmer is required to write a parser for a specific sub-grammar. This includes not only the sub-grammar of any subset of HTML, but also other languages like CSS, Markdown, shell commands, file paths, etc. The parser/sanitizer is not easy to define and build.

Difficulty of anticipating unsanitized paths. Even with a perfect sanitizer for each sub-grammar, an app still has the classical security challenge - path explosion. It is difficult for a programmer to anticipate paths that an attacker's data may flow through. Some data that need sanitization may go through a path without any sanitization logic. We will describe several vulnerabilities of this type, which allow the attacker's payload to be deployed via strings representing local files and hostnames. We suspect that the programmers do not even consider these strings as the "user input", so the data paths are completely unsanitized.

There is another type of bugs that we call "de-sanitization". They are attributed to cross-team misunderstandings about the app's own sanitization logic. For example, a sanitization done in one app module can be undone by another. A thorough cross-team understanding is always a challenge in software development, because every team's code logic may introduce special circumstances.

In the following subsections, we will analyze cases of incorrect and insufficient sanitizations. Today, eliminating these issues is entirely a responsibility on the programmers' shoulders. As shown by the examples below, the attack data and paths are often unexpected with respect to the app designs. It is an unfortunate security situation when programmers are required to expect all the unexpected.

B. Case study: Microsoft Teams

Microsoft Teams, abbreviated as Teams in this paper, is a business communication application. It is one of the flagship products of Microsoft. The product includes a desktop client, a web client and mobile clients for Android and iOS. The desktop client is now an Electron app. It is implemented as a thin layer (235 kLoC) wrapping around the web client (2170 kLoC). Our description below applies to the desktop and web clients. The mobile clients are still in native code, rather than Electron apps.

The main functionality of Microsoft Teams is instant messaging. On the client side, Teams provides a WYSIWYG editor to compose HTML-enabled messages. For example, bold text is converted to a `<b>` element. A link is converted to an `<a>` element. An image pasted into the editor is automatically uploaded to a backend server, and converted to an `<img>` element whose `src` attribute points to the uploaded image. The server forwards messages between clients. The server performs its sanitization logic similar to that on the client. This helps reduce the attack surface of the client, although the client's sanitization logic tries to be sound and complete on its own, not to rely on the server's logic.

1) Sanitizers: Teams has two different sanitizers for the server and the client. No library or code snippet is shared between the two, as they are written in different languages (C# for server, TypeScript for client).

Client-side sanitizer. The sanitizer of the client is based on *sanitize-html* [2]. Sanitize-html first uses *htmlparser2* [3] to parse an input string into HTML tokens (i.e., tags, attributes, comments, text, etc.), then applies a whitelist supplied by Teams to filter out dangerous tags (e.g., `<script>`, `<iframe>`) and attributes (e.g., `onload`, `onclick`). The sanitization logic for HTML elements undertakes many tasks crucial to user's security and privacy, such as:

- If the `src` attribute of an `<img>` (i.e., image) element points to a URL of a domain not whitelisted by Teams, the sanitizer rewrites the `src` to `undefined`. This is needed to prevent an attacker from using an `<img>` to leak user information (e.g., IP address) to the attacker's website.
- The sanitizer filters out special characters in `src` and `href` attributes to prevent a URL from "escaping" from the attribute and being evaluated as an HTML or Angular variable [4].
- The sanitizer upgrades a URL from `http` to `https` in the `src` and `href` attributes.
- The sanitizer rewrites the `rel` attribute of an `<a>` (i.e., anchor) element to ensure that the link target never receives the referrer URL.
- The sanitizer rewrites the `target` attribute of an `<a>` element to ensure that it always opens a new window, rather than navigates the main Teams window.
- Many other rewritings based on custom sub-grammars.

In addition to normal HTML elements, the client-side sanitization also handles important transformations regarding

CSS (Cascading Style Sheet). For example, the following are performed for the `style` attribute of every HTML element:

- Removing `url()` and `attr()` functions
- Removing `position: fixed` declarations
- Removing comments
- Filtering out CSS property names not whitelisted

Moreover, the sanitizer also deals with lexical complexity. There are many legitimate encodings. It requires much care to parse the user input based on the encoding rules, especially when it may be decoded into special characters. For example, curly braces (i.e., { and }) can be represented as the following strings. Missing any one of the strings may result in security consequences.

{	&#0*123;?	&#x0*7B;?	&lcub;	&lbrace;	\u007b
}	&#0*125;?	&#x0*7D;?	&rcub;	&rbrace;	\u007d

Server-side sanitizer. Although independently developed, the server-side sanitizer is conceptually similar to the client-side. It consists of the following steps:

1) Parsing input string into a stream of HTML tokens
2) Filtering HTML tags, attributes, classes and protocols
3) Filtering CSS in the `style` attribute
4) Removing Angular expression delimiters

As the server-side sanitizer is independently developed, its sanitization rules are slightly different from its client-side counterpart. For example, it does not remove comments in CSS, and does not enforce a whitelist of CSS property names.

2) Vulnerabilities discovered: We discover two vulnerabilities in Teams, exploitable by any user who can send a message to a group chat.

Vulnerability to allow user tracking. As discussed in Section III-B1, it is important to disallow an image to be loaded from an arbitrary website. Otherwise, an attacker can send a message containing a visible or invisible image to track other users in the chat, so that whenever the image is loaded or reloaded, the users' IP addresses are disclosed. The attacker can use this repeatedly to monitor other users' activity patterns and geographic locations over a long duration.

We study the client-side and server-side sanitizers, and construct the following string which, when sent to a group chat, can successfully carry out the attack. Portions of the string are highlighted and underlined to help our explanation.

```
<div style="width: 1px; test1:; background-
image: ur/*x*/l(https://evil.com/tracker.png);
test2:; height:1px;"> Hello! </div>
```

After sanitization, the string is transformed into the following HTML element, which loads an image from `evil.com`.

```
<div style="width: 1px; background-
image: url(https://evil.com/tracker.png);
height: 1px;"> Hello! </div>
```

The reason is because of the sanitization logic on the client-side and server-side, summarized as the following 3 steps of string transformation:

1) The server-side sanitizer rejects any illegitimate HTML content. If the value of `background-image` was "`ur/*x*/l(https://evil.com/tracker.png)`", it would be rejected. However, our string is legitimate because the *entire* underlined portion (from the semicolon to `test2:`) is enclosed inside the single quotes, thus parsed as the value of `test1`.
2) The client-side sanitizer then uses semicolon as the delimiter to scan through the string. Because `test1` and `test2` are unknown style properties, the two highlighted portions are removed.
3) The client-side sanitizer then removes "`/*x*/`" as a comment, turning "`ur/*x*/l`" into "`url`".

Vulnerability to allow fake chat messages. It is important for Teams to sanitize the `style` attribute of every chat message to ensure that the message's appearance does not mislead other users. For example, if a message has its `z-index` set to a value greater than 0, and its `position` property set to `fixed`, it can occupy the whole chat window and opaquely overlay on top of all other messages. Figure 1 shows a demo attack that exploits the vulnerability described next. Our fake messages appear to come from the company's CEO and CTO. The messages have full HTML functionality. Hyperlinks can be used to take users to dangerous websites, for example.

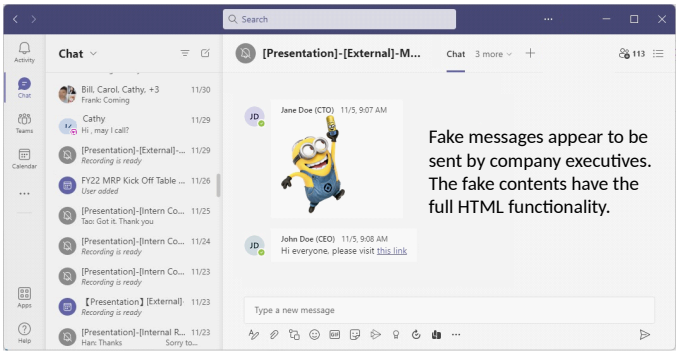

Fig. 1: Fake messages in a group chat

The chat message that can exploit the vulnerability is in the box below, in which "`CodeMirror-fullscreen`" is the name of a CSS class that sets `z-index` to 9, and sets `position` to `fixed`. We obtain this knowledge by searching the keyword "`z-index`" in the entire source code. The attack would be simple if the sanitizers did not prohibit the CSS-class assignment `class="CodeMirror-fullscreen"`. However, this obvious attack opportunity is blocked by the sanitizers, which remove "`class`" in all HTML elements in chat messages.

```
<span itemscope="" itemtype="http://schema.
skype.com/Reply" value="CodeMirror-fullscreen
"> Fake messages in HTML! </span>
```

In the source code, we find a message postprocessing module for certain types of interactive contents, including attachment, @-mention, quoted reply and emoji. It infers a message's item-type by searching for a list of schema URLs, such as `http://schema.skype.com/Reply`, and rewrites the message in different ways depending on the schema. To customize the style of the "quoted reply" item-type via CSS, this module assigns the string in its `value` attribute to its `class` attribute, resulting in the vulnerability.

In this vulnerability, the sanitizers do not make any mistake regarding HTML and CSS. The path is sanitized at first, but the input then triggers an unexpected postprocessing module after sanitization. For a complex app like Teams, understanding the semantics of all modules is challenging.

C. Case study: GraSSHopper

GraSSHopper is an SSH client with a rich set of features, such as multi-tab, remote file explorer, command history, etc. Figure 2 shows this app with two tabs open. They connect to two SSH servers, named "server" and "another-server", as user "root". Accordingly, the titles of the two tabs are "root@server", "root@another-server". Similar to tabs in many other apps, when the mouse hovers over a GraSSHopper tab, a popup appears to show the title of the tab in its entirety. In the terminal area, the user can select a piece of text. If the text represents a file path, a clickable popup appears to show the "cd" command to the directory. As shown in Figure 2, the popup shows "cd /etc/apache2". Note that Figure 2 overlays the two mouse positions on the same screenshot to save space for the paper.

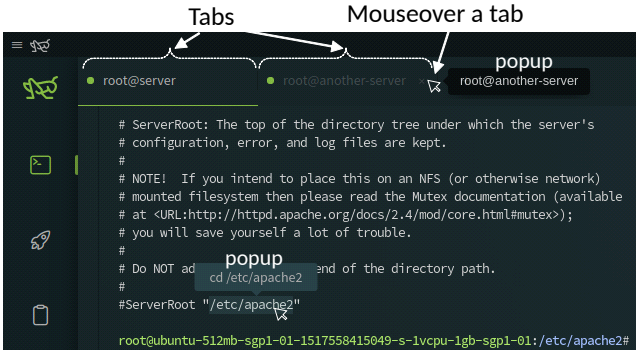

Fig. 2: GraSSHopper, with two tabs open

Next, we explain two vulnerabilities we find, which lead to the execution of an arbitrary script.

Script injection via selected text. The first vulnerability is exploited when a victim user is enticed by the attacker to select a block of text like the following.

```
<iframe/srcdoc=&lt;script&#47;nonce=&quot;fCRqK3
cHTuOfzyEnqua4UQ==&quot;&gt;alert(window.parent
.document.body.innerHTML)&lt;&#47;script&gt;>
```

The selected text is considered by GraSSHopper as a file path, because the app uses the following regular expression to determine whether a string is a file path. The regular expression matches a slash-delimited string that does not contain any space (the normal ASCII space), single quote or double quote.

```
^((\/[^/\"'\s]+)|([^/\"'\s]+\/))+([^/\"'\s]+)?$
```

Note that the selected text takes advantage of the flexibility of the HTML language to satisfy the regular expression, as it (1) uses a slash rather than a space to separate the tag name (`iframe`) and its first and only attribute (`srcdoc`), (2) skips the quotes that surround the attribute value of `srcdoc`, and (3) escapes the special characters (<, >, /, ", ') inside it. Since the text is taken as a file path, it is assigned to the innerHTML property of the clickable popup. At this point, it is interpreted as the following HTML content. The script `alert(...body.innerHTML)` is an idiom in many DOM-attacks that indicates the full control of a DOM document. In our scenario, when the text selection is made, we see this success indication.

```
<iframe srcdoc="<script nonce='
TuOfzyEnqua4UQ==''>alert(window.parent.
document.body.innerHTML)</script>">
```

The attack string can be contained in a large block of normal-looking text, which may use a Unicode whitespace to separate words. The victim is unlikely to feel suspicious when asked to select such a text block, which, when selected, still deploys the payload. Furthermore, the terminal supports RGB-color text, so the payload string can be made invisible.

We will explain the purpose of the script's nonce attribute after describing the second vulnerability below.

Script injection via hostname. The second vulnerability can be exploited when a user is enticed to copy the following "SSH connection string", paste it into the hostname box, and start an SSH connection. Obviously, there is no host on the Internet with this long and strange "hostname". GraSSHopper creates a tab for the attempt to connect to the "host". When the user moves the mouse over the tab, the popup appears, taking the "hostname" as a part of its `innerHTML` property, so the payload is deployed.

```
ssh.org:connection=.<iframe srcdoc="<script
nonce='fCRqK3cHTuOfzyEnqua4UQ==''>window.
alert(window.parent.document.body.innerHTML)
</script>">.nonexistent.com
```

Bypassing content security policy (CSP). GraSSHopper employs a CSP [8] that disallows `unsafe-inline`, preventing the execution of inline event handler (`<img src="x" onerror="...">`). The CSP limits `script-src` to a whitelist of nonce values, which means that only a script carrying a whitelisted nonce value can be executed by the app. The nonce mechanism is a common practice for a web app to selectively import a third-party script. For a web app, the nonce value list in CSP is dynamically generated by the server, and is refreshed every time a user visits the web page. Using the same mechanism for a Electron app is inherently invalid, as the CSP and its nonce values are hardcoded into the client app. We use one of the nonce values to bypass the CSP.

D. Case study: Visual Studio Code

Visual Studio Code, or VS Code, is an IDE built on Electron. Like Teams, it is also a product of Microsoft, but it does not need to process rich content like HTML or multimedia contents. Since text editing is the main UI functionality, the situations to build HTML strings from user contents are extremely limited. Thus, the app is carefully designed to only parse and accept a small subset of Markdown to produce HTML. This is also known as Markdown rendering. It uses Marked [5] for rendering, then passes the rendering result to its sanitizer, which is based on Insane [6], a configurable sanitizer similar to *sanitize-html* used by Microsoft Teams.

Although VS Code's Markdown is highly restrictive, it still allows the `<img>` element to be generated in the rendered HTML. We identify two situations where an attacker-controlled Markdown can be rendered as HTML that contains

an external image, as shown in Figure 3. The figure overlays these two situations on the same screenshot to save space for the paper. The consequence is like the Microsoft Teams user-tracking vulnerability in Section III-B2. The user's IP address is sent to the attacker's server when the mouse hovers over certain texts.

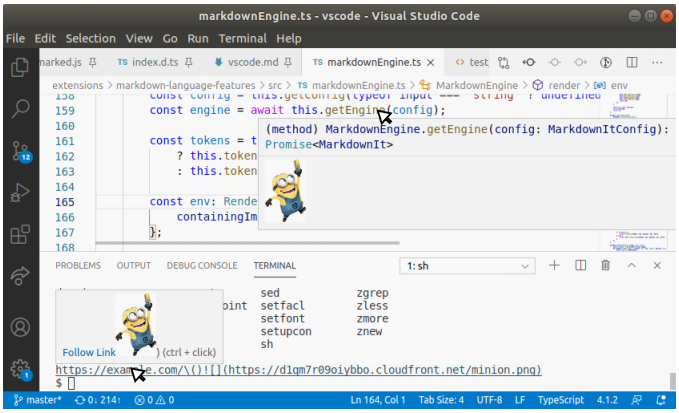

Fig. 3: Markdown rendering in Visual Studio Code

Vulnerability 1: extracted URL as Markdown. When the mouse hovers over a URL-like text in the terminal area (the lower portion in Figure 3), a popup appears, in which the string "`[Follow Link](...) (ctrl + click)`" is rendered as Markdown. The substring "..." is the URL extracted from the text.

The extraction is done by a state-machine-based parser. There are many details in the parser logic. A fact relevant to this vulnerability is that the following string is extracted as a URL in its entirety. When the parser scans through the string, it will exit when it encounters a ")" without a matching "(". The parser logic does not regard the backslash as an escape, so it takes the entire string, including what is beyond "\()".

```
https://example.com\()![](https://d1qm7r09oiybbo.
cloudfront.net/minion.png)
```

However, when this extracted "URL" is placed in the "..." position, the content of the popup becomes the following. When it is processed by VS Code's Markdown renderer, the backslash is treated as an escape, so the first ")" closes the first "(", causing the highlighted substring to become a valid Markdown image reference to be rendered as HTML. Since the terminal window can display the program's execution output, it is easy for the attacker to place a dangerous URL in it.

```
[Follow Link](https://example.com\()![](https:
//d1qm7r09oiybbo.cloudfront.net/minion.png))
(ctrl + click)
```

Vulnerability 2: code comment as Markdown. When editing source code of certain languages (e.g., TypeScript), VS Code shows a popup when the mouse hovers over a function or variable name, e.g., `getEngine` in Figure 3. The popup displays the code comment above the definition of the function or variable name. Because the popup renders in Markdown, an external image can be loaded into the popup, causing an IP address exposure. A scenario of the threat is when a third-party library contains image references in its function definitions. Every programmer who uses the library has the risk of IP address exposure and location tracking.

E. Apps with unsanitized paths

In previous subsections, we discussed vulnerabilities that demonstrate the challenges for sanitizers to exhaustively cover all potential exploit possibilities. Not surprisingly, we spent a great amount of effort analyzing the apps to find these bugs in sanitization logic, although this should not be a hurdle for the attacker community with dedicated efforts. However, when investigating three other apps – Antares, Homura and OhHai Browser, we relatively easily found data paths that did not go through any sanitization logic.

Antares. Antares is a SQL client used to browse and query data in a MySQL or PostgreSQL database. We find Antares shows database table names in HTML without sanitization. When the victim connects to a database where a table name is attacker-controlled, the attacker can execute arbitrary JavaScript with the same privilege as Antares' own code.

Homura. Homura is an RSS reader. An article in an RSS feed is usually a preview of a website article, which commonly includes HTML. However, Homura neither implements a sanitizer for RSS contents, nor isolate them using an iframe. As a result, if an attacker places malicious contents on a website which Homura is subscribed to, arbitrary JavaScript can be executed with the same privileges as Homura's own code, which includes local filesystem access.

OhHai Browser. OhHai Browser is a browser built on Electron, utilizing Electron's webview to render web pages in an isolated environment. Although the web pages are safely isolated, data flowing out of the webview into the browser UI still requires sanitization. We find the titles of in-history pages and bookmark items are rendered as HTML without sanitization. An attacker-controlled webpage can execute arbitrary JavaScript in the browser UI through a JavaScript payload in its title. As in-history pages are persistent and rendered every time the browser starts, the attacker's control is also persistent.

Second-round investigation. To broaden the understanding about real-world Electron apps, we conduct the second-round investigation, covering 70 apps crawled from Electron's official website using a semi-automatic approach. We build a modified version of Electron with a hook added to the HTML parser, and run every app on it. Whenever the app parses HTML, the hook function records the string being parsed. This helps us identify the inputs that can be used to inject raw HTML. Navigating through the app's functionalities is left as manual effort. With this light-weight approach, we confirm the 13 apps in Table I are vulnerable. They are all caused by the attacker input becoming extraneous DOM-tree elements or attributes. Note that the second-round investigation does not study any sanitizer logic, which would require significant effort like in the first round. As a result, the 13 vulnerable apps are all unsanitized-path cases.

IV. Our Proposed Methodology

From the traditional perspective, vulnerabilities like the ones in Section III are often put under the umbrella of "sanitization errors". Indeed, they are due to flawed sanitization logic, unsanitized path or post-sanitization change. Sanitization errors have been a focused problem in the web security community for over two decades. Not surprisingly, Electron-app programming inherits the problem. In fact, the sanitization

App	Injection point
Jukeboks	filename
Poddycast	podcast title
Tess	filename
WAIL	MIME type
Advanced REST Client	HTTP header
Altair	error message
Another Redis Desktop Manager	file path
Appium Desktop	error message
Blankup	markdown
Blockbench	filename
Boost Note	markdown
DeckMaster	opened file
ElectroCRUD	database records

TABLE I: Confirmed vulnerable apps in the second-round investigation

logic for a real-world Electron app, e.g., Microsoft Teams or VS Code, is often much more complex than that for a typical web application. It consists of many steps of string transformations, using string substitution, regular expression, state-machine of characters or HTML/CSS tokens, and sub-grammar parsing.

The fundamental reason why eliminating sanitization errors is hard is because it requires a programmer to anticipate the unexpected, i.e., to enumerate all strange input data that are against the programmer's intention. We believe that a secure programming methodology should only rely on a programmer to correctly express his/her intention, not the negation of it.

Intuition behind our methodology. The methodology we propose is not focused on sanitization, but on the DOM-tree mutations during an app execution. Intuitively, in every case in Section III, we see that the attacker causes a DOM-tree to mutate to a form that can do something extra, beyond the programmer's intention. Therefore, our methodology aims to achieve two goals: (1) to enable the programmer to express the intended DOM-trees, (2) at runtime, to prohibit every mutation resulting in an unintended DOM-tree.

Our inspiration initially comes from Trusted Types [9], a relatively new browser mechanism to help prevent unsanitized texts to be assigned to some well-known XSS-injection "sinks". We will give more details about Trusted Types in the related work section. It is very different from our methodology, but at the conceptual level, it reminds us that "type" is a mechanism to express the intention about data and objects, and that it can be applied to HTML-based security.

Our inspiration also comes from a technology called "types-from-data" [21], which attracts much attention from the programming language community. The goal is to base on *sample data* in structured formats (e.g., XML, JSON, etc.) to build static types for programming languages (e.g., F#). During the execution, the static types are useful to tame the *actual data* to be processed by the program. Data cannot turn into an unintended object without being caught as a type violation.

Our goal of catching an unintended DOM-tree mutation is similar. A DOM-tree is in HTML, a structured format. If we can define a notion of *DOM-tree type*, and enhance the Electron platform to help programmers build the static type from test-runs of an app, then the platform will ensure that every exploit in Section III becomes a type violation.

A. Architecture and programmer's workflow

The architecture of our enhanced Electron platform and the programmer's workflow are shown in Figure 4. TypeBuilder is a dev-utility to help the programmer build the DOM-tree type. TypeEnforcer uses the DOM-tree type to safeguard the actual run of the app, and turn any unintended mutation into a type violation. A shared component called DOM Interceptor is responsible for intercepting all DOM-tree change events. It consists of a number of hook functions defined in the Blink rendering engine of Chromium. It is important to note that the DOM-tree type generated by TypeBuilder must be reviewed by the programmer. It is a part of the app code to release.

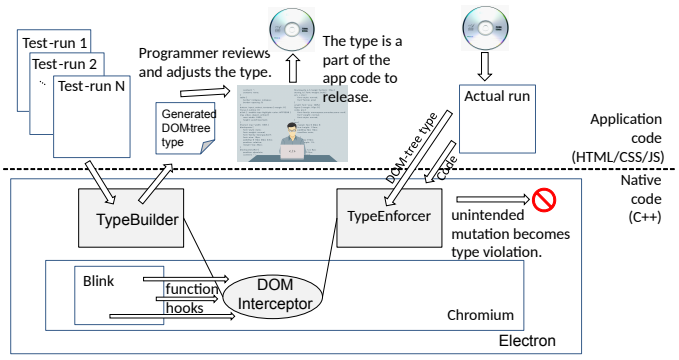

Fig. 4: Architecture and workflow overview

In theory, the programmer can handwrite a DOM-tree type for a simple app. However, for a real-world app, it can be difficult to enumerate all variations of the DOM tree. TypeBuilder is very helpful in this situation. After TypeBuilder is turned on, the programmer tests the app by going through its features as thoroughly as possible. During the test run, TypeBuilder monitors the DOM tree to construct a DOM-tree type. The process can also be split into multiple test-runs, each covering a subset of the app's features.

The testing process ends when the DOM-tree type *converges*, i.e., when the DOM-tree type no longer changes upon more test runs. For some apps, the DOM-tree type naturally converges after a number of test runs. However, some app features require manual adjustments to help the convergence. In Section IV-D, we will describe these mechanisms, namely subtree-flattening and attribute-value-wildcarding, which enables the programmer to generalize the learned DOM-tree type. In Section VII, we will show that, even when this programmer effort is needed, it is usually small.

B. Definitions

In this subsection, we give the definition of DOM-tree type, and explain how it is constructed from the mutations of an app's DOM tree. Specifically, the next two subsections explain how a DOM-tree type evolves its generality during test-runs to represent more variations of the DOM tree. The explanation include how TypeBuilder works, and how the programmer can discretionally adjust the DOM-tree type. We will then explain how DOM tree mutations are constrained by a DOM-tree type, i.e., how TypeEnforcer works. We provide a precise definition in Backus-Naur Form (BNF) in Appendix B.

Similar to a DOM tree, a DOM-tree type also has a tree structure. To avoid confusion, we refer to an element in the

DOM-tree type as a *shadow element*, and an element in the actual DOM tree as an *actual element*. A shadow element has four data fields:

1) Children: Like an actual element, a shadow element can have children. However, the children of a shadow element is a set rather than a list, i.e., they are unordered and deduplicated.

Text nodes. We do not include text nodes, i.e., plaintext content within an element, in the DOM-tree type. They are purely for display purpose, therefore can never become a payload-injection sink. The only two exceptions are text within a `script` element, and text within a `style` element. For a `script` element, according to HTML standard [12], its script-text only gets executed once during page loading. After that, if the script-text is changed, or a new `script` element is injected to the DOM tree, it has no effect at all. Since the initial DOM tree of an app is constructed locally, an attacker has no chance to inject a `script` element during page loading.

Actual DOM tree	DOM-tree type

`<div id="sidebar">`
`<ul>`
`<li>Chapter 1</li>`
`<li>Chapter 2</li>`
`</ul>`
`</div>`
`<div id="content">`
`<ul>`
`<li>Apple</li>`
`<li>Banana</li>`
`</ul>`
`</div>`
	`<div id="sidebar">`
	`<ul>`
	`<li></li>`
	`</ul>`
	`</div>`
	`<div id="content">`
	`<ul>`
	`<li></li>`
	`</ul>`
	`</div>`

TABLE II: Element deduplication

2) Identifier: Every shadow element has an identifier used for deduplication, which consists of a tag name (e.g., `div`, `form`, `a`) and the `id` attribute (empty value allowed). The example in Table II shows two `<li>` elements under "sidebar" represented by a single shadow element. If more chapters are added to the sidebar later, the DOM-tree type remains unchanged. The same applies to the `<li>` elements under "content". However, the two `<div>` elements ("sidebar" and "content") are represented by different shadow elements as they differ in the `id` attribute.

3) Attributes: Like an actual element, a shadow element has a map (i.e., a key-value store) of attributes. However, in a shadow element, each attribute has a set of values rather than a single value. By default, a shadow attribute contains a set of strings, with special rules applied to URL and script attributes.

For a URL attribute (defined by Blink's `IsURLAttribute` function [10], e.g., `src`, `href`), its shadow becomes a set of origins. In Table III, the two `img` elements with the same identifier are represented by one shadow element. However, since the first `img` element has the origin `https://foo.com` in its `src` attribute, and the second one has a different origin `file:`, the shadow attribute is a set containing two values.

For a script attribute (defined by Trusted Types [11], e.g., `onclick`, `onload`), its shadow becomes a set of JavaScript token sequences. The example in Table III shows two `li` elements with `onclick` attribute. As both attributes contain the same sequence of JavaScript tokens – an identifier, a left parenthesis, a number, and a right parenthesis, their shadow attribute contains only one value. This shadow attribute can also match `switchTo(3)`, but a script-injection payload such as `switchTo(1);alert(99)` will be a non-match, because the highlighted portion is not a "`Num`", but a token sequence "`Num RParen Semicolon Ident LParen Num`".

Actual DOM tree	DOM-tree type

`<img src="https://foo.com/hello.jpg">`
`<img src="img/hello.jpg">`
`<li onclick="switchTo(1)">Chapter 1</li>`
`<li onclick="switchTo(2)">Chapter 2</li>`
	`<img src="file: | https://foo.com">`
	`<li onclick="Ident LParen Num RParen">`
	`</li>`

TABLE III: Handling URL and script attributes

4) Style properties: A map of style properties is also included in a shadow element. Slightly different from an attribute value, a style property value has a type, and some properties accept multiple types of values. Therefore, for each style property, we maintain a set for string values, a set of origins for URL values, and a range (i.e., a `min` and a `max`) for numeric values.

Note that an element's style properties are not equivalent to its `style` attribute. For each element, Blink maintains an internal data structure holding its style properties, which cannot be directly set by the programmer. Rather, it is the result of style calculation using a complex "cascading" algorithm, whose inputs include global styles (.css files, `<style>` elements), local styles (the element's own `style` attribute), other attributes (e.g., `id`, `class`) and the element's parent styles, etc. Recall the fake chat message vulnerability in Section III-B2, although the attacker cannot directly control the `style` attribute of any element, he can still introduce a style property (`z-index = 9`) from a global stylesheet. In other words, a sound defense mechanism should not prohibit "`z-index = 9`" from being a legitimate global style property, but should only detect a violation when it is attached to a specific element. Therefore, we include the style properties, which are always attached to individual elements, in the DOM-tree type.

Layout-dependent properties. Among all 367 style properties, we exclude 29 properties from the DOM-tree type, which are categorized by Blink as *layout-dependent properties* [13], listed in Appendix A. They include `width`, `height`, `margin-left`, but not `z-index` and `position` [13]. These property values are expected to change with window resizing, and differ across machines with different screen resolutions. We consider the layout changes as expected app behaviors, so the DOM-tree type excludes them.

C. TypeBuilder

Explained in the previous subsection, a DOM-tree type represents a set of intended DOM trees. For example, in Table II, one `li` element in the DOM-tree type represents a set of `li` elements at the position in the actual DOM tree. The `li` elements can contain arbitrary content, but must not have any extra attribute or child element. Similarly, in Table III, the `img` element in the DOM-tree type represents a set of `img` elements at the position in the actual DOM tree, as long as their images are loaded from `file:` (a local file) or `https://foo.com`.

Of course, witnessing the DOM tree only once is insufficient for TypeBuilder to generalize all intended DOM trees of an app. Some variations of the DOM tree can only be triggered when the programmer tests certain features. TypeBuilder works by monitoring changes to the DOM tree, and extending the DOM-tree type to represent previously unseen variations.

Table IV shows how TypeBuilder updates the DOM-tree type when an element, i.e., the highlighted `div` element ("cat") on the left column, is inserted to the DOM tree. Before it is inserted, there is a pre-existing `div` element ("dog") which similarly contains an `img` and a `h1` element. In this situation, TypeBuilder only adds the previously unseen parts to the DOM-tree type, i.e., a `p` element, an `onclick` attribute of the `img` element, and a new value for its `src` attribute.

Actual DOM tree	DOM-tree type

`<div>`
` <img src="dog.jpg">`
` <h1>Dog</h1>`
`</div>`
`<div>`
` <img src="https://foo.`
` com/cat.jpg" onclick="`
` meow()">`
` <h1>Cat</h1>`
` <p>The cat is a ...</p>`
`</div>`
	`<div>`
	` <img src="file: |`
	` https://foo.com"`
	` onclick="Ident LParen`
	` RParen">`
	` <h1></h1>`
	` <p></p>`
	`</div>`

`<div>`
` <img src="cow.jpg">`
` onclick="moo()">`
`</div>`	

TABLE IV: Element insertion

Now suppose a third `div` element ("cow") is inserted to the DOM tree. Its `img` element has a `src` attribute with a `file:` origin (since it is a local filename), which matches the first `div`. Moreover, it has an `onclick` attribute that matches the second `div`. In this situation, TypeBuilder does not add anything to the DOM-tree type. In other words, TypeBuilder does not keep a set of whole subtrees it observes during test-runs to match every incoming subtree against the set. Rather, it breaks down them into individual elements and attributes, and merges them into the DOM-tree type. This generalizes unseen variations of the DOM tree, as they match any recombination of the subtrees seen during the development time. In this sense, a DOM-tree type can be thought of as an HTML "template". There are many template engines for JavaScript to generate HTML pages [34]. The task of TypeBuilder is an inverse of a template engine, because it abstracts from concrete HTML pages to a "template".

Besides element insertion, we will explain in Section V that there are other types of DOM-tree changes TypeBuilder observes, including: (1) element replacement, (2) element removal, (3) attribute modification and (4) style recalculation. The basic rule is, TypeBuilder only adds items to a DOM-tree type, but never removes them. When an element is replaced with another element, TypeBuilder simply treats it as an element-insertion event, i.e., merging the new element into the DOM-tree type, while keeping the shadow of the old element. TypeBuilder does nothing when an element is removed. For attribute modification and style recalculation, TypeBuilder adds the new value to the shadow attribute or style property, while keeping the old value in the set.

D. Programmer's adjustments

We provide two mechanisms to generalize a DOM-tree type, which are manually applied by the programmer.

Attribute-value-wildcarding allows the programmer to use wildcard character "?" and "*" to match any character or string in an attribute value. They are needed mainly for attributes containing random or incremental IDs (e.g., `<span id="item*">`). Since there is no reliable way to infer from a limited number of test-runs how these values are generated, the programmer should annotate them using wildcards. In Appendix B, BNF rule 19 specifies that the use of wildcard characters is limited to `WildcardString`, which is in turn limited to attribute values (rule 11), style property values (rule 15), or an `Origin` (rule 18).

Subtree-flattening is needed to handle a common pattern, which we call structure-agnostic subtree. In apps such as article readers and markdown editors, the user's rich-format content is usually displayed in a content area, which is a dedicated subtree. It is often inconvenient or impractical to specify all legitimate structures. In this case, the programmer can choose to *flatten* the subtree, so that it is treated as a one-layer structure. Today, sanitizers handle the same situation by filtering according to a whitelist of allowed tag names, attributes, URL origins, etc., such as the chat-message sanitizer in Teams. A flattened subtree is equivalent to the whitelist. The example in Table V shows that, if the `div` element is marked as flattened, any combination and nesting level of `h1`, `h2`, `u` and `i` element is considered legitimate. In Appendix B, rule 1 specifies that an `Element`'s child can be either `Elements`, or a `StructureAgnosticSubtree` which is a flat list (rule 3).

Actual DOM tree	DOM-tree type

`<div id="editor">`
` <h1>`
` <u>Title</u>`
` </h1>`
` <h2>`
` <i>Chapter 1</i>`
` </h2>`
`</div>`
	`<div id="editor"`
	` flatten="true">`
	` <h1></h1>`
	` <u></u>`
	` <h2></h2>`
	` <i></i>`
	`</div>`

TABLE V: Subtree flattening

E. TypeEnforcer

The way TypeEnforcer works is very similar to TypeBuilder, because it also monitors DOM-tree changes and tries to locate corresponding elements and attributes in the DOM-tree type. The only difference is that, when TypeEnforcer detects a missing element, a missing attribute, or a missing value for an attribute, it does not add the missing part to the DOM-tree type. Instead, it rejects the DOM-tree change, and raises an exception. In the example in Table IV, when the highlighted `div` element and its subtree are inserted during an actual execution, TypeEnforcer will forbid the insertions of the `<img>`'s `src` and `onclick` attribute and the `<p>`, leaving a `<div>` containing an `<img>` with no attributes and a `<h1>`.

Note that different from TypeBuilder, a key requirement from TypeEnforcer is the ability to *intercept* DOM-tree changes, which is more than passive monitoring. On some occasions, a DOM-tree change can have side effects happening before the type-checking is done. Intercepting them

requires modifications to Blink. The detailed discussion will be provided in the next section.

V. Realizing the concept in Electron

As discussed previously, our mechanism requires the capability to fully monitor and intercept DOM-tree changes (including style property changes). We need to inspect *every* DOM-tree change to decide if it should be permitted or rejected. And if we reject the change, it should be fully rolled back, without causing persistent effect.

To identify a complete set of chokepoint methods of DOM-tree changes, we carefully studied the source code of Blink, focusing on basic classes defining HTML nodes. Figure 5 shows the C++ class hierarchy. `Node` is the base class for all objects in the DOM tree. Its `SetComputedStyle` method is the sole interface to update the node's style after a style recalculation. `ContainerNode` defines a node that may have children, i.e., a non-leaf node. Methods `InsertBefore`, `AppendChild` and `ReplaceChild` are the chokepoints of all node-insertion events. `Element` inherits from `ContainerNode` and holds the set of attributes of the element. Method `WillModifyAttribute` is invoked before every attribute-modification event.

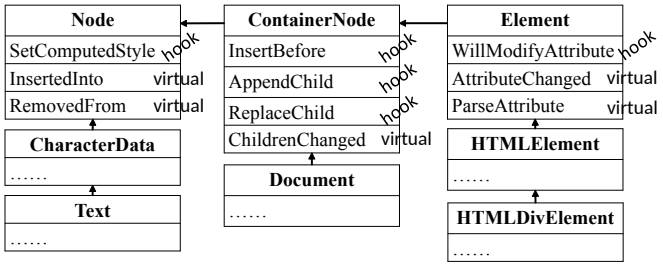

Fig. 5: Class hierarchy of DOM-tree nodes and our hooks

The DOM Interceptor is implemented by hooking on these five methods, and sending the observed DOM-change events to the algorithms discussed in Section IV-C and IV-E.

A. DOM-tree change completion vs. persistent effects

The description above is conceptually simple, but it hides a major source of complexity, as it implicitly assumes that no persistent effect can be done before the DOM-tree change is complete and available for TypeEnforcer to inspect. We thoroughly study the HTML standard and find this assumption invalid. For this reason, the MutationObserver API [14], which may notify a listener function after the DOM tree is modified, is not suitable. More in-depth work is needed inside Blink.

The important concepts involved in this complexity are DOM tree, Document and persistent effect. According to the DOM standard [15], an element always belongs to a Document, but is not always connected to a DOM tree. A Document provides a set of APIs that can cause persistent effects on behalf of the Document, such as accessing the filesystem, making a web request, registering event handlers, etc. The only way for an element to cause a persistent effect is to ask its containing Document to make such an API call.

Next, we explain the complexity and our solution. The complexity may seem to originate from some obscure corner-case details, but not solving it would make TypeEnforcer fail to protect every vulnerable app in Section III, because it is an easy path for exploitation, once it is known by the attackers.

B. Disconnected elements

When an element is not connected to the DOM tree, its state is set to "*disconnected*". In fact, every element is always created in the "disconnected" state, no matter if it is created explicitly by JavaScript through `createElement`, or implicitly by the HTML parser. Later, it can be connected to a DOM tree through methods like `appendChild`, `insertBefore`, etc.

It is worth noting that, in some situations, disconnected elements can also form a separate tree. We call it a disconnected tree because its root node is not a Document. One of the situations is when element `foo`'s `innerHTML` property is changed. Blink parses the new `innerHTML` value into such a separate tree. When the parsing is complete, the separate tree is connected to the DOM tree as the subtree under element `foo`. Only at this point is every element set to the "connected" state, because it is now in a tree of which the root is a Document.

C. Persistent effects triggered by element changes

Regardless of whether it is connected to a DOM tree, an element can be changed in the following ways: (1) inserting or removing a child element, (2) being inserted or removed from a parent element, (3) modifying an attribute. These changes may trigger persistent effects. For example, when the `src` attribute of an `<img>` element is set to a new value, it triggers a network request *immediately*, which may further trigger its `onload` or `onerror` event handler when the request is finished.

D. Solution: deferring persistent effects of disconnected elements

The essence of our approach is that we use the DOM-tree type to provide the context for the decision-making about an element. For example, when we see an element `<img src="http://foo.com">`, the decision is not whether `<img>` elements are allowed in the DOM tree, or whether they are allowed to load from `foo.com`. Instead, it is about whether the *particular* `<img>` element at *this DOM-tree position* is allowed to exist and load from `foo.com`. In other words, the decision can only be made when an element is connected to the tree.

Because disconnected elements can sometimes trigger persistent effects, our solution is to defer the effects until the elements are connected to the DOM tree. The deferral does not limit apps' functionalities. There are three situations worth discussing. First, when disconnected elements are implicitly created via innerHTML parsing, since the effects, e.g., file/network accesses, are asynchronous, an app programmer cannot assume they are completed by the time the elements are connected to the DOM tree. Second, consider an app programmer explicitly creates a disconnected element, and waits for its persistent effects to complete before connecting it to the DOM tree. We have not seen this need in reality, but can only imagine one meaningful scenario – the programmer wants to hide the loading of an element for visual smoothness. For this purpose, a classical and better approach is to use the `visibility` or `display` CSS property on the connected element. Third, if an app programmer explicitly creates a disconnected element that is never connected to the DOM tree, the only purpose

is to cause a persistent effect like file/network access. This is an unreasonable scenario that we have never seen in any app, because the effect can be simply fulfilled by JavaScript without using a disconnected element.

Detailed study. The hierarchy in Figure 5 shows that Blink defines five virtual methods in these basic classes. Every derived class can override them to implement the actions to take when they are invoked. For example, the `<img>` class overrides `ParseAttribute`, in which a network request may be made when its `src` attribute is changed. Also, the `HTMLElement` class overrides `ParseAttribute`, in which it registers event handlers that are common to all HTML elements (e.g., `onclick`, `onfocus`).

There are 121 descendant classes of `HTMLElement`. We study the source code of them. A total number of 53 classes override at least one of the five virtual methods, which are the places where persistent effects may be triggered. The following is a complete list of persistent effects that can be triggered by a disconnected element when certain attributes are modified:

- Disconnected `<body>`, `<input>`, `<iframe>`, `<portal>` elements can register event handlers.
- Disconnected `<img>`, `<video>`, `<audio>`, `<source>`, `<track>` elements can request file/network resources.
- Disconnected `<a>` elements may cause DNS prefetch.

In addition, disconnected `<img>`, `<source>`, `<track>` elements can request file/network resources when they are inserted under another disconnected element. For example, when a `<track>` is inserted under a `<video>`, a network request according to its `src` attribute is made. Element removal can also cause a persistent effect. Suppose a `<picture>` element contains multiple `<source>` elements. Removing one `<source>` element may trigger Blink's algorithm to select one of the remaining `<source>` elements as the effective one, and make a network request accordingly.

VI. Limitations

Since the objective of our defense mechanism is to prevent unintended DOM-tree mutations, it cannot prevent attacks that do not need to change DOM trees. Specifically, vulnerabilities due to the following browser functionalities are not prevented.

- JavaScript `eval(user_str)`. If an app directly passes a user input string to `eval()`, the attacker can execute arbitrary JavaScript without modifying the DOM tree.
- Script-initiated top-level navigation. Since Electron is essentially a Chromium browser, if the top-level document contains scripts like `window.open(user_str)` or `location.href=user_str`, the attacker can navigate the app away to a page containing malicious scripts. If `user_str` is "`javascript:attack_payload`", the payload will be executed in the top-level document.

We do not consider those as significant limitations. The web community is well aware of eval's security risk, and strongly discourages its use. We surveyed eval usages in Electron apps by adding a hook to the runtime JavaScript compiler in Electron, which not only covers eval, but also eval-like APIs that interpret strings as JavaScript code, e.g., setTimeout [16]. Among all 76 Electron apps studied in Section III, we find only 13 apps using eval. We inspected their source code and confirmed that, except for a calculator app using eval to evaluate mathematical expressions and a plugin system implemented using eval, every other eval occurrence takes a string constant as input, which has no security concern.

Regarding the threat of top-level navigation, the Electron community is well aware of it, and understands that there is no reason for an app to navigate itself away. Usually, when an app needs to display an external content, it either contains the external content in an iframe, or opens a regular browser to visit the URL, which is facilitated by Electron's built-in "will-navigate" mechanism [17]. It allows the app to register a listener for window navigation events. Whenever it sees a navigation, it checks the target URL to decide if the navigation should be permitted, or redirected to an external browser instead.

VII. Implementation, evaluation and disclosure

We implement TypeBuilder, TypeEnforcer, DOM Interceptor and the Blink patches on Electron 12.0.0, which is based on Chromium 89. The implementation is primarily in C++ code. We also write TypeScript code to expose an interface to Electron apps, which adds the following methods to Electron's webFrameMain [18] module:

- `SetDOMTreeType` loads a DOM-tree type from an HTML string.
- `SetTypeEnforcerMode` switches our module between builder and enforcer mode.
- `OutputDOMTreeType` serializes the current DOM-tree type into an HTML string.

In the C++ part, we reuse a few high-level functionalities of Blink. In our implementation, a DOM-tree type itself is a DOM tree and consists of HTML elements. Since a standard DOM tree does not support multiple values in an attribute, we join them by "|" and store it as a single attribute. To store an element's style properties, which are dynamically computed, we serialize and store them as attributes. For example, an element's `background-image` property is stored as `dtt-s-background-image` attribute. We also reuse the JavaScript tokenizer [19] from V8 JavaScript engine to parse script attributes into tokens. Blink's HTML parser and serializer are also used to load and save a DOM-tree type.

Reusing these modules not only reduces the engineering effort, but more importantly, it eliminates the possibilities of parsing inconsistencies described in Section III. Since we rely on Blink to parse the languages of HTML, CSS and JavaScript, and maintain the data structure for the DOM-tree type, there is no need for us to implement any parser of our own. It is a big advantage that we only need to interface with the whole grammars of HTML, CSS and JavaScript, with no concern about application-specific sub-grammars, such as "the longest URL-like string prefix that contains matching parentheses" in VS Code, or "key-value pairs delimited by semicolons that are not enclosed inside single or double quotes" in Teams.

Next, we present the evaluation results about validity, security and performance.

A. Validity and security

The proposed methodology is tested on 18 apps in Table VI, including VS Code, GraSSHopper, Antares, Homura, OhHai Browser, and all 13 vulnerable apps in the second round. Microsoft Teams is not included, because it runs on a variant of Electron that is proprietary. To show the methodology's practicality for a complex app with an open extension ecosystem, we apply it to 6 VS Code extensions in Table VII.

Name	Programmer's manual adjustment		Secured?
VS Code	Attr.	`id = list_id_*`	✓
GraSS-Hopper	Attr.	`id = sizzle*`	✓
	Style	`content = *`	
Antares	Attr.	`id = id_* | editor-*`	✓
	Style	`content = *`	
	Style	`background-image = *`	
Homura	Attr.	`<a> href = *://*`	✓
	Attr.	`<img> src = *://*`	
	Style	`<main dtt-flatten>`	
OhHai Browser	Attr.	`<webview> id = wv_*`	✓
	Attr.	`<webview> src = *://*`	
	Attr.	`<img> src = *://*`	
Jukeboks	none		✓
Poddycast	none		✓
Tess	none		✓
WAIL	none		✓
Advanced REST Client	Attr.	`id = anypoint-input-label-* | anypointAutocompleteInput* | anypointlistbox-*`	✓
Altair	Attr.	`id = cdk-overlay-*|nb-option-*`	✓
	Style	`text-shadow = *`	
	Style	`border-bottom-left-radius = *`	
Another Redis Desktop Manager	Attr.	`id = el-popover-*|el-tooltip-*|dropdown-menu-*|treeId*|el-autocomplete-*|tab-*|pane-*`	✓
Appium Desktop	Style	`box-shadow = *`	✓
Blankup	Attr.	`<a> href = http://*|https://*`	✓
	Attr.	`<img> src = http://*|https://*`	
	Flat.	`<div id="editorContainer" dtt-flatten>`	
Blockbench	Attr.	`<li> id = *`	✓
Boost Note	Attr.	`id = tree-*|portal-anchor-*|react-select-*|user-content-*|backlink__*|search-recently-visited-*|topbar__breadcrumb__*|sidebar__search__*|context__menu-*`	✓
DeckMaster	Style	`animation-duration = *`	✓
	Style	`transition-duration = *`	
Electro-CRUD	Attr.	`id = cdk-overlay-*|nb-option-*`	✓

TABLE VI: Evaluation on Electron apps

1) Electron apps: For every app on the list, we first run it with our enhanced Electron under the *builder* mode, test its features as thoroughly as we can to build a DOM-tree type. We monitor the DOM-tree type during testing. For some apps, the DOM-tree type converges by itself, i.e., even with continued testing, no new elements, attributes or style properties are added. When we encounter a "non-convergence" situation, we examine the part of the DOM-tree type that keeps growing, either add an attribute wildcard or flatten the subtree, then continue testing until it converges. Table VI shows the adjustments needed for each app. The effort is small.

We rerun the tests under the *enforcer* mode, which confirms that no normal functionality is affected by TypeEnforcer. Then, we repeat the attacks in Section III. They are all thwarted by TypeEnforcer. For each attack, we examine the violating mutation, which confirms that the design and implementation, as well as our understanding about the DOM tree in the attack situation, are correct. We list the DOM-tree type violation resulting from each attack in Appendix C. For example, the GraSSHopper attack via text selection would add an extra `iframe` element to the location (`/HTML/BODY/DIV[@id="container"]/...`) in the DOM tree, which would be caught by TypeEnforcer.

2) VS Code extensions: VS Code is an example of an extensible app. As a more diligent evaluation of our methodology, we incrementally extend the DOM-tree type for the extensions in Table VII, which include a "dummy" extension, three syntax-highlighters for Golang, TypeScript and Mojom, a to-do list and a FTP client. For each extension, we record the number of elements, attributes and style properties added to the DOM-tree type, and the manual adjustments needed to make the DOM-tree type converge.

While the dummy extension adds no visible functionality, the numbers in all columns are the highest among these extensions. Also, it is the only extension requiring a manual adjustment. Under the hood, the dummy extension contains the infrastructure common for all extensions, e.g., installation confirmation/progress, an info page, an "installed extension" entry, etc. Beyond the commonality, the variation of the DOM-tree type for each of the other five extensions is small.

As explained in Section IV-C, TypeBuilder iteratively performs a union operation to incorporate every newly witnessed type into the existing type. Installation of an extension also performs such a union operation. For this reason, although the app can install an arbitrary combination of extensions, the complexity of the DOM-tree type only grows linearly with the number of installed extensions.

Name	Manual adjustment		Elem.	Attr.	Prop.
Dummy	Attr.	`<div> id = *`	32	3	94
Golang	none		15	0	84
TypeScript	none		2	0	12
Mojom	none		0	0	0
Todo Tree	none		25	2	18
FTP Simple	none		16	0	89

TABLE VII: Evaluation on VS Code extensions

B. Performance

The performance evaluation uses the Speedometer 2.0 benchmark [20], which includes 16 different implementations of a sample app (a to-do-list app), and 3 types of user actions in the app – *adding*, *finishing*, and *deleting* a to-do item. First, we run the workloads under the builder mode to generate the DOM-tree types, then switch to the enforcer mode for performance measurement. For comparison, the baseline performance is obtained by running the same benchmark on the unmodified Electron of the same version.

For every implementation of the app, we fire 100 add-actions to add 100 to-do items, then fire 100 finish-actions for the items, and finally fire 100 delete-actions for the items. The frequency of our firings ranges from 10 actions/sec to 100 actions/sec, as shown in Figure 6. For each frequency, we calculate the average completion time of an action. Because there are 16 implementations of the app, and we fire 300 actions in every test, each completion time shown in Figure 6 is the average of 300 * 16 = 4800 measurements. For example, the average completion times of the baseline and our Electron platform, under the frequency 1000/80 (i.e., 12.5 actions/sec), are 80.8473 and 81.0793. The slowdown is (81.0793 - 80.8473) / 80.8473 = 0.287%.

Freq (1/s)	**1000/100**	**1000/90**	**1000/80**	**1000/70**	**1000/60**
Baseline (s)	101.055	90.9504	80.8473	70.7489	60.6612
Ours (s)	101.056	90.9951	81.0793	71.2372	61.5176
Freq (1/s)	**1000/50**	**1000/40**	**1000/30**	**1000/20**	**1000/10**
Baseline (s)	50.5433	40.4531	30.3582	20.2579	10.2518
Ours (s)	51.7759	42.2021	32.6014	22.7461	12.9292

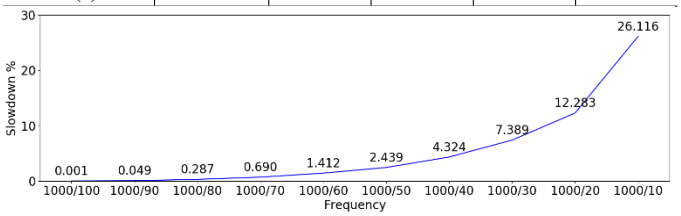

Fig. 6: Completion time and slowdown w.r.t. user-action frequency

This performance result proves that our mechanism does not cause a noticeable slowdown in a realistic scenario. We consider 10 actions/sec an aggressive load, for which the slowdown is 0.001%. What the curve shows is a stress-test using unreasonably aggressive loads, which help us understand the intrinsic performance characteristics. It is also worth noting that, because our modules are separate from Blink, they are not in the optimal position to perform. For example, the CSS "cascading" algorithm is complex. It produces intermediate data that Blink uses to trim subtrees that are not affected in a change. We are unable to utilize these intermediate data. Also, Blink uses a method to compress several style properties into one integer to expedite comparison operations. If our modules were integrated into Blink, it would benefit from this mechanism. Another optimization we have not done, which Blink does in similar situations, is to "compile" the DOM-tree type checking into "if-then-else" statements, instead of traversing the explicit tree structure, since the DOM-tree type has been decided at the development time. Despite these potential opportunities, the performance result demonstrates that our current platform is already practical to be adopted.

C. Responsible disclosure and code repository

We have reported the vulnerabilities of all 19 apps to the vendors. As of writing time, the vulnerabilities in Antares, Tess, Altair, Blockbench and Advanced REST Client have been confirmed and fixed. The vendors of Microsoft Teams, GraSSHopper, Homura, Jukeboks, DeckMaster, Poddycast, Boost Note and Appium Desktop have confirmed the vulnerability, and are fixing the vulnerabilities. The source code of our enhancement to Electron is publicly available (anonymized) [35]. Reference [36] is an anonymized website showing videos and screenshots of the exploits against the vulnerable apps.

VIII. Related work

Our work is related to defense techniques against web attacks, such as XSS. Many of them focus on input sanitization. Other techniques develop and enforce security policies, including some that learn policies based on data.

Techniques focusing on sanitization. DOMPurify [22] is a client-side sanitizer implemented in JavaScript. It is still actively maintained and widely used. It parses HTML string into DOM nodes, then flattens them as a list of `<tag name, attribute name>` pairs to apply filtering rules. XSSAuditor [23], implemented in the WebKit engine [24], is an early example of in-browser sanitizer that focuses on reflective XSS attacks. It places hooks in the interface between the HTML parser and the JavaScript engine, inspecting fully parsed DOM nodes rather than raw HTML. If the JavaScript engine tries to execute a string which previously appears as a part of the HTTP request, it is highly suspicious. Because XSSAuditor only deals with reflective XSS, Stock et al. [27] develop a taint-analysis-based method as a complementary defense. It can thwart client-side XSS attacks too.

We mention Trusted Types [9] in Section IV. It ensures that a set of well-known dangerous "sinks", e.g., `innerHTML`, the `src` attribute of `<script>`, etc., always receive sanitized contents. A sink does not accept a raw string, but only an object wrapping the raw string into one of the three types – `TrustedHTML`, `TrustedScript` and `TrustedScriptURL`. For every type, the programmer is required to provide its constructor which takes a raw string argument. The constructor is essentially a sanitizer. Thus, to obtain an object of the type, a raw string must go through sanitization.

Researchers also discuss important properties of sanitizers for security. Hooimeijer et al. [25] argue for the importance of commutativity and idempotence, and propose a programming language for sanitizer development that makes those properties verifiable. Saxena et al. [26] point out that context mismatch and non-commutativity are major sources of sanitization vulnerabilities, and propose a taint-analysis-based approach to apply the right sanitizer (or sanitizers) for a given context.

Techniques focusing on policies. PoliDOM [28] allows a programmer to specify parts of the DOM tree as read-only to defend against DOM-based XSS. The policies are written in CSS selector syntax. However, it does not handle the situation of disconnected elements, so we believe this is a vulnerability. ScriptChecker [29] allows web pages to restrict the capabilities, such as DOM access and network request, of individual JavaScript tasks to safely execute untrusted JavaScript code. CSPAutoGen [30] crawls a website to infer a CSP, with a focus on preventing unsafe usages of eval and inline scripts. In its training phase, scripts used by the website are converted to abstract syntax trees (ASTs) and added to a whitelist. In enforcing phase, an incoming script is allowed to execute only if its AST matches the whitelist. Mentioned in Section II, Synode [31] statically analyzes Node.js modules to compute a JavaScript and shell-script template for every call of `eval` or `exec`. The templates are expressed as ASTs. We mention in Section IV that "Types-from-data" [21] infers a type for structured data, e.g., JSON and XML, from a number of samples. It focuses on type safety rather than security.

IX. Conclusion

Our study about real-world Electron apps shows the fact that it is impractical for programmers to anticipate all the unexpected inputs that attackers may potentially think of. DOM-tree type-checking is a methodology to let programmers explicitly specify their intentions instead of non-intentions. We build TypeBuilder and TypeEnforcer into Electron. The methodology is practical, as it only requires a small amount of programmer effort. It prevents a DOM tree from gaining a functionality that is unintended in the programmer's mind, thus blocks all exploits we show in the study. Our responsible disclosure about the security issues has been positively responded. The source code of our Electron enhancement is public. We hope the community embraces the methodology to safeguard this new programming paradigm moving forward.

Acknowledgment

We thank the anonymous reviewers for valuable feedback. The work also benefits from discussions with Haoxiang Lin, Fan Yang and Mao Yang. Zihao Jin is in part supported by the Microsoft Research internship program. Jianjun Chen is in part supported by the National Natural Science Foundation of China (grant #62272265). Haixin Duan is in part supported by the National Natural Science Foundation of China (grant #U1836213 and #U19B2034).

References

[1] Electron. https://www.electronjs.org/.

[2] Sanitize-html. https://github.com/apostrophecms/sanitize-html.

[3] Felix Böhm. Htmlparser2. https://github.com/fb55/htmlparser2.

[4] Angular - Text interpolation. https://angular.io/guide/interpolation.

[5] Christopher Jeffrey. Marked. https://github.com/markedjs/marked.

[6] Nicolás Bevacqua. Insane. https://github.com/bevacqua/insane.

[7] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. "Scriptless Attacks: Stealing the Pie Without Touching the Sill." Proceedings of the 2012 ACM conference on Computer and communications security. 2012.

[8] Content Security Policy Level 3. https://w3c.github.io/webappsec-csp/.

[9] Trusted Types. https://w3c.github.io/webappsec-trusted-types/dist/spec/.

[10] element.h – Chromium Code Search. https://source.chromium.org/chromium/chromium/src/+/main:third_party/blink/renderer/core/dom/element.h;l=717;drc=1946212ac0100668f14eb9e2843bdd846e510a1e?q=IsURLAttribute&sq=&ss=chromium%2Fchromium%2Fsrc.

[11] 4.3.6. Enforcement in event handler content attributes - Trusted Types. https://w3c.github.io/webappsec-trusted-types/dist/spec/#enforcement-in-event-handler-content-attributes.

[12] Section 4.12.1 The script element – HTML Standard. https://html.spec.whatwg.org/#the-script-element.

[13] longhands.h – Chromium Code Search. https://source.chromium.org/chromium/chromium/src/+/main:out/Debug/gen/third_party/blink/renderer/core/css/properties/longhands.h.

[14] Section 4.3.1. Interface MutationObserver - DOM Standard. https://dom.spec.whatwg.org/#interface-mutationobserver.

[15] Section 4.4. Interface Node - DOM Standard. https://dom.spec.whatwg.org/#interface-node.

[16] setTimeout() – Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/setTimeout.

[17] Event: 'will-navigate'. https://www.electronjs.org/docs/latest/api/web-contents#event-will-navigate.

[18] WebFrameMain. https://www.electronjs.org/docs/latest/api/web-frame-main.

[19] Scanner.cc – Chromium Code Search. https://source.chromium.org/chromium/v8/v8.git/+/edf3dab4660ed6273e5d46bd2b0eae9f3210157d:src/scanner.cc.

[20] Speedometer 2.0. https://browserbench.org/Speedometer2.0/.

[21] Tomas Petricek, Gustavo Guerra, Don Syme. "Types from data: Making structured data first-class citizens in F#". In proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2016.

[22] Mario Heiderich, Christopher Späth, and Jörg Schwenk. "DOMPurify: Client-Side Protection Against XSS and Markup Injection." European Symposium on Research in Computer Security. Springer, Cham, 2017.

[23] Daniel Bates, Adam Barth, and Collin Jackson. "Regular Expressions Considered Harmful in Client-Side XSS Filters." Proceedings of the 19th International Conference on World Wide Web. 2010.

[24] WebKit. https://webkit.org/.

[25] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. "Fast and Precise Sanitizer Analysis with BEK." 20th USENIX Security Symposium (USENIX Security 11). 2011.

[26] Prateek Saxena, David Molnar, and Benjamin Livshits. "SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications." Proceedings of the 18th ACM conference on Computer and communications security. 2011.

[27] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. "Precise Client-side Protection against DOM-based Cross-Site Scripting." 23rd USENIX Security Symposium (USENIX Security 14). 2014.

[28] Junaid Iqbal, Ratinder Kaur, and Natalia Stakhanova. "PoliDOM: Mitigation of DOM-XSS by Detection and Prevention of Unauthorized DOM Tampering." Proceedings of the 14th International Conference on Availability, Reliability and Security. 2019.

[29] Wu Luo, Xuhua Ding, Pengfei Wu, Xiaolei Zhang, and Qingni Shen. "ScriptChecker: To Tame Third-party Script Execution With Task Capabilities." NDSS. 2022.

[30] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. "CSPAutogen: Black-box Enforcement of Content Security Policy upon Real-world Websites." Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. 2016.

[31] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. "SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS." NDSS. 2018.

[32] Xing Jin, Xunchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. "Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation." Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. 2014.

[33] Wikipedia. Single-page application. https://en.wikipedia.org/wiki/Single-page_application.

[34] DeveloperDrive. Seven JavaScript Templating Engines with Code Examples. https://www.developerdrive.com/best-javascript-templating-engines/.

[35] Source code repository of our project (anonymized). https://github.com/1qaz2wsx7u8i9o0p/DOM-Tree-Type.

[36] Videos and screenshots of the exploits. https://1qaz2wsx7u8i9o0p.github.io/.

APPENDIX

A. Style properties excluded in DOM-tree type

```
top, bottom, left, right, width, height, block-size, inline-size, grid-template-columns,
grid-template-rows, margin-block-end, margin-block-start, margin-bottom, margin-inline-end,
margin-inline-start, margin-left, margin-right, margin-top, padding-block-end, padding-block-start,
padding-bottom, padding-inline-end, padding-inline-start, padding-left, padding-right, padding-top,
perspective-origin, transform, transform-origin
```

B. DOM-tree type in BNF[1]

1	`Element`	=	`"<" TagName *AttributeOrStyleProperty ">" (*Element / StructureAgnosticSubtree) "</" TagName ">"`[2][3]
2	`TagName`	=	`String`
3	`StructureAgnosticSubtree`	=	`*("<" TagName *AttributeOrStyleProperty "></" TagName ">")`
4	`AttributeOrStyleProperty`	=	`ScriptAttribute / URLAttribute / StringAttribute / StyleProperty`
5	`ScriptAttribute`	=	`ScriptAttributeName "=" *TokenSequence`[4][5]
6	`ScriptAttributeName`	=	`"onclick" / "onload" / "onerror" / ...`[6]
7	`TokenSequence`	=	`*Token`[7]
8	`Token`	=	`"IDENTIFIER" / "LPAREN" / "NUMBER" / "RPAREN" / "STRING" / ...`[8]
9	`URLAttribute`	=	`URLAttributeName "=" *Origin`[4][5]
10	`URLAttributeName`	=	`"src" / "href" / ...`[9]
11	`StringAttribute`	=	`StringAttributeName "=" *WildcardString`[4][5]
12	`StringAttributeName`	=	`"id" / "name" / "target" / "method" / ...`[10]
13	`StyleProperty`	=	`StylePropertyName "=" StylePropertyValue`[5]
14	`StylePropertyName`	=	`"background-color" / "opacity" / "z-index" / ...`[11]
15	`StylePropertyValue`	=	`*Origin "," *WildcardString "," *StylePropertyValueSequence "," StylePropertyValueRange`[4]
16	`StylePropertyValueSequence`	=	`*StylePropertyValue`[7]
17	`StylePropertyValueRange`	=	`"<" Float "," Float ">"`
18	`Origin`	=	`WildcardString "://" WildcardString ":" WildcardString`
19	`WildcardString`	=	`(String ("*" / "?") WildcardString) / String`

Notes:

[1] Following conventions from RFC5234: Augmented BNF for Syntax Specifications: ABNF, specifically, the following sections:
- 2.3. Terminal Values
- 3.1. Concatenation
- 3.2. Alternatives
- 3.5. Sequence Group
- 3.6. Variable Repetition

[2] For simplicity, white spaces before `*Attribute` and between `Attributes` are omitted.

[3] The two `TagNames` should match each other. Due to space limitations, we do not enumerate every possible tag name of HTML elements.

[4] For simplicity, vertical bars (`|`) between `TokenSequences`, `Origins`, `Strings`, and `StylePropertyValueSequences` are omitted.

[5] For simplicity, quotes (`"`) around `*TokenSequence`, `*Origin`, `*String`, and `StylePropertyValue` are omitted.

[6] Scripting attributes defined by Blink's `Element::IsScriptingAttribute` function.

[7] For simplicity, white spaces ( ) between `Tokens` and `StylePropertyValues` are omitted.

[8] Tokens defined by V8 JavaScript engine's `v8::Token` class.

[9] URL attributes defined by Blink's `Element::IsURLAttribute` function. Whether an attribute is a URL attribute depends on its attribute name and the element's tag name, e.g., `href` of `<a>`, `src` of `<img>` or `<video>`, etc.. Due to space limitations, we do not enumerate every valid `TagName-URLAttributeName` pair.

[10] A few important attributes that contain only plain strings, as opposed to URL or JavaScript code. For simplicity, we only list attribute names rather than `TagName-StringAttributeName` pairs.

[11] CSS Properties defined by Blink's `css_properties.json5`, excluding layout-dependent ones.

C. DOM-tree type violations

Name	Location	
Visual Studio Code	URL extraction	/HTML/BODY/DIV/DIV/DIV/DIV[@id="quickInput_list"]/DIV/DIV/DIV/ DIV[@id="list_id_*"]/DIV[@id="list_id_*"]/DIV/DIV/DIV/DIV[@id=""]/DIV/ /DIV/DIV/DIV/DIV[@id="workbench.parts.editor"]/DIV/DIV/DIV/DIV/DIV/ DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV[@id="list_id_*"]/ DIV[@id="list_id_*"]/P
	Code comment	/HTML/BODY/DIV/DIV/DIV/DIV[@id="quickInput_list"]/DIV/DIV/DIV/ DIV[@id="list_id_*"]/DIV[@id="list_id_*"]/DIV/DIV/DIV/DIV[@id=""]/DIV/ /DIV/DIV/DIV/DIV[@id="workbench.parts.editor"]/DIV/DIV/DIV/DIV/DIV/ DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV[@id="list_id_*"]/ DIV[@id="list_id_*"]/P
GraSSHopper	Text selection	/HTML/BODY/DIV[@id="container"]/DIV[@id="wrapper"]/DIV/DIV/DIV/ DIV[@id="sizzle*"]/DIV/DIV/DIV/DIV/DIV/DIV/DIV
	Host name	/HTML/BODY/DIV[@id="container"]/DIV[@id="wrapper"]/DIV/DIV/DIV/ DIV[@id="sizzle*"]/SPAN/DIV/SPAN
Antares	/HTML/BODY/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DETAILS/DIV/DIV/UL/LI/A/SPAN	
Homura	/HTML/BODY/DIV/DIV/DIV/DIV/DIV/MAIN/DIV	
OhHai Browser	/HTML/BODY/DIV/DIV/ACC-PANEL/ACC-ITEM/HIST-LIST/DIV/DIV[@id="HistList"]/DIV/DIV/ DIV/A	
Jukeboks	/HTML/BODY/DIV/CONTENT/DIV/DIV/NAV/DIV	
Poddycast	/HTML/BODY/DIV/DIV[@id="content-right"]/DIV[@id="content-right-body"]/DIV/UL/LI/ DIV	
Tess	/HTML/BODY/DIV/DIV/DIV/DIV/DIV	
WAIL	/HTML/BODY/DIV[@id="wail"]/DIV/DIV/DIV/DIV[@id="addSeedCard"]/DIV/DIV/DIV/DIV/ P[@id=”checkSeedResults”]	
Advanced REST Client	/HTML/BODY/DIV/DIV/MAIN[@id="main"]/ARC-REQUEST-WORKSPACE/SECTION/ ARC-REQUEST-PANEL/RESPONSE-VIEW/DIV[@id="panel-headers"]/DETAILS/HEADERS-LIST/ DIV/DIV/SPAN	
Altair	/HTML/BODY/APP-ROOT/APP-ALTAIR/DIV/NZ-LAYOUT/NZ-LAYOUT/NZ-LAYOUT/NZ-CONTENT/	
Another Redis Desktop Manager	/HTML/BODY/DIV/DIV/DIV/P	
Appium Desktop	/HTML/BODY/DIV/DIV[@id="serverMonitorContainer"]/DIV/DIV/SPAN/SPAN/SPAN/SPAN	
Blankup	/HTML/BODY/DIV[@id=""]/DIV/DIV	
Blockbench	/HTML/BODY/UL/LI/UL/LI	
Boost Note	/HTML/BODY/DIV[@id="root"]/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/DIV/ DIV/DIV/DIV	
DeckMaster	/HTML/BODY/DIV/DIV/DIV/DIV/DIV/A	
Electro-CRUD	/HTML/BODY/APP-ROOT/NB-LAYOUT/DIV/DIV/DIV/DIV/DIV/NB-LAYOUT-COLUMN/APP-VIEW/ APP-VIEW-VIEW/NB-LAYOUT/DIV/DIV/DIV/DIV/DIV/NB-LAYOUT-COLUMN/NB-CARD/NB-CARD-BODY/ NGX-DATATABLE/DIV/DATATABLE-BODY/DATATABLE-SELECTION/DATATABLE-SCROLLER/ DATATABLE-ROW-WRAPPER/DATATABLE-BODY-ROW/DIV/DATATABLE-BODY-CELL/DIV/SPAN	