SoK: On the Analysis of Web Browser Security

Jungwon Lim*, Yonghwi Jin*†, Mansour Alharthi, Xiaokuan Zhang,
Jinho Jung, Rajat Gupta, Kuilin Li, Daehee Jang‡, Taesoo Kim
Georgia Institute of Technology †Theori Inc. ‡Sungshin Women's University

***Abstract*—Web browsers are integral parts of everyone's daily life. They are commonly used for security-critical and privacy sensitive tasks, like banking transactions and checking medical records. Unfortunately, modern web browsers are too complex to be bug free (*e.g.*, 25 million lines of code in Chrome), and their role as an interface to the cyberspace makes them an attractive target for attacks. Accordingly, web browsers naturally become an arena for demonstrating advanced exploitation techniques by attackers and state-of-the-art defenses by browser vendors. Web browsers, arguably, are the most exciting place to learn the latest security issues and techniques, but remain as a black art to most security researchers because of their fast-changing characteristics and complex code bases.**

To bridge this gap, this paper attempts to systematize the security landscape of modern web browsers by studying the popular classes of security bugs, their exploitation techniques, and deployed defenses. More specifically, we first introduce a unified architecture that faithfully represents the security design of four major web browsers. Second, we share insights from a 10-year longitudinal study on browser bugs. Third, we present a timeline and context of mitigation schemes and their effectiveness. Fourth, we share our lessons from a full-chain exploit used in 2020 Pwn2Own competition. We believe that the key takeaways from this systematization can shed light on how to advance the status quo of modern web browsers, and, importantly, how to create secure yet complex software in the future.

I. Introduction

Web browsers play an integral role in the modern, Internet-connected lifestyle. We rely on web browsers to pay mortgages, schedule vaccines, and connect to people worldwide. In other words, web browsers become the gatekeeper to cyberspace, and their insecurity is a critical threat to the safety, fairness and privacy of our society. Unfortunately, web browsers have been the most attractive, valuable target of cyber attacks—50% of 0-day exploits found in the wild were attacking web browsers in 2021 [58] and threatened *every single* person on the Internet [103], [141], [195], [196], [211], [230], [231].

Accordingly, modern web browsers naturally become a battlefield for attackers who wish to break in with novel exploit techniques, and browser vendors who want to keep users safe with the most advanced mitigation schemes. Browser vendors are indeed the essential players that advance modern security practices by 1) open sourcing not only the current architecture and code but also the design process itself [45], [52], [210]; 2) introducing bug bounty awards to encourage the discovery of 0-day bugs [116], [176]; and 3) proactively finding exploitable bugs by developing and running state-of-the-art fuzzers on the cloud [109], [111].

Unfortunately, detailed design decisions for security and insights on new mitigations against novel exploitation are often considered as a black art, keeping their lessons learned within the web browser community. This is partly because of their complex architecture, fast-changing implementation, and overwhelming size of code bases, but mainly because it is non-trivial to systematize the knowledge of all major web browsers coherently and objectively simultaneously. Experts from each browser vendor have attempted to provide their perspectives on security design and decisions, *e.g.*, Chrome [44], IE [51], Firefox [54]. Previous industry reports published in 2017 [90], [226] mainly focus on describing individual techniques and defenses in an ad-hoc, as-it-is manner without developing academic perspectives or providing insights and lessons that are useful to envision the next directions for the community.

This paper makes a bold attempt to systematize the security landscape of modern web browsers. We first provide a unified model of four major web browsers as they pertain to security, and compare and contrast their security decisions by using the provided model. Second, based on the model, we analyze security bugs found in each open source browser in the last 10 years, and show their relation to the development of new mitigation schemes, bug bounty programs, and known exploitation techniques used in the wild. Third, based on our study, we convey our insights and lessons to inspire researchers and developers who are shaping the future of web browsers. We hope that our systematization attempt can help them to understand the approaches of each vendor in a holistic manner and thus enhance their security designs to minimize security impacts and attack surfaces.

Challenges. Three unique characteristics make it challenging to systematize the knowledge of web browser security.

1) **A moving target.** Browser vendors make decisions rapidly (*e.g.*, weekly updates) and their development is at a much faster pace than any other software that humans have built. To infer insightful lessons, we strive hard to stay focused on fundamental design issues and approaches in web browsers.
2) **Overwhelming size.** Modern web browsers are built with a few million lines of code, *e.g.*, Chrome consists of 25 million lines of code [16]. In addition to the project size, information on web browsers, such as 0-day exploits and mitigations, is scattered all over the Internet and fails to provide a holistic summary and overview of the security landscape. In this paper, we limit our interest to the four major web browsers, namely, Chrome, Firefox, Safari, and Edge, and study multiple, public sources for their security issues: issue trackers [43], [48], CVE reports [2], [4], [5],

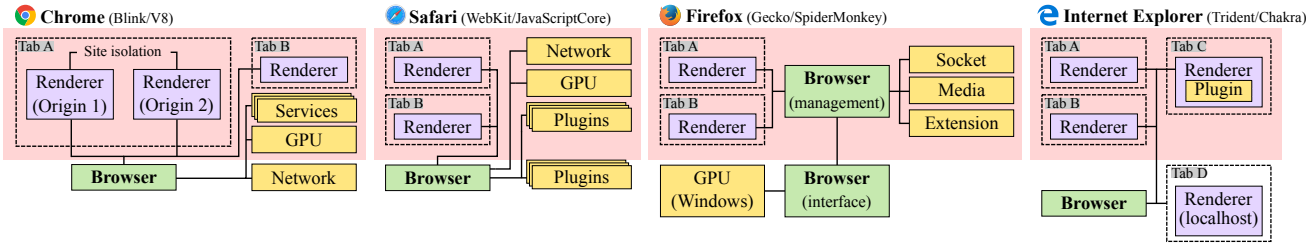


Fig. 1: Internal architecture of four major web browsers. All browsers deploy a sandbox (pink region) to restrict the renderer, while the detailed sandboxing methodology differs based on the underlying OS. There are subtle but important differences across browsers.

[7], [9], [11], [12], [14], [55], code repositories [30], [45], [52], [60], and technical reports from the vendors [49], [58], [65], [67], [79], [86], [152], [161], [163], [170], [171], [204], [208], [230], [231].

3) **Unique designs.** It is also important to provide an objective perspective on their security issues; each browser has its own restrictions and requirements in making decisions (*e.g.*, release deadline), and it is critical to focus on the fundamental issues that our community can learn from. To solve this, we provide a unified architecture that compares and contrasts each browser's design conceptually without compromising their implementation details.

Contributions. This paper makes the following contributions:

- Provide a detailed comparison of modern browser architectures in terms of sandboxing schemes;
- Present a 10-year longitudinal study on browser bugs;
- Categorize browser vulnerabilities with detailed analysis;
- Study state-of-the-art generic mitigations on browsers;
- Perform a detailed study on a real-world full-chain exploit.

Non-goals of this SoK. The main focus of this study is *web browser* security, concerning the security of its own vulnerabilities. We do not consider other web-related security problems, like *web* or *web server* security issues such as Cross-Site Scripting (XSS), SQL Injection, etc. Note that although *Universal Cross-Site Scripting (UXSS)* [166] sounds similar to XSS, it commonly originates from problems in the browser's implementation and design, so it is considered *web browser* security (§III-E).

II. Modern Browser Architecture

This section provides a unified model of each web browser that enables an objective comparison of their approaches.

A. Overview

Modern web browsers adopt the principle of least privilege by using an OS process as a protection domain. By using the process domain, each web browser can be described using three types of processes, namely, a *browser* process (marked in green), *renderer* processes (marked in magenta), and task-specific processes (marked in yellow), as shown in Figure 1.

Browser process. When a web browser launches, the browser process runs with the same privilege level as the user (*i.e.*, a higher privilege) and passes a sandbox profile to the OS to restrict the privileges of other processes to be spawned with (*i.e.*, a lower privilege). It manages all child processes (*e.g.*, renderer) and is the only process that directly interacts with users via system calls (and a user interface).

Renderer process. This process is responsible for parsing and rendering the untrusted web content. The ever-growing kinds of data served on the web have caused the renderer process to include a wide variety of components, such as media parsers, DOM and JS engines. Since they are major sources of browser bugs, they are confined in a restrictive sandbox (see §II-C). The renderer processes are typically spawned per browser tab or per web page origin. The isolation policy of each renderer varies by security policy or features (*e.g.*, site isolation) of each web browser, available resources at runtime (*e.g.*, low memory in mobile), or even user configuration.

Other process. A modern browser's architecture is highly modular. This modular design enables browsers to have different privilege levels based on the process's role. Services that interact with external drivers (*e.g.*, networking or GPU processes) are isolated as a separate process, which enables more restrictive sandboxing for the processes that don't require such access like renderer process. Web browsers also commonly put extensions and plugins in separate processes. This protects plugins that are at a higher privilege level from malicious web content, and protects browsers from being hijacked in the case of a malicious plugin.

Inter-Process Communication (IPC). Since these processes cannot directly access each other's memory, they always communicate via IPC channels provided by the OS, and communications are usually mediated by the browser (*broker*) process. In other words, the browser process works as a reference monitor that restricts direct accesses to important data or high-privileged operations (*e.g.*, cookies or system calls) from other processes. Thanks to this multi-process architecture, an attack is always initiated from a low privileged process like a renderer process, and the attacker's goal is to break into the browser process running as a user's privilege. At the same time, it makes it possible to recover from crashes caused by a benign bug in the renderer process, making the browser resilient against stability issues.

Same-Origin Policy (SOP). In reality, websites consist of contents from numerous sources with varying origins, *e.g.*, using CDN for common JavaScript libraries, embedding external sites via *iframes*, or enabling a *like* button from a social network. The complex nature of websites leads to numerous security policies and unique features of each web browser. Based on the origin of each website [94], the browser process and the renderer process restrict which resources (*e.g.*, cookies)

a web page is allowed to interact with, which is the same-origin policy (SOP) [94].

B. Differences in Browsers

The so-far discussed design is equally applied to all four major browsers. However, as shown in Figure 1, some implementation details differ depending on the design of the browsers and their underlying operating systems. For example, the GPU processes in Chrome and Safari are separated from the renderer processes, with a sandbox profile that enables them to access the platform 3D APIs [67] (see §II-C). Also, Chrome, Firefox, and Safari each has a separate process to handle the network service, while the Chrome network service is placed outside the sandbox. Chrome team is currently implementing the sandbox of its network service [28].

Site isolation. The sandbox mechanism can indeed protect browsers; however, with the discovery of universal cross-site scripting (UXSS) attacks, it turned out that attackers could steal user data without needing to escape the sandbox. To address such attacks, the Chrome team came up with *Site Isolation* [186] to further separate different site origins into different processes. It creates a dedicated process for each site origin, so that there is no implict sharing among different origins. Site isolation is an effective measure to address UXSS, but it is also beneficial for preventing hardware-based transient execution attacks [137], [147]. Firefox has a similar project named *Fission* [175], and it is shipped in Firefox 88 Beta [174].

JavaScript engines. JavaScript engines are the core of modern browsers, which convert JavaScript code into machine code. Major browsers use just-in-time (JIT) compilation [36] [34] [33] to speed up the code execution. Also, JIT compilers model the result and side-effects of all operations and run various analysis passes to optimize the code. If any of these goes wrong, native code with memory corruption issues can be emitted and executed, which can lead to severe security implications [26], [121]. While each engine has different implementations, they share similar design principles and have common attack surfaces (§III-D). Therefore, attackers can build generic attack primitives which work across different engines, such as *fakeobj* and *addrof* primitives [154], [189] and *element kind transitions* [153], [190]. JavaScript engines are being used outside browsers as well (*e.g.*Node.js uses V8), amplifying the impact of security bugs in JavaScript engines. We discuss issues caused by homogeneous browser engines in §VI-A.

Rendering engines. Rendering engines are responsible for interpreting resources and rendering webpages. Each of the four major browsers has its own rendering engine: Safari uses WebKit; Chrome uses Blink (forked from WebKit); Firefox uses Gecko; Edge uses Blink (replacing EdgeHTML). *Web Standards* [219], [224] serve as baseline specifications and references for browser vendors to implement their rendering engines. Since *Web Standards* continuously evolve with new features, there are rapid changes in rendering engines, *i.e.*, implementing new features or dropping deprecated ones. Due to different decision process and implementation strategy, the feature sets implemented in the rendering engines in different browsers are quite different [42], resulting in different attack surfaces [228]. We discuss the attack surfaces in §III.

C. Variances in Sandbox Schemes

The sandbox restricts the program execution from deviating from its intended mission. However, the underlying technology and architecture for building a sandboxed environment significantly differs among OSes. To examine the internals of sandbox implementations, we 1) audit the source code of browsers, 2) monitor the behavior of the sandbox APIs, and 3) analyze the predefined sandbox policy file (*e.g.*, Safari browser's configuration). We summarize our findings in Table I.

Categorizing sandbox primitives. In Table I, we categorize sandboxing primitives into three categories based on their roles: *a) privilege reduction* applies more restricted privileges to the sandboxed processes using the permission system of platforms such as DAC/MAC; *b) domain separation* allocates a separated space of resources that a sandboxed process will have access to; *c) attack surface reduction* limits accesses to system services, kernel or device drivers.

Browser-specific characteristics. Browser vendors utilize different primitives depending on the given constraints (*e.g.*, available memory). For example, Site Isolation prevents RCE exploits to be transformed into UXSS or sandbox escapes by putting an origin-wise, process-level security boundary between a compromised renderer and privileged web pages [107], [108].

OS-specific behaviors. We also compare the sandbox features from different OSes, namely, Windows, Linux and macOS.

Windows. Windows restricts each process by using a security token [117]. Similar to the capability-based model, a process obtaining a certain token level can access privileged resources with proper security descriptor level. For example, the renderer process runs with a *low integrity* token level, and the broker process runs with a *medium integrity* token level, so any write accesses from a renderer process to the broker process will be restricted by default.

However, there is no unified protocol for fine-grained access control. To resolve this, Chrome and Firefox support fine-grained rulesets using their own IPC mechanisms and binary-level code patches on resource-related functions [117]. Microsoft introduced AppContainer in Windows 8 to enforce more fine-grained access control to resources by adding a notion of capabilities attached to process tokens. Edge created a sandbox based on AppContainer [89]. Starting from the *deny-by-default* policy, Edge created a set of capabilities for required system resources. Chrome is also experimenting with an AppContainer-based sandbox [28]. Browsers also utilize various features for mitigating sandbox escape. For example, *alternate desktop* and *alternate window station* can be used to mitigate UI-based attacks such as Shatter [180]; *lockdown default DACL* [24] and *Random Restricting SIDs* [38] were introduced to enforce more restricted DACLs, so that compromised sandboxed processes cannot access other sandboxed processes.

Legend†: GMP: Gecko Media Plugin, **UNT:** Untrusted, **MED:** Medium, **LMT:** Limited, **LKD:** Lockdown, **NAD:** Non-Admin, **LTU:** Limited User, **IZG:** Inherited from Zygote, **BSC:** seccomp-BPF + Sandboxed IPC

		Chrome									Firefox							Edge			Safari		
		Zygote	Renderer	Storage	Flash	NaCl	GPU	Network	Audio	Crashpad	Content	Media	GMP†	Socket	NPAPI	Flash	GPU	Internet	Flash	Extension	WebProcess	GPU	Network
	Sandboxed	✓	✓		✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			
		✓	✓	✓		✓					✓	✓											
			✓		✓	✓	✓	✓	✓		✓	✓	✓	✓		✓					✓	✓	✓
Privilege reduction	Integrity†		UNT		UNT	UNT	Low	MED	Low	MED	Low	UNT	UNT	UNT	MED	Low	MED	Low	Low	Low			
	AppContainer																	✓	✓	✓			
	Access token level†		LMT		LMT	LMT	LKD		LKD		LMT	LKD	LKD		NAD	LMT							
	Hardened token	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
	Job level†		LKD		LKD	LTU	LTU		LKD		LKD	LKD	LKD	LKD			None						
	NS:User†	✓	IZG	IZG	IZG	✓					✓	✓											
Domain separation	Lockdown default DACL		✓		✓		✓		✓		✓	✓	✓	✓		✓							
	Random restricting SIDs						✓				✓	✓	✓	✓		✓							
	Alternate desktop	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						
	Alternate window station	✓	✓	✓	✓	✓	✓	✓	✓	✓													
	Chroot	✓	✓	✓	✓	✓					✓	✓											
	NS:Network†	✓	IZG	IZG	✓					✓	✓												
	NS:IPC										✓	✓											
	NS:PID	✓	✓	✓	✓	✓																	
Attack surface reduction	Win32k lockdown		✓		✓							✓		✓									
	Disable non-system font		✓		✓	✓	✓		✓							✓							
	Hypervisor based sandboxing																	✓	✓	✓			
	System call allowlisting†	BSC	BSC	BSC	BSC	BSC					BSC	BSC											
	System call allowlisting																				✓		
	# of allowed Mach services		10		7	6	25	16	10		14	9	1	4		55					53	56	54
	# of allowed IOKits		3		1	2	13	0	2		10	7	1	0		15					1	18	2
	Site Isolation		Origin		Origin						Tab							Zone	Zone	Zone			

TABLE I: Sandbox comparison (Chrome, Firefox, Edge, Safari on Windows, Linux, MacOS)

Linux. Unlike on Windows, the Linux sandbox is mainly based on *seccomp*, *chroot* and *namespace*. First, *seccomp* is a standard system call filter based on the eBPF language. Since the default *seccomp* configuration is overly tight, browsers define their own filtering rules. For example, Chrome applies its custom *seccomp* rules to all processes except the broker process, and the detailed rules vary for each process. Second, to restrict file access, Linux-based browser sandboxes utilize *chroot* jailing. Once a process is confined with *chroot*, no upper hierarchy of the file system is reachable. For example, Firefox applies *chroot* jailing to all renderers and only allows them to access specific files based on file descriptors obtained from the broker process. Also, browsers use *namespaces* [74] to create separated spaces for various resources, such as user, networking, and IPC. For example, creating and joining a user namespace enables a sandboxed process to be in a separate UID and GID, effectively disabling access to other unsandboxed processes.

macOS. While Windows and Linux support various types of sandboxing primitives, macOS supports a specifically formatted *sandbox profile* (`.sb`) [75] to describe the sandbox policy for a given process. Typically, the file provides an allowlist of absolute file paths that are allowed to be accessed and blocks all other file accesses by default. The profile also defines the capability of accessing other resources such as network and shared memory, and supports systemcall-based filtering like Linux's *seccomp*, though it is only deployed on Safari.

Mobile platforms. Since a process-based sandbox uses a non-trivial amount of memory, mobile platforms introduce subtle differences in sandbox policies or disable them depending on the available resources. For example, on Android, Site Isolation in Chrome is enabled only when the device has enough memory (>1.9GB), and the user need to enter passwords on the website [77]. On iOS, Safari uses sandbox rules that are different from macOS because different system services and IOKit drivers are exposed on mobile. Due to such differences, some exploits may work only on mobile platforms [152].

D. Exploiting Browsers

The goal of browser exploitation is to steal sensitive data from its user or to install malware for further action. Attackers can execute attacks like UXSS to directly steal data, or get a code execution first and then try to escape the sandbox. An attacker might attempt to gain a system's privilege and escape the sandbox by attacking the kernel, which is out of the scope of this paper. Thanks to various mitigation schemes (see §III,§IV), an attacker should chain multiple bugs (*e.g.*, 4 bugs until sandbox escape [134]) together to gain an arbitrary execution. Even after the control-flow is hijacked, since the renderer process runs inside the sandbox, the attacker should find another set of bugs in the broker process to escape the sandbox. Depending on exploits available in one's arsenal, an attacker often attempts to exploit a bug in the system services, drivers, or the kernel instead of the broker process to break out of the sandbox [155].

III. Browser Vulnerabilities and Mitigations

In this section, we first perform a measurement study on publicly reported browser bugs in the past decade to reason about the trends and then present dominant types of vulnerabilities (*e.g.*, JavaScript engine bugs) and their corresponding mitigations deployed by the vendors.

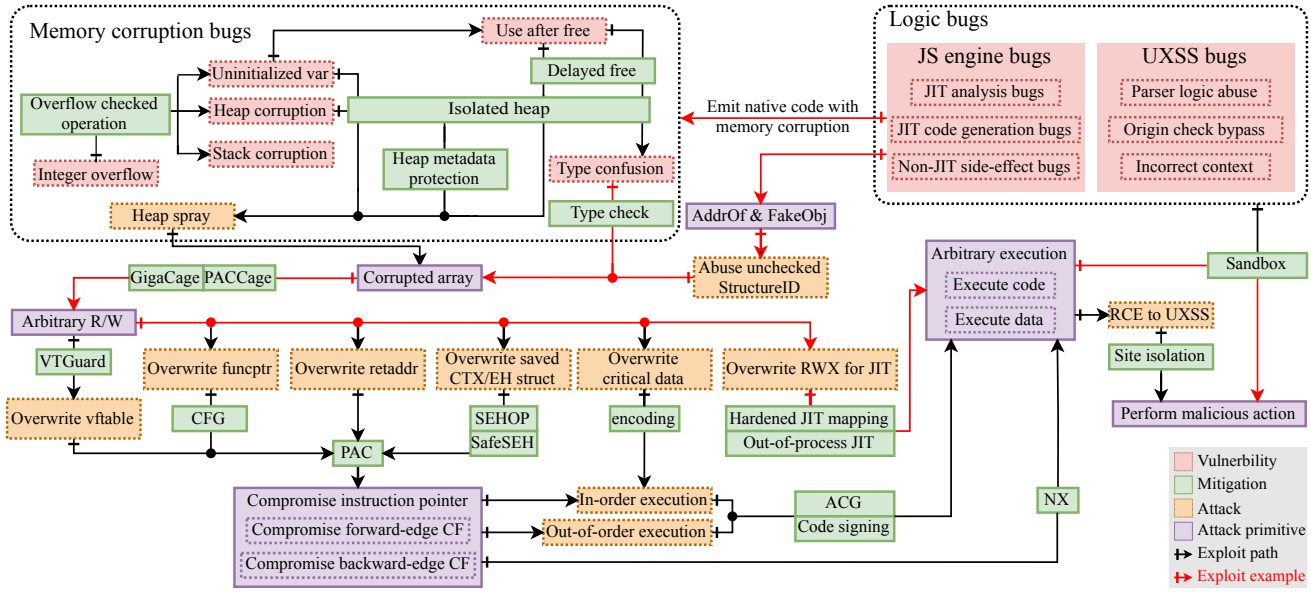


Fig. 2: Browser exploitation scenarios and bug classification. We mainly focus on browser security-specific issues and omit basic software attack/defense techniques such as ROP, NX and ASLR. The **Exploit example** arrows depict the exploit path described in §V.

	Firefox†	Chromium†	Safari/WebKit‡	Edge/IE‡
(1) Total CVEs	2190	2582	1436	2278
(2) Collected CVEs	2066	1912	1436	2278
(3) Pwn2Own [61]	14	9	37	31
(4) Google P0 [58]	7	22	8	22

TABLE II: (1) The total number of CVEs in the NVD database [55]. (2) The number of collected bugs. For open source browsers†, we collected extended bug information from vendors' bug trackers [48] [43]. For those browsers, we ignored confidential bugs and bugs not linked to bug tracker issues. For closed source browsers‡, we used NVD [55] as the sole source of CVE data. (3) & (4) are the sources used to collect data of exploited bugs.

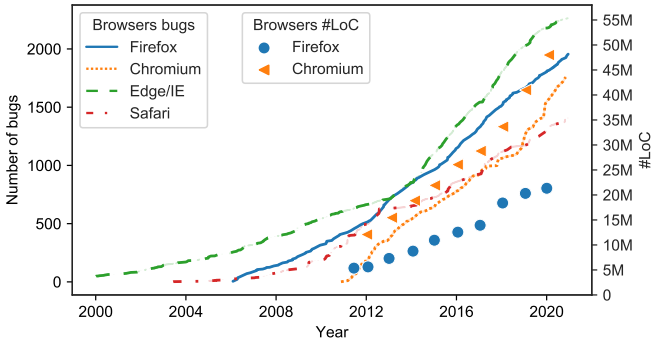


Fig. 3: Left y-axis: number of security bugs; Right y-axis: LoC of two open-source browsers: Firefox and Chromium. LoC is based on the first major version bump each year.

A. Trends of Browser Bugs

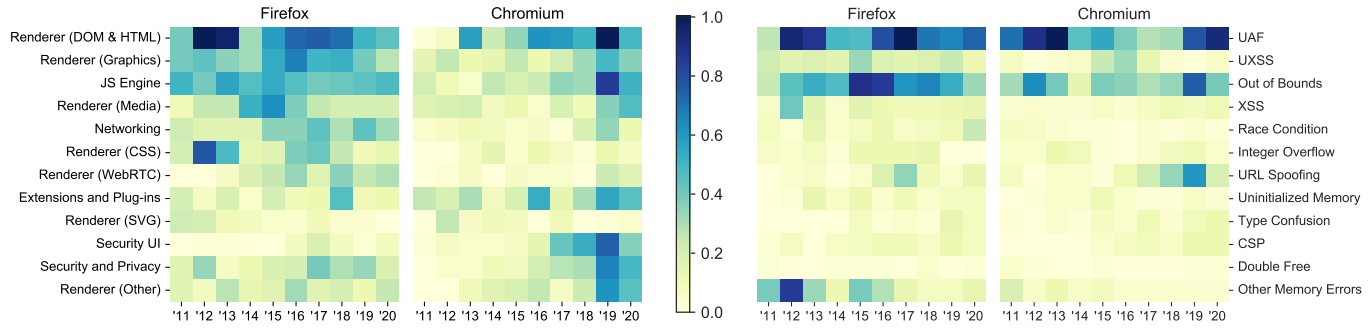
Data collection. We study public CVEs and vulnerability reports for four major browsers: 1) routinely updated security advisories from 'browser vendors [53], 2) public issue trackers released by vendors [43] [48], 3) open source code repositories that have a convention of linking bug-fixing commits to published vulnerabilities [30] [45] [60] [52], 4) CVE reports in the National Vulnerability Database (NVD) [55], 5) security bugs used in real-world exploits such as bugs used in Pwn2Own [61], and Google Project Zero reports [49], [58]. Table II summarizes the yield of our data collection efforts.

Bugs and codebase size. Figure 3 shows the sharp increase of security bugs in all browsers, specifically starting after 2010. We correlate this increase in bugs to the ever-growing codebase of browsers, as new features are added constantly. In addition, the advances in bug-finding techniques after 2010 played a considerable role, which we highlight in §VI-C.

Dynamic attack vectors. The enormous size and the continuously changing nature of browsers make the attack vectors change constantly. For the open source browsers, Firefox and Chromium, we map bugs to their respective host components and bug classes in Figure 4. For both browsers, we use the developer's assigned flairs to map bugs to their host browser components, and we use keyword matching in bug descriptions to categorize their classes.

Renderer bugs are dominant in both Firefox and Chromium since renderers are the core of browsers. The rise of URL spoofing bugs for Chromium since 2016 highlights the ease of finding bugs in previously unexplored areas. Memory bugs in general, and UAF bugs specifically, remain the greatest common denominator bug class for both browsers.

Another general observation is the varying number of bugs across the two browsers along the years in both dimensions. For example, for bug components, Chromium has more DOM & HTML bugs recently, but the number of DOM & HTML bugs is decreasing for Firefox. For bug class, in 2019 most of the bugs in Chromium were classified as UAF, OOB, and URL spoofing bugs, but Firefox depicts a relatively uniform bug distribution across the years. Thus, this discrepancy visualizes not only the changing attack vectors, but also the changing policies of triaging security bugs for different browsers.



(a) Browser Components **(b)** Bug Classes

Fig. 4: Mapping of bugs to host browser components and bug classes in Firefox and Chromium. The figure highlights the changing nature of browsers' attack surfaces year-on-year. The numbers in each figure are Min-Max scaled.

Browsers' efforts against bugs can also be identified in the figures. Chromium's Site Isolation [39], [186] as mitigation to UXSS bugs led to the apparent decrease of said bugs after Site Isolation was implemented in 2017 (Figure 4b). Some parts remain as the main source of bugs such as the DOM & HTML component, which we detail in §III-C.

Memory-safe language. Memory-safety bugs are critical and dominant in browsers. For example, Chromium labels over 70% of their high severity bugs as memory-safety issues, half of which are UAF bugs [50]. We show the ratio of memory-safety bugs in browsers in Figure 5. As shown in the figure, memory-safety bugs remain dominant for the past decade despite existing mitigations [212] [59]. Recently, there have been efforts in rewriting browsers using memory-safe languages (*e.g.*, Rust) to mitigate memory-safety bugs. For example, Mozilla is rewriting parts of Firefox in Rust with an ongoing project called Oxidation [57]. Up until 2020, the Oxidation project had replaced 12% of Firefox's components with Rust equivalents. Five of the replaced subcomponents fall under the renderer's media parsing component. We also plot the number of memory-safety bugs in the renderer's media parsing component in Figure 5. It is clear that the number of memory-safety bugs in Firefox has shown a small but steady decline since Oxidation started in 2015, with a noticeable drop in memory-safety bugs in the renderer's media component. Despite several attempts from browser vendors to counter memory safety issues, none of them resulted in a high impact like Firefox's Oxidation.

Lesson 1: Using memory safe languages is an effective mitigation against memory-safety bugs.
As shown in Figure 5, the use of Rust in Firefox effectively reduces the memory safety bugs. Though it takes a lot of effort, it is a fundamental way, and the most promising way to eliminate memory safety bugs. We suggest that other browser vendors follow this best practice, and gradually shift their browsers to memory-safe languages.

Bug bounty programs. Major browser vendors such as Google provide rewards for proper security bug reports that help them to fix vulnerabilities [116]. In most cases, these payouts account for multiple factors such as bug type, exploitability, and significant additional effort made by the reporter. Higher payouts indicate higher incentives for researchers and whitehats to find bugs. We correlate the average payout amounts per year to the yearly number of bugs in Chromium in Figure 6. We particularly show memory-safety, UXSS, and URL spoofing bugs since they depict interesting patterns with respect to their payout amounts.

Payout amounts have an influence on the number of bugs found for respective classes (Figure 6). Bug classes that have had average bounty amounts above the overall average amount (*e.g.*, UXSS in 2014-2016 and Mem bugs in 2017-2020) seem to increase in numbers on those exact years. This correlation does not work both ways, however: an increase of the number of bugs of a certain class does not encourage higher bounty amounts. This figure further emphasizes other important factors that guide researchers' efforts while looking for bugs besides seeking higher payouts, such as 1) seeking the perks of exploring uncharted attack vectors (URL spoofing), 2) aiming for bugs with higher impact (UXSS increased in 2016), and 3) avoiding bug classes that have effective mitigations (Site Isolation released in 2017 and UXSS bugs decreased in 2018).

Lesson 2: Higher payouts motivate more bug reports.
Browsers try to increase coverage and payout of bug bounty programs, which led to more bug reports. Therefore, increasing bug bounty payouts can effectively attract the interest of security researchers and reduce attack surfaces.

Divergence of bug severity ratings. The Common Vulnerability Scoring System (CVSS) [46] was developed as a free and open source standard for bug severity assessment. The National Vulnerability Database [55] uses the CVSS standard to provide bug severity base scores for each issued CVE number. Similarly, Firefox and Chromium provide an assessment of a bug's severity in their bug trackers [43] [48] and security advisories [53] but use their own bug rating systems. Table III compares the bug severity assessments of NVD's CVSS-V3 against those of Firefox and Chromium. The aim of this study is to measure the effectiveness of using NVD's CVSS-V3 scores as a unified scale for bug severity in browsers.

In the table, we notice a divergence between the rating systems (Vendor vs. NVD). NVD rates more than half of

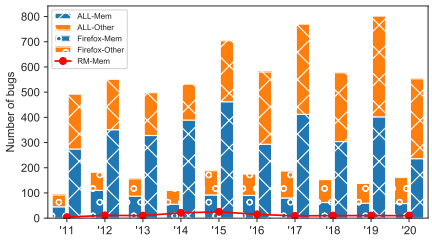


Fig. 5: The number of memory-safety bugs vs. other bugs in Firefox and other browsers. **RM-Mem** is the number of memory-safety bugs in the media parsing component in Firefox's renderer, depicting a decline after it was partially rewritten in Rust starting in 2015.

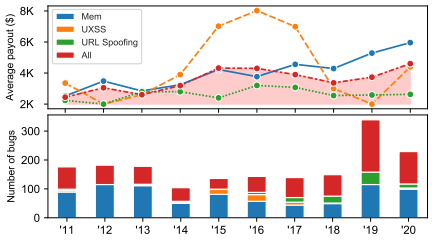


Fig. 6: Correlations between average payout amounts (top chart) to the yearly number of bugs in Chromium (bottom chart). The red area is the average bounty amount for all classes[1]. Bug classes crossing the red area indicate a higher bounty than the average.

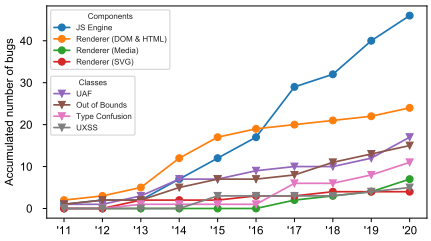


Fig. 7: The trends of browser components and classes of exploited bugs. The data include bugs from all browsers. Lines are the accumulated numbers. The JavaScript engine and UAF bugs are dominating the exploited bug components and classes, respectively.

Browser	Vendor				NVD's CVSS-V3				Total (Diff)
	L	M	H	C	L	M	H	C	
Firefox	**13%**	34%	35%	17%	**1%**	37%	39%	22%	947 (491)
Chromium	**16%**	39%	43%	2%	**1%**	46%	47%	6%	1191 (505)

TABLE III: A comparison between NVD-assigned CVSS-V3 scores vs. vendors' bug severity scores. Bug ratings are: **L**ow, **M**oderate, **H**igh, and **C**ritical. Bugs that are too old[2] to have CVSS-V3 scores are omitted from the table. The last column is the total number of bugs used for this comparison, and the number of bugs that have different bug ratings assigned. The table emphasizes the divergence of bug ratings in two dimensions: 1) CVSS scores vs. vendors' scores as seen in the number of bugs triaged as **L**ow (emboldened), and 2) bugs rated as **C**ritical from vendor to vendor (underlined).

Firefox's bugs with different severity scores, while it is only in agreement with Chromium on 58% of Chromium's bugs. While full agreement between the two rating systems is not expected, the big gap between the ratings is surprising. The divergence between ratings also occurs between vendors. Looking at the number of critical bugs in the two browsers, we can see that Firefox assigns a much higher percentage of its bugs as Critical compared to Chromium. Our analysis results align with previous concerns regarding the use of CVSS scores as a metric for bug triaging and prioritizing [132] [205] [156].

Bugs in browser exploits. Bugs used in real-world browser exploits deserve more attention, as they are indications of favorable attack vectors from the attackers' point of view. To study such bugs, we collect information from credible sources that only acquire highly exploitable bugs. For bugs used in the wild, we refer to Google's routinely updated Project Zero report, which tracks all publicly known cases of zero-day exploits since 2014 [58]. We also collect the bugs exploited in Pwn2Own [40], a real-world hacking competition sponsored by the Zero Day Initiative [61]. We highlight the top exploited bugs in the past decade in Figure 7, based on their bug class and the host browser components.

As shown in Figure 7, for browser components, DOM bugs were dominant until they were overtaken by JS engine bugs in 2017. Nevertheless, DOM bugs remain relevant today and show a slow but steady increase even after adding many mitigations. For bug type, memory-safety bugs such as UAF bugs are still favorable over other bug classes in real-world exploits despite all the mitigations in place. One interesting observation is the trend of emerging bug classes and components. For most of the lines in the figure, we see a rather steep increase in the early years, but the increase slows down afterward (except for JavaScript engine bugs). This trend visualizes the attacker's endless effort to find and explore new attack techniques, and the vendors' reactive countermeasures to eliminate and mitigate new bugs.

Bug types included in this section. Based on the trend of bugs presented in Figure 7, in this section, we mainly discuss the trending types of bugs, namely: 1) Parser bugs (§III-B), 2) DOM bugs (§III-C), 3) JavaScript Engine bugs (§III-D), 4) SOP bypass and UXSS bugs (§III-E).

B. Parser Bugs

Parsers often suffer from memory corruption bugs; there is no exception for parsers in browsers. In web browsers, the majority of parser bugs have been found in media parsers or network protocol parsers. As shown in Figure 4a, *Renderer (Media)* takes a large share. These bugs are easier to exploit in the renderer process since they can be utilized to corrupt JS objects and create more powerful exploit primitives.

Current status. After the hardening of heap allocators (§IV-B), such exploits were made much more difficult or infeasible, mainly because of the compartmentalization of JS objects on the heap. Also, large-scale fuzzers like ClusterFuzz [111] have discovered many parser bugs. Browser vendors are working on sandboxing networking code and rewriting browser code using memory-safe languages like Rust [57]. As a result, these bugs have become scarce and harder to exploit. Still, there are multiple dependencies of third-party libraries when it come to parsing data, so tight control of security updates are needed.

C. DOM Bugs

DOM bugs were popular targets for attackers; according to Figure 7, the majority of exploited bugs in 2014 were DOM bugs. Since most of them were UAF bugs, mitigations have been deployed to reduce the exploitability of them, such as *isolated heap* and *delayed free* §IV-B.

[1]CVE-2011-3046 [3], the largest bounty rewarded by Chromium ($60k), is an outlier that was removed from the figure.

[2]NVD started to use the CVSS-V3 scoring system in 2015 which has four levels of ratings, matching vendors' rating levels, which justifies our comparison.

Current status. While fuzzers continue to identify new DOM bugs [111], [129], [227], as shown in Figure 7, recent known in-the-wild full-chain exploits tend to use bugs in other components due to the increased difficulty of exploiting DOM bugs.

Lesson 3: UAF mitigations are effective towards reducing DOM bug exploits.
Since DOM bugs mostly rely on UAF problems, they have been mostly mitigated by UAF mitigations. Generic exploitation techniques relying on pointer type confusion have become infeasible since heaps are isolated by object types, and there are no publicly known alternative techniques. As a result, exploiting DOM bugs is no longer a preferred way to compromise renderers.

D. JS Engine Bugs

In recent browser exploits, JS engine bugs are one of the most popular targets of browser exploits, especially optimization bugs. At least 34% of exploited bugs (Figure 7) have utilized JS engine bugs for compromising the renderer process, which is usually the first step for full-chain browser exploits. JS engine bugs can be utilized to easily generate powerful exploit primitives like `addrof` (to leak the address of any JS object) and `fakeobj` (to access an arbitrary address as an object).

As mentioned in §II-B, JIT compilers in JS engines use speculative optimization. Bugs in these optimizations are far more critical than conventional memory safety bugs such as use-after-free or buffer-overflow, as they are hard to mitigate but provide powerful exploitation primitives to attackers. On a high level, JS engine bugs can be mainly divided into four categories:

- *JIT analysis bugs:* Bugs in the analysis process or models of the JIT compiler. Such bugs have the highest exploitability and impact.
- *JIT code mutation/generation bugs:* Bugs in the process of manipulating JIT graphs or emitting code. They often result in an outright unexploitable crash.
- *Non-JIT side-effect bugs:* Side-effect bugs in JavaScript built-in functions, which are mostly related to fast-paths.
- *Non-JIT traditional memory corruption bugs:* Other memory corruption bugs that don't fall into the categories above.

We examined the 45 exploited bugs in Figure 7; there are 13 JIT analysis bugs, 9 non-JIT side-effect bugs and 11 traditional memory corruption bugs, but there are no JIT code mutation/generation bugs. We suspect that this is because generation bugs are hard to exploit. Most of the bugs in the JIT compiler are logic bugs. Since it is a compiler infrastructure, logic errors can be amplified to memory safety errors in JIT-compiled code. Therefore, it is hard to make a general mitigation for JIT bugs. Here, we introduce three major categories of defenses: primitive elimination, overwrite protection and jit-based code-reuse mitigations.

Primitive elimination. Primitive elimination techniques aim to prevent attackers from 1) converting vulnerabilities to exploit primitives and 2) escalating exploit primitives to stronger ones[3].

a) Object shape authentication. This type of mitigation aims to prevent attackers from crafting a valid object using the *fakeobj* primitive. For example, in JavaScriptCore, StructureID Randomization encodes the StructureID with seven random entropy bits, which makes it hard for the attacker to guess [121], [204]. Since StructureID indicates the `type` and `shape` of the JS object, incorrectly guessing the StructureID will lead to invalid `shape`, and accessing it will ultimately crash the process [221].

b) Address space isolation. This category of mitigations provides isolation of different objects to prevent objects from being faked or overwritten. GigaCage [83] is a 4GB virtual memory region that separates different objects into different heaps (or `HeapKinds`). The key idea is to prevent memory access across different heaps and use the relative offset from the heap base address to locate a GigaCaged object, instead of using absolute addresses. As such, even if a pointer is corrupted it cannot point to anything outside its original heap. PACCage [121] is applied to protect the backing store buffer pointers of `TypedArray` with Pointer Authentication Codes (PAC) on top of GigaCage to enhance the security even further. Chrome V8 Heap Sandbox [119], which is experimental, has goals similar to GigaCage, but it tries to protect external pointers using a separate pointer table, so that attackers cannot create arbitrary values for external pointers.

Overwrite protection. Overwrite protections are standard protection mechanisms to prevent attackers from introducing arbitrary executable code, which can be seen as the last line of defense in the context of browser exploits. They mainly include four mechanisms: W $\oplus$ X [106], hardened JIT mapping [139], fast permission switch [128], [139], and out-of-process JIT [164].

a) W $\oplus$ X. W $\oplus$ X [106] is an important security principle that enforces memory to be either executable but not writable or writable but not executable. This mitigation made traditional shellcode injection attacks completely obsolete and provided the foundation for many other protection techniques [62], [232]. Surprisingly, JIT code pages are often exempt from this basic mitigation and mapped as `rwx` for performance reasons [106].

b) Execute only memory. iOS 10 on ARMv8 devices landed hardware support for *execute-only memory* (XOM) [139], enabling JIT-compiled code to contain secret data as an immediate value. Safari utilizes XOM to hide the address of the writable-executable mapping from attackers, by introducing an execute-only `jit_memcpy` function that has the base address of JIT mapping inside. This makes arbitrary read/write insufficient for the JIT code page overwrite, and forces attackers to take an alternative path *e.g.*, hijacking control-flow to call `jit_memcpy`.

[3]For example, to construct reliable and stable read/write primitives, an attacker can leverage the *addrof* and *fakeobj* primitives to fake an `ArrayBuffer` object with a fully controlled backing store pointer, which is an escalation of primitives.

c) Fast permission switch: APRR & MPK. Hardware support for fast permission switching was introduced to reduce the overhead of switching page permissions using `mprotect()`. Since iOS 11 on ARMv8 devices, APRR [139] was deployed to enable per-thread permissions by mapping page permissions (`r,w,x`) to eight dedicated registers that indicate the actual page permission of the thread. Similarly, Intel MPK [128] adds a separate 4-bit integer per page, to enforce two additional protections: *disable access* and *disable write*. Consequently, the JIT region will always be `r-x`, and only the write operations from a dedicated data-copying thread are allowed by invoking an `unlock` function, which changes the permission to `rw-` only for the target thread.

d) Out-of-Process JIT. On Windows, mitigations like Arbitrary Code Guard (ACG) ensure that a process can only map *signed* code into its memory. However, browsers heavily use JIT compilers for performance purposes, which generate unsigned native code in a content process. Out-of-process JIT [164] was introduced to enable ACG with JIT compilers. Consequently, the JIT functionality was moved to a separate process that runs in its own sandbox, and it is responsible for compiling JS code and mapping it into the process. As such, the content process is never allowed to map or modify its own JIT code pages.

JIT-based code-reuse mitigations. *JIT spray* [158] is a technique that injects a vast amount of attacker-controlled JIT code (marked as `executable`) into a predictable address in the memory to bypass ASLR/DEP, similar to *Heap spray* [95]. To mitigate JIT spray, browsers put a size limit on JIT code and switched to 64-bit platforms with high-entropy ASLR, which made JIT-spray infeasible. Still, it is possible to utilize the JIT code gadgets if their addresses are known to the attacker. Such attacks are called *JIT-based code reuse attacks* (JCRAs). Here, we briefly summarize mitigations for such attacks.

a) Controlled bytes elimination. JCRAs have a fundamental assumption that with control of immediate operands and specific opcodes, an attacker can control the generated JITed code in heap memory. Therefore, mitigations were proposed to eliminate the predictability of attacker-controlled bytes, such as obfuscating large constants [71], [144], permuting the register allocation of immediate operands and local variables [144], [222] and jumbling the instructions in a function's call frame [144], [222].

b) Internal randomization. Attackers can also leverage the relative location of instructions with each other or predictable offsets from the base address. Some of the mitigations aim to diversify the JIT code layout, including: randomizing the relative offsets between different pairs of instructions [126], [144], [225], and inserting free space randomly before the first unit of code [106], [144].

Current status. While there are some trials to prevent certain types of errors (§IV-D), it's hard to cover all of them. As a result, mitigations in JS engines focus on eliminating attack primitives. Recently, the Edge team added a new security feature called Super Duper Secure Mode (SDSM) [17], [19], which basically disables JIT compilation. Users can choose to disable JIT on websites that are less frequently visited. While sacrificing some performance, it is a good approach for reducing attack surfaces. For JCRAs, although multiple mitigations have been introduced, they are still viable [87], [106] since vendors did not put many resources into implementing or maintaining mitigations.

Lesson 4: Mitigating JS engine bugs is difficult.
JavaScript engine bugs, especially JIT compiler bugs, are very powerful since the attacker can emit code with memory corruption issues. Many mitigations aim to prevent escalation of exploit primitives because it is hard to mitigate logic bugs in general. Therefore, vendors often deploy mitigations that aim to break exploit paths, and enhance them continuously to prevent future attacks.

E. SOP-Bypass and UXSS Bugs

Same origin policy (SOP) [94] is enforced by web browsers to keep a security boundary between different origins. SOP-bypass bugs can be used to compromise SOP to varying degrees, from leaking one bit to stealing full-page data. UXSS bugs are the most powerful type of SOP-bypass bug that can be used to facilitate cross-origin JavaScript code execution. In UXSS attacks, the attacker can inject scripts to any affected context by exploiting bugs in web browsers [13], [14] or third-party extensions [31], [35], achieving the same effect as exploiting the XSS vulnerability in the target website.

Current status. *Site Isolation* [39], [186] is one of the most significant mitigations against UXSS attacks. Site Isolation enforces SOP at the process level, which made most existing UXSS bugs unexploitable. The number of reported UXSS bugs was significantly reduced after site isolation was gradually applied after 2017, as shown in Figure 6. However, UXSS vulnerabilities in third-party extensions still exist; multiple UXSS bugs have been found in popular extensions [31], [35], which have enabled attackers to bypass site isolation and steal the user's credentials.

Lesson 5: UXSS bugs are mostly mitigated by Site Isolation.
Site isolation is an effective mitigation against UXSS bugs. However, only Chrome and Firefox have site isolation deployed, since it requires a considerable amount of engineering effort (Appendix D).

F. Summary

Due to threat research and improved patch deployments, the impacts of 1-day exploits are reduced, and in-the-wild 0-day exploits get patched quickly once they are caught. However, offensive research is still much ahead of the vendors. Although vendors are trying, they are consistently behind in this arms race. Mitigations from vendors are mostly reactive, which means they are developed long after each wave of attacks. By the time an attack surface is finally closed, attackers have already come up with a better exploit. It's a difficult task, but vendors should be more proactive and implement new features with security implications in mind, *e.g.*, studying potential new attacks before deploying new features.

IV. More Security Mitigations in Browsers

In this section, we present more generic mitigations implemented by browser vendors that are not covered in previous sections. We present a longitudinal study on the mitigations implemented in the four major browsers in the past decade, as well as the dates when they were applied and retired, in Table IV. In this section, we discuss a few of them in detail.

A. Sandbox

The sandbox is crucial to browser security because it confines the effect of bugs in the renderer process, which contains various error-prone components. Except for cases like UXSS, attackers need to escape from the renderer sandbox using an exploit for the kernel, system services, or the browser process. Consequently, it significantly raises the bar for attacks because attackers need to exploit both components (renderer and sandbox) to have a full-chain 0-day exploit.

Win32k lockdown. Since most Windows kernel vulnerabilities have been in Win32k system calls, Microsoft introduced the System Call Disable Policy aka. Win32k lockdown [130] for Windows in 2012. This allows the developer of a Windows application to completely block access to the Win32k system call table, significantly reducing the attack surface. Edge, Chrome, and Firefox have already adopted this mechanism to protect the browsers. As a result, achieving sandbox escape from the renderer process has become much more complex.

Hypervisor based sandboxing. Windows Defender Application Guard (WDAG) [161] was introduced by Microsoft to isolate untrusted websites or resources (*e.g.*, files) in enterprise scenarios. WDAG uses Hyper-V to create a fresh instance of Windows at the hardware layer, which includes a separate copy of the kernel and the minimum Windows Platform Services to make sure that the Edge browser functions normally. WDAG is implemented in Edge to protect against advanced attacks that can bypass the browser sandbox. With WDAG, the attacker needs to escape both the sandbox and the Hyper-V virtual machine.

B. Hardened Allocators

Browsers use specialized heap allocators for many objects for performance and security reasons [96], [150]. These allocators use specific designs which help reduce damage by limiting attack primitives.

Isolated heap. Isolated heap is an effective defense to prevent use-after-free (UAF) attacks being escalated to type confusion attacks. By isolating objects based on their 1) type, 2) security hazard level (*e.g.*, embedding v-table pointer), and 3) JavaScript reachability (*e.g.*, `ArrayBuffer`), isolated heap effectively raises the bar for UAF exploitation. The isolation prevents an attacker from re-claiming the freed object with an object with a different layout, which is typical for exploiting UAFs in browsers.

Modern browsers implement a basic level of heap separation between JavaScript-reachable objects and other objects [83], [85], [100], [123]. However, it was still possible to create type confusion via UAF among the objects in the same heap but in other types. To prevent this attack, Safari [73] and Firefox [98] introduced separate heaps for every type in specific categories, which provided a much more fine-grained isolation. Therefore, there is no public, generic exploitation methodology for exploiting UAF bugs in all browsers.

Delayed free. Another mitigation, *elayed free*, effectively increases the difficulty of exploiting UAF bugs, but this approach cannot restrict the reclamation of dangling pointers. Browsers use various garbage collection (GC) algorithms to deallocate heap-allocated objects with no references. Some variants of GC additionally scan stack and heap [96], [150] areas to find possibly overlooked references, which is known as *conservative scanning* [80] or *delayed free* [123]. Notably, Firefox dropped this in favor of *exact rooting* and wrote a static analysis tool to find unsafe usage of references from the stack [99], [167]. Chrome also has a similar tool [15], but it is only enforced on specific areas. However, *delayed free* has introduced side-channel primitives that can be used to defeat ASLR since it cannot distinguish pointers to the heap and user-controlled integers by design [41], [81], [104], [123].

Heap metadata protection. Heap metadata protection is an approach that checks the metadata portion of heap chunks to prevent heap corruption and silent error propagation in the heap. For example, a heap allocator may put a random value [151] before dangerous data structures to detect heap exploits. *PartitionAlloc* in Chrome removed in-line metadata and placed guard pages to prevent linear heap overflow from overwriting metadata [85]. There are also some OS-level efforts on metadata protection [148], [151].

Other mitigations on heap. *Frame poisoning* in Firefox deallocated chunks of memory with addresses pointing to non-accessible memory pages [98]. Similarly, in Edge, this is done by filling zeros when freeing heap chunks [228]. *GWP-ASan* [115] in Chrome randomly places a small portion of allocated objects right before/after guard pages and deallocates the entire page when the chunk is freed to detect heap errors in the wild.

C. Control-Flow Integrity

Since attackers often manipulate the values of instruction pointers to achieve code execution, control-flow integrity is enforced to prevent them from hijacking control flows, making the attack more difficult. The compiler infrastructure, OS and hardware support provide most mitigations in this category, such as protecting virtual function tables by introducing canary values [165] and allowing listing indirect branches by checking the destination address [159], [177].

There is ongoing work to prevent arbitrary memory writes from modifying code regions that are executable by attackers (§III-D). Based on hardware support, browsers could apply additional mitigations without a dramatic decrease of performance, such as adding pointer integrity checks using PAC on ARM64 [149] and adding additional W $\oplus$ X protection on JIT-compiled code using Intel MPK [118] and APRR [198].

★: Mitigation applied, ✗: Mitigation retired
C: Chrome, **F**: Firefox, **S**: Safari, **I/E**: IE / Edge

Attack		↓Bypass OS mitigations [1], [2] ↓Bypass SELinux [20]				↓Bypass CFG [229] ↓Sandbox escape [4]						↓JS [26] ↓JS core [5]				↓S/M [137]				↓Glitch [105] ↓Code sign [120]									
Mitigation		Browser				2010				2011				2012				2013				2014				2015				2016			
		C	F	S	I/E	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
JIT	StructureID randomization [204]			✓																											
	GigaCage [83]			✓																											
	PACCage [121]			✓																											
W⊕X	W⊕X protection [21]	✓	✓	✓	✓						★			★								★				★						
	Hardened runtime [68]	✓	✓	✓																											
	Out-of-process JIT [164]				✓																										
	Code signing enforcement [25], [102], [164]	✓	✓		✓																										
	Arbitrary code guard [32], [101], [164]	✓	✓		✓																										
Heap	Isolated heap [83], [85], [100], [228]	✓	✓	✓	✓														★				★								
	Delayed free [96], [99], [228]	✓	✓	✓	✓		★															✗		★			★			★		
S/M	Branchless security checks [97]	✓		✓	✓																										
	Disabling SharedArrayBuffer [97]	✓	✓	✓	✓																										
	Reducing timer accuracy [86], [171], [209]	✓	✓	✓	✓																										
CFI	VTGuard [228]				✓										★																
	Control flow integrity [159], [177]	✓	✓		✓																		★								
SBX	Seccomp (+bpf) sandbox [146]	✓	✓			★													★												
	Namespace (setuid, userns) [74]	✓	✓			★																		★							
	SELinux [200]	✓					★											✗													
	macOS sandbox [140]	✓	✓	✓		★																★									
	Restricted tokens [18]	✓	✓		✓	★	★																★								
	AppContainer [228]				✓											★															
	Non-system font filtering [23]	✓																									★				
	Win32k lockdown [130]	✓																										★			
	Plugin sandboxing (*e.g.*, Flash) [188]	✓	✓	✓	✓					★				★	★				★												
Other	XSS auditor [72]	✓		✓	✓	★	★																								
	64-bit ASLR	✓	✓	✓	✓					★				★				★				★									
	Hypervisor based sandboxing [160], [162]				✓																							★			
	Site isolation [186]	✓																											★		
	Pointer authentication code [149]	✓		✓																									★	★	

Attack		↓Glitch [105] ↓Code sign [120]																													
Mitigation		2017				2018				2019				2020																	
		1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4														
JIT	StructureID randomization [204]												★																		
	GigaCage [83]			★																											
	PACCage [121]										★																				
W⊕X	W⊕X protection [21]																														
	Hardened runtime [68]			★			★				★																				
	Out-of-process JIT [164]	★																													
	Code signing enforcement [25], [102], [164]	★								★		★																			
	Arbitrary code guard [32], [101], [164]	★								★	★																				
Heap	Isolated heap [83], [85], [100], [228]			★		★																									
	Delayed free [96], [99], [228]																														
S/M	Branchless security checks [97]					★	★	★	✗	✗																					
	Disabling SharedArrayBuffer [97]			★		★	★	★	✗								✗														
	Reducing timer accuracy [86], [171], [209]					★	★	★																							
CFI	VTGuard [228]																														
	Control flow integrity [159], [177]	★				★																									
SBX	Seccomp (+bpf) sandbox [146]																														
	Namespace (setuid, userns) [74]																														
	SELinux [200]																														
	macOS sandbox [140]																														
	Restricted tokens [18]																														
	AppContainer [228]																														
	Non-system font filtering [23]																														
	Win32k lockdown [130]			★																											
	Plugin sandboxing (*e.g.*, Flash) [188]																														
Other	XSS auditor [72]												✗																		
	64-bit ASLR																														
	Hypervisor based sandboxing [160], [162]																														
	Site isolation [186]					★																									
	Pointer authentication code [149]						★	★																							

TABLE IV: Mitigations in browsers.

D. Side Channels

Browsers are also vulnerable to side-channel attacks. To date, studies have shown that sensitive information in browsers can be inferred via 1) microarchitectual state [137], [147], [157], [179], [197]; 2) GPU [142], [178]; 3) floating-point timing channels [64] and 4) browser-specific side channels [201], [216]–[218]. Researchers have introduced defense mechanisms [84], [138], [194], [202] to protect the browsers from side-channel attacks, such as DeterFox [84] and FuzzyFox [138]. Also, browser vendors have implemented defenses [97], [114], [127], which can be classified into two categories as follows:

Reduce the resolution of timers. Since most of the attacks rely on accurate timing, to hinder the detection of small timing differences, browser vendors reduced the resolution of the precise timer (*e.g.*, `performance.now()`) and introduced random jitters to prevent resolution recovery [22], [66], [168]. After the discovery of Spectre [137] and Meltdown [147], the vendors further lowered the precisions of timers [86], [171], [209]. Since `SharedArrayBuffer` can be used to create a high-resolution timer, shortly after the discovery of Spectre [137], `SharedArrayBuffer` was disabled in all modern browsers [88].

Prevent resource sharing. Another mitigation technique is to prevent resource sharing between the victim and the attacker. Site Isolation [39], [186] (§II-B) effectively mitigates the Javascript-based transient execution. Cross-Origin-Opener-Policy (COOP) and Cross-Origin-Embedder-Policy (COEP) [131] were introduced to set up a cross-origin isolated environment. COOP allows a website to include a response header on a top-level document, ensuring that the cross-origin documents do not share the same browsing context group with itself, thus preventing direct DOM access. COEP prevents a page from loading any cross-origin resources that do not give explicit permission to this page. These are both enforced using HTTP headers, and they were shipped in Chrome 83, Firefox 79, and Edge 83 [92], [93], while Safari does not support them yet as of November 2021.

After the introduction of Site Isolation and COOP/-COEP, Chrome and Firefox re-enabled the use of `SharedArrayBuffer` [27], [172]. However, a study showed that the re-introduction of `SharedArrayBuffer` increases the covert channel capacity 2,000-fold and 800,000-fold, respectively [187]. Two recent papers [63], [135] suggested that despite Site Isolation in Chrome, the attacker can still learn cross-origin sensitive information.

E. Other Mitigation Efforts

UAF mitigation. To fundamentally fix the UAF problems that are not covered by garbage collection or other safety measures, the Chrome team introduced a term called *MiraclePtr* [184], [185], which stands for a set of algorithms that can wrap raw pointers in C/C++ so that they are not exploitable via UAF. *MiraclePtr* is expected to be in production soon [79].

Improving memory safety. The Chrome team has explored improvements for their C++ codebase that can eliminate/reduce

specific types of bugs by limiting the use of specific language features (*e.g.*, C++ exceptions [112]) and introducing wrapper classes around integer operations [212].

Improving JIT compiler. There have been efforts to safeguard dangerous optimizations inside JIT compilers. For example, many exploits make use of bounds check elimination [192] that removes seemingly redundant bounds checks. To mitigate this, the Chrome team introduced a patch that marked such checks as *aborting* instead of simply removing them [133]. Therefore, the attacker can only trigger a `SIGTRAP` at best. Moreover, to make bytecode generation for standard JS functions less error-prone, the Chrome team made a domain-specific language, *Torque* [113], which replaced the existing C++ implementations and reduced a lot of LoC.

Lesson 6: Collaborative efforts on mitigations are good. When one vendor deploys a mitigation, other vendors are likely to follow. In Table IV, we saw that most of the mitigations have been adopted by multiple browsers. If there are bugs found in one browser, the vendor can quickly share the information with other vendors and they can work together to build better mitigations using collective knowledge. In the case of Spectre/Meltdown attacks [137], [147], browser vendors worked together to build a plan for mitigating the immediate threats [86], [171], [209], which is a great example of collaborative effort.

V. Case Study: Full-chain Exploits

Because modern browsers are heavily compartmentalized with different security capabilities, browser exploitation often requires chaining multiple attacks to ultimately execute malicious action. Combining all such steps is usually referred to as *full-chain exploitation*. As a representative case study for full-chain browser exploitation, we analyze a winning attack against Safari [134] in 2020 Pwn2Own competition, *i.e.*, the *exploit example* shown in Figure 2.

This attack infiltrates the renderer process, starting from a JIT compiler optimization error [37]: The DFG compiler in Safari JavaScript renderer incorrectly models a side effect of *in* operator when a special condition regarding *proxy object* is met. This bug allows the players to construct the standard *addrof/fakeobj* primitives, which yields arbitrary memory read/write and ultimately, arbitrary code execution. To construct a valid object using *fakeobj*, the players utilize a publicly known technique [221] to bypass *object shape authentication* (*StructureID randomization* in §III-D). After faking a JavaScript object, they use a known technique [183] to bypass Address Space Isolation (*Gigacage* in §III-D) and get an arbitrary read/write primitive in the renderer process.

Once the renderer process is compromised, sandbox escaping is the next step and is more challenging. In this attack, the players cleverly stitch multiple logic/memory errors together to escape the sandbox. The players first additionally obtain arbitrary code execution from the *CVMServer* XPC service (part of the built-in OpenGL framework), which, though sandboxed, has the capability to create *symbolic link*, while the renderer process does not have such capability. Also, there is an IPC method in Safari, `didFailProvisionalLoad()`, that can launch an arbitrary app if a symbolic link pointing to the app folder is provided. By combining them, the players can launch arbitrary apps via Safari. At this point, the sandbox is successfully breached, as they can execute arbitrary applications outside the renderer sandbox, similar to a user who launches Safari.

The Pwn2Own example we summarized is specific but impactful. Based on this, we describe the full-chain browser exploitation in a more generic way. First, to find vulnerabilities in the renderer, one can leverage fuzzing techniques [122], [125], [182] or manually audit the browser source code. Discovering an exploitable bug would be one of the most challenging steps. After such a bug is found, the next step is to achieve an arbitrary code execution primitive within the renderer process context. However, taking control over the renderer is only a beginning, since renderers are confined by the *sandbox* mechanism. To break out of the sandbox, the attacker typically targets flaws in the browser process, the OS kernel, or IPC protocols. Unlike attacking the renderer, sandbox escape usually requires chaining high-level logical exploits against multiple system components. Once the sandbox is escaped, the attacker can execute an arbitrary program with an equal security level as the browser, and full-chain exploit is achieved.

VI. Discussion

In this section, we discuss several aspects related to browser security. There are more discussions in the Appendix.

A. Patch-gapping Problems

Due to the existence of public repositories and issue trackers, patches in open source browsers can be published before a new release is done and made available to end-users, enabling attackers to assess the exploitability of patches. For example, iOS Safari was exploited due to the 1.5-month patch gap [193]. To shrink the gap, Chrome introduced bi-weekly security updates and reduced the release cycles from six weeks to four weeks [78]. Firefox holds back pushing security fixes to the repository before releases [173], [193] and recommends not including vulnerability information in patch commits [173].

B. Homogeneity of Browser Engines

Many secondary browsers use the same browser engine as the leading browsers (*e.g.*, Chrome V8). As a result, a vulnerability in one browser engine can affect other browsers that share it. Among the 15 most popular browsers [206], 11 of them are based on Chrome's engine (including Microsoft Edge [136]), as shown in Table V. When a new version of Chrome is released with bug fixes, it is not applied immediately to secondary browsers since there is a time gap before secondary browsers integrate them.

According to the release history of secondary browsers there are time gaps before applying released security patches, which provides an attack window for the attacker. For example, one WebKit bug was exploitable on PlayStation firmware several months after being reported to the WebKit bug tracker [29]. This was also an issue in Android, where apps are shipped with

Engine	Browser	UI Engine	Server App
Chrome	Chrome, Edge, Opera, UC, Android, 360Safe, QQ, Yandex, Whale, Puffin, KaiOS	Electron, Android WebView, Qt WebEngine	Node.js, CouchDB
Safari‡	Chrome, Safari, Edge, Opera, UC, QQ, Whale, Puffin	iOS WebView, Qt WebKit	
Firefox	Firefox		MongoDB
IE/Edge	Edge Legacy, IE†, QQ†, Whale†	MSHTML	
Total	24	6	3

TABLE V: Homogeneity of browser engines. Some browsers ship multiple engines to ensure compatibility of web pages (†) or due to specific platform requirements, such as WebKit on iOS(‡) [69].

bundled rendering engines *e.g.*, a UXSS bug was reported on Samsung Internet [6] around one month after being reported to Chromium [7]. Apple solved this problem in iOS by enforcing all apps to use WebKit libraries provided by the OS and rejecting non-compliant apps in their App Store [69].

Moreover, the use of web browser components such as renderers and JavaScript engines further extends to applications using frameworks such as Electron and Android WebView. Also, Node.js [56] and Deno [47] utilize Google's V8 engine to enable JavaScript outside the context of browsers (*e.g.*, for implementing web servers). As a result, bugs and exploitations of browser engines have a broader impact beyond just browsers themselves, expanding the need for better defense mechanisms to avoid catastrophic consequences.

Lesson 7: The homogeneity of browser engines creates a serious problem; better patching approaches are needed. Due to the homogeneity of browser engines, browser bugs in one browser engine can affect many other browsers and applications. We suggest leading browsers such as Chrome provide their JavaScript engine as a shared library for other apps to use, so that it is easier to deploy patches via over-the-air updates, instead of manually integrating patches.

C. Bug-finding Tools

Multiple efforts have been made to develop state-of-the-art tools for finding browser engine bugs, which can be mainly divided into two categories: fuzzing and static analysis.

Fuzzing. Fuzzing is one of the most effective strategies for finding bugs and has been applied to uncover browser bugs since 2012. We summarize the papers about browser fuzzers in the past decade in Table VI (Appendix), which includes the bug statistics they have found in Chrome, Firefox, Safari, and Edge (based on both ChakraCore and V8), along with their key techniques. These fuzzers choose between two classic modes: mutational fuzzing (*e.g.*, Montage [143]) and generational fuzzing (*e.g.*, CodeAlchemist [122]). Some fuzzers like LangFuzz [125] and DIE [182] leverage a mix of both modes coupled with coverage feedback. Constructing syntactic and semantic aware inputs like DIE [182] and LangFuzz [125] is useful for generating more crashes. Some industrial efforts on fuzzing browsers are highly effective on finding complex browser bugs. For example, ClusterFuzz [111] runs on over 25,000 cores [109] and found over 29,000 bugs [110] in Chrome.

Static analysis. Recently, there has been another line of work in the fuzzing-dominated field of browser bug finding. SYS [82], a first-of-its-kind static/symbolic tool for finding bugs in browser code, showed that static analysis could be scaled for the huge codebases of browsers by breaking them into small pieces. Specifically, SYS uses static checkers to find potential error sites and then uses their extensible symbolic execution to analyze those error sites. Therefore, SYS has highlighted an excellent direction for future works in the field of browser bug finding by static analysis.

Lesson 8: Automated bug-finding tools are great, but they still need improvement. State-of-the-art fuzzers from industry are doing a good job of capturing bugs in browsers. However, despite their good performance, such tools still cannot replace manual audits, which remain the dominant approach for finding complex logic bugs. Thus, more advanced bug-finding techniques are needed from academia as well as industry.

D. Proactive Mitigations

Most existing mitigations are reactive, meaning they are implemented after an exploit has been found, which is not good enough. It would be ideal if a mitigation could be in place before the attack happens (proactive approach), which can defeat unknown threats. For example, Site Isolation [186] was originally designed to mitigate UXSS attacks using out-of-process *iframes*, but it also helped defeat the Spectre/Meltdown attacks, which were found by the researchers long after the Site Isolation project started. This is a good example of a proactive approach against unknown threats.

In the game of exploit mitigations, defenders can never beat attackers because the actions of the defenders are transparent to attackers. Vendors can change this situation by secretly deploying new mitigations, for example, in their sandboxes in a safe browsing infrastructure. This can also help to detect in-the-wild exploits and kill bugs by collecting samples that are highly likely to be malicious. Also, vendors can try more aggressive mitigations that are likely to affect user experiences in such an environment. For example, if *StructureID randomization* (§III-D) was deployed in a safe browsing sandbox before public announcement, most JIT exploits involving the `fakeobj` primitive would have been detected.

VII. Conclusion

In this paper, we present the first SoK of browser security. We first provide a unified model to study the security design of four major browsers, and present a 10-year longitudinal study of browser bugs to study the trends. Then we introduce prominent bug types, and present state-of-the-art mitigations. We also study a real-world full-chain exploit from Pwn2Own 2020 in detail. This paper sheds light on the area of browser security, and presents key takeaways that can enlighten researchers and browser vendors on future directions to improve browser security.

References

[1] CVE-2003-1048. Double free vulnerability in mshtml.dll.

[2] CVE-2006-5579. Access of previously freed memory in Internet Explorer 6.

[3] CVE-2011-3046. Universal XSS in Chromium with largest reward amount ($60k).

[4] CVE-2013-6632. Memory corruption leads to sandbox escape in Chrome browser.

[5] CVE-2016-4622. Remote code execution on WebKit.

[6] CVE-2017-17859. SOP Bypass on Samsung Internet referred as CVE-2017-5124 in Chromium Engine.

[7] CVE-2017-5124. SOP Bypass on Google Chrome.

[8] CVE-2019-5647. Insufficient Session Expiration in Chrome Plugin.

[9] CVE-2019-6481. Second-Factor Auth Bypass in Chrome Plugin.

[10] CVE-2020-15655. Bypass same-origin policy in Firefox extension.

[11] CVE-2020-6554. Use After Free in Chrome extension.

[12] CVE-2020-6809. Arbitrary read on local files in Firefox extension.

[13] CVE-2021-1879. UXSS in Webkit iOS 14.4.2 and iPadOS 14.4.2.

[14] CVE-2021-34506. Microsoft Edge (Chromium-based) Security Feature Bypass Vulnerability.

[15] gcmole. https://github.com/v8/v8/tree/master/tools/gcmole.

[16] Languages - Chromium (Google Chrome). https://www.openhub.net/p/chrome/analyses/latest/languages_summary.

[17] Microsoft unveils 'Super Duper Secure Mode' in latest version of Edge. https://portswigger.net/daily-swig/microsoft-unveils-super-duper-secure-mode-in-latest-version-of-edge.

[18] Restricted Tokens - Win32 apps. https://docs.microsoft.com/en-us/windows/win32/secauthz/restricted-tokens.

[19] Super Duper Secure Mode. https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/.

[20] SELinux leaked file descriptor, 2010. https://bugzilla.redhat.com/show_bug.cgi?id=581256.

[21] Add an option to mark JIT pages as non-writable. https://bugzilla.mozilla.org/show_bug.cgi?id=977805, 2014.

[22] Reduce resolution of performance.now to prevent timing attacks. https://bugs.chromium.org/p/chromium/issues/detail?id=506723, 2015.

[23] Add New Process Mitigation Policies for Win10+. https://bugs.chromium.org/p/chromium/issues/detail?id=504006, 2016.

[24] Security: Block GPU Process Opening Renderer Processes. https://bugs.chromium.org/p/chromium/issues/detail?id=596862, 2016.

[25] Enable new FORCE_MS_SIGNED mitigation, 2017. https://bugs.chromium.org/p/chromium/issues/detail?id=750886.

[26] Off-by-one causes JIT optimization error, 2017. https://bugs.chromium.org/p/chromium/issues/detail?id=762874.

[27] Re-enable sharedarraybuffer + atomics. https://bugs.chromium.org/p/chromium/issues/detail?id=821270, 2018.

[28] Sandbox the network service on Windows, 2018. https://bugs.chromium.org/p/chromium/issues/detail?id=841001.

[29] setAttributeNodeNS UAF Write-up. https://github.com/Cryptogenic/Exploit-Writeups/blob/master/WebKit/setAttributeNodeNS%20UAF%20Write-up.md, 2018.

[30] ChakraCore, The core part of the Chakra JavaScript engine that powers Microsoft Edge, 2019. https://github.com/microsoft/ChakraCore.

[31] Comply with new security requirements for Chrome, 2019. https://github.com/uBlockOrigin/uBlock-issues/issues/710.

[32] Enable ACG for jitless v8 in pdfium, 2019. https://bugs.chromium.org/p/chromium/issues/detail?id=961831.

[33] JavaScriptCore, The built-in JavaScript engine for WebKit, 2019. https://trac.webkit.org/wiki/JavaScriptCore.

[34] SpiderMonkey, JavaScript engine for Mozilla products, including Firefox, 2019. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[35] UXSS bug found in Lastpass, 2019. https://blog.lastpass.com/2019/09/lastpass-bug-reported-resolved/.

[36] V8, Open source JavaScript and WebAssembly engine for Chrome and Node.js, 2019. https://v8.dev/.

[37] 2020. Incorrect JIT modeling in WebKit that leads to type confusion.

[38] Add support for random restricted SID. https://chromium-review.googlesource.com/c/chromium/src/+/2085751, 2020.

[39] The Chromium Projects. Site Isolation, 2020. https://www.chromium.org/Home/chromium-security/site-isolation.

[40] Zero Day Initiative - Pwn2Own Returns to Vancouver for 2020. https://www.zerodayinitiative.com/blog/2020/1/8/pwn2own-returns-to-vancouver-for-2020, 2020.

[41] Bypassing ASLR using Oilpan's conservative garbage collector. https://bugs.chromium.org/p/chromium/issues/detail?id=1144662, 2021.

[42] Can I use... Support tables for HTML5, CSS3, etc. https://caniuse.com/, 2021.

[43] Chromium Bug Tracker, 2021. https://bugs.chromium.org/p/chromium/issues/list.

[44] Chromium Security - The Chromium Projects, 2021. https://www.chromium.org/Home/chromium-security.

[45] Chromium source code, 2021. https://chromium.googlesource.com/chromium/src.

[46] Common Vulnerability Scoring System SIG, 2021. https://www.first.org/cvss/.

[47] Deno - A modern runtime for JavaScript and TypeScript, 2021. https://deno.land/.

[48] Firefox Bugzilla, 2021. https://bugzilla.mozilla.org/home.

[49] Google's Project Zero bug tracker, 2021. https://bugs.chromium.org/p/project-zero/issues/list?q=&can=1.

[50] Memory Safety in Chromium, 2021. https://www.chromium.org/Home/chromium-security/memory-safety.

[51] Modern security protection for vulnerable legacy apps, 2021. https://docs.microsoft.com/en-us/deployedge/microsoft-edge-security-iemode-safer-than-ie.

[52] Mozilla Firefox source code, 2021. https://hg.mozilla.org/.

[53] Mozilla Foundation Security Advisories, 2021. https://www.mozilla.org/en-US/security/advisories/.

[54] Mozilla Security Blog, 2021. https://blog.mozilla.org/security/.

[55] National Vulnerability Database, 2021. https://nvd.nist.gov/.

[56] Node.js, 2021. https://nodejs.org/en/.

[57] Oxidation, 2021. https://wiki.mozilla.org/Oxidation.

[58] Project Zero: 0day "In the Wild", 2021. https://googleprojectzero.blogspot.com/p/0day.html.

[59] Smart Pointer Guidelines in Firefox, 2021. https://firefox-source-docs.mozilla.org/dom/workersAndStorage/CodeStyle.html#plain-pointers.

[60] V8 source code, 2021. https://chromium.googlesource.com/v8/v8.

[61] Zero Day Initiative - Published advisories, 2021. https://www.zerodayinitiative.com/advisories/published/.

[62] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *TISSEC 09*.

[63] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking chrome strict site isolation via speculative execution. In *SP22*.

[64] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *SP15*.

[65] Apple. Changeset 160983 in webkit. https://trac.webkit.org/changeset/160983/webkit, 2013.

[66] Apple. Changeset 186208 in webkit for trunk/source/webcore/page/performance.cpp. http://trac.webkit.org/changeset/186208/webkit/trunk/Source/WebCore/page/Performance.cpp, 2015.

[67] Apple. Changeset 253098 in webkit. https://trac.webkit.org/changeset/253098/webkit, 2015.

[68] Apple. Allow Execution of JIT-compiled Code Entitlement, 2018. https://developer.apple.com/documentation/bundleresources/entitlements/com_apple_security_cs_allow-jit.

[69] Apple. App Store Review Guidelines. https://developer.apple.com/app-store/review/guidelines/, 2020.

[70] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS19*.

[71] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. In *NDSS15*.

[72] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *WWW10*.

[73] Niklas Baumstark. Compressing Type Information in Modern C++ Programs using Type Isolation, 2019.

[74] Eric W Biederman and Linux Networx. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*.

[75] Dionysus Blazakis. The apple sandbox. In *Black Hat DC 11*.

[76] blink-dev Google Groups. Intent to implement: Shared array buffers. https://groups.google.com/a/chromium.org/g/blink-dev/c/d-0ibJwCS24, 2015.

[77] Google Chromium Blog. Recent Site Isolation improvements. https://blog.chromium.org/2019/10/recent-site-isolation-improvements.html, 2019.

[78] Google Chromium Blog. Speeding up Chrome's release cycle. https://blog.chromium.org/2021/03/speeding-up-release-cycle.html, 2021.

[79] Google Security Blog. An update on memory safety in chrome. https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html, 2021.

[80] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*.

[81] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *SP16*.

[82] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. In *USENIX Security 20*.

[83] Sam Brown. Some brief notes on webkit heap hardening. https://labs.f-secure.com/archive/some-brief-notes-on-webkit-heap-hardening/, 2018.

[84] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *CCS17*.

[85] R. Chris. Partitionalloc - a shallow dive and some rand, 2016. https://struct.github.io/.

[86] Chromium. Clamp performance.now() to 100us. https://chromium-review.googlesource.com/c/chromium/src/+/853505, 2018.

[87] Chromium. Security: Constant blinding bypass via Wasm, 2020.

[88] ComputerWorld. Browser makers build bulwarks to stump spectre attacks. https://www.computerworld.com/article/3246210/browser-makers-build-bulwarks-to-stump-spectre-attacks.html, 2018.

[89] Crispin Cowan. Strengthening the Microsoft Edge Sandbox, 2017.

[90] Cure53. Cure53 Browser Security White Paper. https://github.com/cure53/browser-sec-whitepaper, 2017.

[91] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases.

[92] MDN Web Docs. Cross-origin-embedder-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy, 2021.

[93] MDN Web Docs. Cross-origin-opener-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy, 2021.

[94] MDN Web Docs. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, 2021.

[95] eEye Digital Security. Microsoft Internet Information Services Remote Buffer Overflow, 2001.

[96] Filip Pizlo. Introducing Riptide: WebKit's Retreating Wavefront Concurrent Garbage Collector. 2017.

[97] Filip Pizlo. What Spectre and Meltdown Mean For WebKit. 2018.

[98] Firefox. Always poison deallocated objects in the frame arena, 2009. https://bugzilla.mozilla.org/show_bug.cgi?id=497495.

[99] Firefox. Exact stack rooting, 2014. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals/GC/Exact_Stack_Rooting.

[100] Firefox. [meta] store content-controlled buffers in a separate heap, 2014. https://bugzilla.mozilla.org/show_bug.cgi?id=1052575.

[101] Firefox. Implement win/osx sandboxing for new RDD process, 2018. https://bugzilla.mozilla.org/show_bug.cgi?id=1498624.

[102] Firefox. Enable Code Integrity Guard on RDD Process, 2019. https://bugzilla.mozilla.org/show_bug.cgi?id=1563774.

[103] Fortinet. Microsoft MSHTML Remote Code Execution Vulnerability Exploited in the Wild (CVE-2021-40444). https://www.fortinet.com/blog/threat-research/microsoft-mshtml-remote-code-execution-vulnerability-exploited-in-wild-cve-2021-40444, 2021.

[104] Ivan Fratric. Dude, where's my heap? 2015.

[105] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *SP18*.

[106] Robert Gawlik and Thorsten Holz. Sok: Make jit-spray great again. In *WOOT18*.

[107] Guang Gong. Security: Pwn2Own mobile case, out-of-bound access in json stringifier. In *Chromium Bug Tracker*, 2015.

[108] Guang Gong. Pwn a nexus device with a single vulnerability. In *CanSecWest*, 2016.

[109] Google. Open sourcing clusterfuzz. https://security.googleblog.com/2019/02/open-sourcing-clusterfuzz.html.

[110] Google. Scalable fuzzing infrastructure. https://github.com/google/clusterfuzz.

[111] Google. Clusterfuzz. https://google.github.io/clusterfuzz/, 2015.

[112] Google. Google C++ Style Guide, 2017.

[113] Google. Torque: Applying leverage to the CodeStubAssembler, 2018.

[114] Google. Chrome - Mitigating Side-Channel Attacks, 2019. https://www.chromium.org/Home/chromium-security/ssca.

[115] Google. GWP-ASan: Sampling heap memory error detection in-the-wild. https://sites.google.com/a/chromium.org/dev/Home/chromium-security/articles/gwp-asan, 2019.

[116] Google. Chrome Vulnerability Reward Program Rules. https://www.google.com/about/appsecurity/chrome-rewards/, 2021.

[117] Google. Chromium design docs - sandboxing. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/design/sandbox.md, 2021.

[118] Google. wasm: Write-protection of generated code with PKEYs/PKU, 2021.

[119] Google Chrome Team. V8 Heap Sandbox - High-Level Design Doc. https://docs.google.com/document/d/1FM4fQmIhEqPG8uGp5o9A-mnPB5BOeScZYpkHjo0KKA8/, 2021.

[120] Samuel Groß. Pwn2Own 2018: Safari + macOS, 2018. https://github.com/saelo/pwn2own2018.

[121] Samuel Groß and Project Zero. Jitsploitation ii: Getting read/write. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-two.html, 2020.

[122] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS19*.

[123] Abdul-Aziz Hariri, Brian Gorenc, and Simon Zuckerbraun. Abusing Silent Mitigations: Understanding weaknesses within Internet Explorer's Isolated Heap and MemoryProtection. In *Black Hat USA 15*.

[124] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. Sofi: Reflection-augmented fuzzing for javascript engines. In *CCS21*.

[125] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security 12*.

[126] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Librando: transparent code randomization for just-in-time compilers. In *CCS13*.

[127] Jann Horn. Mozilla Foundation Security Advisory 2018-01, 2018. https://www.mozilla.org/en-US/security/advisories/mfsa2018-01/.

[128] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals, 2021.

[129] Ivan Fratric. Domato - DOM fuzzer. https://github.com/googleprojectzero/domato, 2017.

[130] James Forshaw. Breaking the Chain. https://googleprojectzero.blogspot.com/2016/11/breaking-chain.html, 2016.

[131] Artur Janc, Charlie Reis, and Anne van Kesteren. Coop and coep explained. https://docs.google.com/document/d/1zDlfvfTJ_9e8Jdc8ehuV4zMEu9ySMCiTGMS9y0GU92k, 2020.

[132] Jeff Aboud. Why You Need to Stop Using CVSS for Vulnerability Prioritization. https://www.tenable.com/blog/why-you-need-to-stop-using-cvss-for-vulnerability-prioritization, 2020.

[133] Jeremy Fetiveau. Circumventing Chrome's hardening of typer bugs. https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/.

[134] Yonghwi Jin, Jungwon Lim, Insu Yun, and Taesoo Kim. Compromising the macOS kernel through Safari by chaining six vulnerabilities. In *Black Hat USA 20*.

[135] Zihao Jin, Ziqiao Kong, Shuo Chen, and Haixin Duan. Timing-based browsing privacy vulnerabilities via site isolation. In *SP22*.

[136] Joe Belfiore and Windows Experience Blog. Microsoft Edge: Making the web better through more open source collaboration, 2018.

[137] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *SP19*.

[138] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security 16*.

[139] Ivan Krstić. Behind the scenes of ios and mac security. In *Black Hat USA 16*.

[140] Ivan Krstić. App sandbox and the mac app store. In *WWDC 2011*, 2011. https://developer.apple.com/videos/play/wwdc2011/204/.

[141] The Citizen Lab. The Million Dollar Dissident: NSO Group's iPhone Zero-Days used against a UAE Human Rights Defender. https://citizenlab.ca/2016/08/million-dollar-dissident-iphone-zero-day-nso-group-uae/, 2016.

[142] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *SP14*.

[143] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. In *USENIX Security 20*.

[144] Wilson Lian, Hovav Shacham, and Stefan Savage. A call to arms: Understanding the costs and benefits of jit spraying mitigations. In *NDSS*, 2017.

[145] Hongyang Lin, Junhu Zhu, Jianshan Peng, and Dixia Zhu. Deity: Finding deep rooted bugs in javascript engines. In *ICCT19*.

[146] Linux. Seccomp BPF (SECure COMPuting with filters). 2012.

[147] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX Security 18*.

[148] LLVM Project. Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html, 2019.

[149] ARM LTD. ARMv8 architecture reference manual, for ARMv8-A architecture profile (ARM DDI 0487C.a). 2017.

[150] Mads Ager and Erik Corry and Vyacheslav Egorov and Kentaro Hara and Gustav Wibling and Ian Zerny. Oilpan: Tracing garbage collection for blink. 2013.

[151] Adrian Marinescu. Windows vista heap management enhancements: Security, reliability and performance. In *Black Hat USA 06*.

[152] Mark Brand and Sergei Glazunov and Project Zero. Analysis of CVE-2020-16010: Chrome for Android ConvertToJavaBitmap Heap Buffer Overflow. https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2020/CVE-2020-16010.html, 2021.

[153] Mathias Bynens. Elements kinds in V8, 2017.

[154] Mathias Bynens. JavaScript engine fundamentals: Shapes and Inline Caches, 2018.

[155] Matt Molinyawe, Abdul-Aziz Hariri, Jasiel Spelman. $hell on Earth: From Browser to System Compromise. In *Black Hat USA 16*.

[156] McAfee Labs. Don't Substitute CVSS for Risk: Scoring System Inflates Importance of CVE-2017-3735. https://www.mcafee.com/blogs/other-blogs/mcafee-labs/dont-substitute-cvss-for-risk-scoring-system-inflates-importance-of-cve-2017-3735/, 2017.

[157] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[158] Microsoft. JIT Spraying Never Dies - Bypass CFG By Leveraging WARP Shader JIT Spraying. https://sites.google.com/site/bingsunsec/WARPJIT.

[159] Microsoft. Control Flow Guard, 2015.

[160] Microsoft. Introducing windows defender application guard for microsoft edge. https://blogs.windows.com/msedgedev/2016/09/27/application-guard-microsoft-edge/, 2016.

[161] Microsoft. Microsoft defender application guard overview. https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-application-guard/md-app-guard-overview, 2021.

[162] Microsoft. Microsoft edge support for microsoft defender application guard. https://docs.microsoft.com/en-us/deployedge/microsoft-edge-security-windows-defender-application-guard, 2021.

[163] Microsoft Defender Security Research Team. Exploit kits remain a cybercrime staple against outdated software – 2016 threat landscape review series. https://www.microsoft.com/security/blog/2017/01/23/exploit-kits-remain-a-cybercrime-staple-against-outdated-software-2016-threat-landscape-review-series/, 2017.

[164] Matt Miller. Mitigating arbitrary native code execution in Microsoft Edge, 2017. https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/.

[165] Matthew R. Miller, Kenneth D. Johnson, and Timothy William Burrell. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities.

[166] Max Moroz and Sergei Glazunov. Analysis of UXSS exploits and mitigations in Chromium. Technical report, 2019.

[167] Mozilla. Static Analysis for Rooting and Heap Write Hazards. https://firefox-source-docs.mozilla.org/js/HazardAnalysis/index.html.

[168] Mozilla. Spy in the sandbox - security issue related to high resolution time api. https://bugzilla.mozilla.org/show_bug.cgi?id=1167489, 2015.

[169] Mozilla. Plugin Roadmap for Firefox. 2016.

[170] Mozilla. Changes affecting Adobe Flash on Firefox for Mac. https://support.mozilla.org/en-US/kb/changes-affecting-adobe-flash-firefox-mac, 2018.

[171] Mozilla. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, 2018.

[172] Mozilla. Firefox 79 for developers. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/79#javascript, 2020.

[173] Mozilla. Security Bug Approval Process. https://firefox-source-docs.mozilla.org/bug-mgmt/processes/security-approval.html, 2020.

[174] Mozilla. Introducing site isolation in firefox. https://blog.mozilla.org/security/2021/05/18/introducing-site-isolation-in-firefox/, 2021.

[175] Mozilla. Project fission - mozillawiki. https://wiki.mozilla.org/Project_Fission, 2021.

[176] Mozilla. Security Bug Bounty Program. https://www.mozilla.org/en-US/security/bug-bounty/, 2021.

[177] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. Analyzing Control Flow Integrity with LLVM-CFI. In *ACSAC19*.

[178] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *CCS18*.

[179] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *CCS15*.

[180] Chris Paget. Exploiting design flaws in the Win32 API for privilege escalation. *White Paper*, 2002.

[181] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website fingerprinting at internet scale. In *NDSS16*.

[182] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *SP20*.

[183] phoenhex team. CVE-2018-4233 Exploit. https://github.com/phoenhex/files/blob/master/exploits/ios-11.3.1/, 2018.

[184] The Chromium Project. Miracleptr aka raw_ptr aka backuprefptr. https://chromium.googlesource.com/chromium/src/+/ddc017f9569973a731a574be4199d8400616f5a5/base/memory/raw_ptr.md, 2021.

[185] The Chromium Project. Miracleptr one pager. https://docs.google.com/document/d/1pnnOAIz_DMWDI4oIOFoMAqLnf_MZ2GsrJNb_dbQ3ZBg, 2021.

[186] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX Security 19*.

[187] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. Sok: In search of lost time: A review of javascript timers in browsers. In *EuroSP21*.

[188] Paul Sabanal and Mark Vincent Yason. Digging deep into the flash sandboxes.

[189] Saelo. Attacking JavaScript Engines: A case study of JavaScriptCore and CVE-2016-4622. http://www.phrack.org/issues/70/3.html, 2016.

[190] Saelo. Compile Your Own Type Confusions: Exploiting Logic Bugs in JavaScript JIT Engines. http://phrack.org/issues/70/9.html, 2019.

[191] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-level fuzzing. In *USENIX Security 21*.

[192] Samuel Groß. New Trends in Browser Exploitation: Attacking Client-Side JIT Compilers. In *Black Hat USA 18*.

[193] Samuel Groß and Project Zero. JSC Exploits. https://googleprojectzero.blogspot.com/2019/08/jsc-exploits.html, 2019.

[194] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS18*.

[195] SecureList. Chrome 0-day exploit CVE-2019-13720 used in Operation WizardOpium. https://securelist.com/chrome-0-day-exploit-cve-2019-13720-used-in-operation-wizardopium/94866/, 2019.

[196] SecureList. PuzzleMaker attacks with Chrome zero-day exploit chain. https://securelist.com/puzzlemaker-chrome-zero-day-exploit-chain/102771/, 2021.

[197] Anatoly Shusterman, Daniel Genkin, Ayush Agarwal, Yossi Oren, Sioli O'Connell, and Yuval Yarom. Prime + Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses.
[198] siguza. APRR | Apple hardware secrets. https://siguza.github.io/APRR/, 2019.
[199] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. *CCS18*.
[200] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*.
[201] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. Browser history re: visited. In *WOOT18*.
[202] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *CCS17*.
[203] Brave Software. Brave Browser: Secure, Fast & Private Web Browser with AdBlocker. https://brave.com, 2021.
[204] WebKit Source. Structureid randomization. https://github.com/WebKit/WebKit/blob/main/Source/JavaScriptCore/runtime/StructureIDTable.h, 2021.
[205] Spring, Jonathan and Hatleback, Eric and Householder, Allen D. and Manion, Art and Shick, Deana. Towards Improving CVSS. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=538368, 2018.
[206] Statcounter Global Stats. Browser Market Share Worldwide, 2021.
[207] Sven Morgenroth. Goodbye XSS Auditor. https://www.netsparker.com/blog/web-security/goodbye-xss-auditor/, 2019.
[208] Google Threat Analysis Group (TAG). How we protect users from 0-day attacks. https://blog.google/threat-analysis-group/how-we-protect-users-0-day-attacks/, 2021.
[209] Microsoft Edge Team. Mitigating speculative execution side-channel attacks in microsoft edge and internet explorer. https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/, 2018.
[210] The Chromium Team. The Chromium Project. http://www.chromium.org/Home, 2021.
[211] Ars Technica. Firefox 0-day in the wild is being used to attack Tor users. https://arstechnica.com/information-technology/2016/11/firefox-0day-used-against-tor-users-almost-identical-to-one-fbi-used-in-2013/, 2016.
[212] The Chromium Team. Safer Usage Of C++, 2021.
[213] Inc. The Tor Project. Tor Project | Anonymity Online. https://www.torproject.org, 2021.
[214] Trishita Tiwari and Ari Trachtenberg. Alternative (ab) uses for HTTP Alternative Services. In *WOOT19*.
[215] Tom Ritter. Bug Bounty Program Updates: Adding (another) New Class of Bounties. https://blog.mozilla.org/attack-and-defense/2020/08/18/exploit-mitigation-bounty/, 2020.
[216] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *CCS15*.
[217] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections. In *USENIX Security 20*.
[218] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security 17*.
[219] W3C. Standards - W3C. https://www.w3.org/standards/, 2021.
[220] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *ICSE19*.
[221] Yong Wang. Thinking outside the JIT Compiler: Understanding and bypassing StructureID Randomization with generic and old-school methods. In *Black Hat USA 19*.
[222] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. Secure dynamic code generation against spraying. In *CCS10*.
[223] WeLiveSecurity. Brave browser's Tor mode exposed users' dark web activity. https://www.welivesecurity.com/2021/02/22/brave-browser-tor-mode-exposed-dark-web-activity/, 2021.
[224] WHATWG. Web Hypertext Application Technology Working Group (WHATWG). https://whatwg.org/, 2021.
[225] Rui Wu, Ping Chen, Bing Mao, and Li Xie. Rim: A method to defend from jit spraying attack. In *ARES12*.
[226] X41. X41 Browser Security White Paper. https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf, 2017.
[227] Wen Xu, Soyeon Park, and Taesoo Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *CCS20*.
[228] Mark Vincent Yason. Understanding the Attack Surface and Attack Resilience of Project Spartan's (Edge) New EdgeHTML Rendering Engine.
[229] Zhang Yunhai. Bypass control flow guard comprehensively. In *Black Hat USA 15*.
[230] Google Project Zero. A very deep dive into iOS Exploit chains found in the wild. https://googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html, 2019.
[231] Google Project Zero. In-the-Wild Series: Chrome Exploits. https://googleprojectzero.blogspot.com/2021/01/in-wild-series-chrome-exploits.html, 2021.
[232] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *SP13*.

APPENDIX

A. Table of Bug-finding Tools

We summarize the papers about browser fuzzers in the past decade in Table VI.

B. Privacy-preserving Browsers

One of the most concerning privacy leakages in web browsing is the user's IP address. As web servers can easily collect and store the IP, the user's geolocation can be instantly exposed with fine granularity depending on network circumstances (e.g., NAT). The Tor browser [213] addresses this problem with the *onion protocol*, which re-routes the user's connection using multiple random nodes in the Tor network, and each node cannot know the user's identity (IP) and the destination at the same time. However, privacy can still be breached via *website fingerprinting* techniques by observing the encrypted network packet sequences [181], [199]. Another browser, Brave [203], prevents websites from tracking users by removing all ads and ad trackers contained in websites, but the user's browsing history can still be leaked [214], [223].

C. Plugins and Extensions

Plugins and extensions are small software programs that customize the the browser's functionality by offering a wide variety of features. Plugins such as Java and Flash operate within the context of web pages, whereas extensions attach additional features to browsers. Despite their benefits, plugins are major sources of browser instability [8], [9]. Plugins also make sandboxing the renderer process impractical, as plugins are written by third-parties and browser vendors have no control over their access to the operating system. Also, extensions have special privileges within the browser, making them appealing targets for attackers [10]–[12].

NPAPI plugins. NPAPI allows browser vendors to develop plugins with a common interface. When the browser visits a page with an unknown content type, it searches for and loads the available plugin to delegate the content processing. As a result, attackers can trigger a vulnerability by assigning a specific content type to a web page that fools the browser into loading a specific plugin that has a vulnerability. Attacks on NPAPI plugins had been prevalent over different browsers and platforms, especially on Java, Flash, and PDF [163]. To mitigate the problem, browsers separated the plugin process from the browser's main process, namely out-of-process plugin

G: Generational, **M:** Mutational, **SM:** Semantic Aware, **SN:** Syntactic Aware, **Cov:** Coverage Feedback, **OS:** Open Source, **C:** Chrome, **FF:** Firefox, **S:** Safari, **E†:** Edge based on V8, **E‡:** Edge based on ChakraCore

Fuzzer	Year	C+E†	FF	S	E‡	G	M	SM	SN	Cov	OS	Key Techniques
SoFi [124]	2021	1	5	1	18	✗	✓	✓	✓	✓	✗	Uses fine-grained program analysis and repair strategy to generate semantically valid inputs
Token-Level Fuzzing [191]	2021	16	3	4	6	✗	✓	✗	✗	✓	✗	Applies mutation at *token-level* by changing or replacing entire words
Favocado [91]	2021	8	NA	5	NA	✗	✓	✓	✓	✗	✓	Generates test cases based on semantic information, and tracking states mutation
DIE [182]	2020	4	NA	16	28	✓	✓	✓	✓	✓	✓	Preserves beneficial properties and conditions called *aspects* across mutation
FREEDOM [227]	2020	4	5	13	NA	✓	✓	✓	✓	✓	✓	Uses customized IR (FD-IR) to describe HTML documents and to define mutation
Montage [143]	2020	1	0	2	34	✗	✓	✓	✓	✗	✓	Transforms JS ASTs into sequences to train Neural Network Language Models (NNLMs)
Nautilus [70]	2019	NA	NA	NA	2	✗	✓	✗	✓	✓	✓	Combines grammar-based input generation with coverage feedback
Deity [145]	2019	NA	NA	1	1	✓	✓	✗	✓	✓	✓	Generates syntactic JS code using previously known bugs and PoCs
Superion [220]	2019	NA	NA	16	3	✗	✓	✗	✓	✓	✓	Employs grammar-aware test input trimming with tree and dictionary-based mutation
CodeAlchemist [122]	2019	2	NA	10	7	✓	✗	✓	✓	✗	✓	Tags code bricks with constraints defining when to combine with other code bricks
LangFuzz [125]	2012	11	20	NA	NA	✓	✓	✓	✓	✗	✗	Generates grammar-aware test inputs, and leverages previously known faulty programs
Total		42	23	54	81							

TABLE VI: Comparison of browser engine fuzzers.

mitigation [170], [188]. However, plugins could still be used for browser exploitation and were accused of being the reason for performance degradation browser crashes. As a result, all browsers discontinued support for NPAPI plugins [169].

D. Difficulty of Deploying Mitigations

It is difficult for browser vendors to deploy mitigations for the following reasons:

a) Compatibility. Third-party code such as browser plugins depend on the browser code to function correctly. When introducing browser mitigations, it is possible to break third-party code, which browser vendors have no control over. For example, when trying to introduce Win32k lockdown for the Pepper Plugin API (PPAPI) for Chrome in Windows, there was a stability issue when applying the patch on Windows 8.1 and below, which the Chrome team could not track down [130], affecting plugins such as Flash, PDFium, and Widevine. As a result, PPAPI Win32k lockdown was only enabled for Windows 10 and not Windows 8/8.1 to avoid stability issues.

b) Performance. Adding security mitigations is expensive. To mitigate security threats, browser vendors sometimes choose to trade performance for security or vice versa. For example, the disabling of `SharedArrayBuffer` (SAB) in all modern browsers in early 2018 as a countermeasure for the Spectre attack, as discussed in §IV-D, greatly jeopardizes performance because SAB was originally designed to achieve lightweight synchronization between workers [76].

c) Security. More code usually means more security vulnerabilities. Often, introducing mitigations or patches increases the attack surfaces. After deploying new patches to browsers, browser vendors often look for bug reports to address the new security issues as soon as possible. For instance, Firefox launched a whole new class of bug bounties only for security vulnerabilities in active mitigations [215].

Reverted mitigations. Some mitigations are deployed temporarily to mitigate immediate threats while better mitigations are being developed. For example, in the SAB case mentioned above, shortly after the introduction of more robust countermeasures, *i.e.*, Site Isolation and COOP/COEP, Chrome and Firefox re-enabled the use of SAB [27], [172]. Despite all the efforts to ensure that the mitigations are safe, performant, and compatible, sometimes mitigations have to be rolled back due to some severe consequences they introduce. For example, in Table IV, XSS Auditor [72], an inbuilt XSS filter for Chrome, suffered from many security side-effects, which led to its retirement in 2019 [207].