The "Web/Local" Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing

Yaoqi Jia
National Univ. of Singapore
jiayaoqi@comp.nus.edu.sg

Zheng Leong Chua
National Univ. of Singapore
chuazl@comp.nus.edu.sg

Hong Hu
National Univ. of Singapore
huhong@comp.nus.edu.sg

Shuo Chen
Microsoft Research
shuochen@microsoft.com

Prateek Saxena
National Univ. of Singapore
prateeks@comp.nus.edu.sg

Zhenkai Liang
National Univ. of Singapore
liangzk@comp.nus.edu.sg

ABSTRACT

Process-based isolation, suggested by several research prototypes, is a cornerstone of modern browser security architectures. Google Chrome is the first commercial browser that adopts this architecture. Unlike several research prototypes, Chrome's process-based design does not isolate different web origins, but primarily promises to protect "the local system" from "the web". However, as billions of users now use web-based cloud services (e.g., Dropbox and Google Drive), which are integrated into the local system, the premise that browsers can effectively isolate the web from the local system has become questionable. In this paper, we argue that, if the process-based isolation disregards the same-origin policy as one of its goals, then its promise of maintaining the "web/local system (local)" separation is doubtful. Specifically, we show that existing memory vulnerabilities in Chrome's renderer can be used as a stepping-stone to drop executables/scripts in the local file system, install unwanted applications and misuse system sensors. These attacks are purely data-oriented and do not alter any control flow or import foreign code. Thus, such attacks bypass binary-level protection mechanisms, including ASLR and in-memory partitioning. Finally, we discuss various full defenses and present a possible way to mitigate the attacks presented.

1. INTRODUCTION

Web browsers were originally monolithic pieces of software without a principled way for separating their objects and resources [42]. As a result, any successful binary-level exploit (e.g., through a memory bug) would take over the entire browser. For many years, this was an ordeal for browser security. To confine malicious web sites, modern browser architectures adopt sandbox techniques to strongly enforce the separation between the web domain and the local domain. The sandbox prevents malicious web sites from affecting local systems even when there are exploitable memory errors in browsers. In addition, recent browser architectures have adopted a process-isolated design. The goal is to further confine the damage of a security exploit within the boundary of the compromised process, so that objects and resources in other processes are unaffected. After initial research prototypes [52], Google Chrome is the first commercial browser to adopt this approach [43]. Internet Explorer (IE) has also adopted it since version 8.0. As of July 2015, Google Chrome and IE (on desktop and mobile) are estimated to have over 80% of the worldwide web browser usage [37]. In this paper, we revisit the security offered by this de-facto architecture, using Google Chrome as a representative.

The Google Chrome's architecture uses privilege separation in two ways. First, Chrome isolates the browser kernel process, which has access to the user's local OS interfaces, from the renderer process, which is sandboxed and hosts the web page's resources. Second, Chrome isolates website instances [28] or tabs into separate renderer processes. Chrome's design highlights a clear deviation from the conceptual proposals in Gazelle [72] and OP2 [53] — a tradeoff between security and performance [50]. A particular difference is about enforcing the *same-origin policy (SOP)* [30]. The same-origin policy prevents any accesses from `a.com` to `b.com` in the browser, unless they communicate through certain explicit cross-domain channels (e.g., PostMessage [36]). In both of the aforementioned research prototypes, enforcing the same-origin policy is an explicit goal of the process isolation. They aim to place objects from different web domains in different processes, so that cross-domain accesses must take place as inter-process communication (IPC) calls [21]. However, when adopting this approach, Chrome makes a compromise between the sandboxing granularity and the affordable performance overhead: the goal of the sandboxing is only to restrict the accesses between contents from the web and those in the local file system, whereas SOP is not enforced by the sandboxing if the contents from different domains are loaded into one process. It is the rendering engine that is responsible for enforcing SOP within a process.

However, there is a common misconception that Chrome's sandboxing granularity is per domain. Actually, Google is upfront about this in an early paper about Chrome's security architecture, in which "origin isolation" (essentially the SOP) is listed as one of the "out-of-scope goals" [43]. Rather, the explicit goal of its process-based sandboxing is to separate "the web" from the "local system"[1].

The notion of web has evolved to include cloud services over the past a few years, which are increasingly integrated with local systems and devices. For example, storage services like Dropbox, OneDrive, and Google Drive are pervasive. These kinds of ser-

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978414

[1]Though there are on-going efforts to improve site-based isolation in the Chromium project, such as the roadmap outlined in [32] and the out-of-process iframes [26], they are not practically available to offer stronger protection.

vices integrate the cloud with the local file system, i.e., a local folder is created to automatically synchronize with the storage in the cloud. Similarly, source code repository services like GitHub also represent new usage scenario of cloud services, in which the user regularly pulls files into the local file system. Likewise, application (app) stores, cloud VM management consoles and remote host-management apps are common scenarios in which cloud services install programs and manage privileges on the local system. Therefore, the reality today is that the web (i.e., the cloud) is an integral extension of the local system, not a foreign landscape that can only be viewed through a browser. Considering this reality, we re-examine the trade-offs made in browser design.

Threat to web/local isolation. In this paper, we present a study to question the effectiveness of such a coarse-grained "web/local" separation in practice. We argue that, if the process-based sandboxing regards the same-origin policy as an "out-of-scope goal", then its promise of maintaining the "web/local" boundary is in doubt.

By compromising the renderer process of a website, an attacker can access the contents of the site on behalf of the user. However, to universally access the contents of arbitrary origins including sites of cloud services, the attacker has to bypass the SOP enforcement. Once SOP is compromised, the attacker can further bypass web/local isolation for any files related to any cloud services the victim user has already authenticated to. Essentially, with a coarse-grained "web/local" isolation enforced by the process-based sandboxing, the burden of the SOP enforcement entirely relies on the renderer logic. We believe this is a dangerous choice, and to substantiate the claim, we revisit Chrome's design and actual implementation for the SOP enforcement. Specifically, we find several weaknesses in Chrome's SOP design against a memory exploit in the renderer process. We have explored several types of local-machine services that can be subverted if SOP is compromised. For instance, we show attacks which in several cases (a) create data files with unrestricted file permissions, (b) remotely control virtual machines, (c) install apps on local system, (d) read device sensors such as geolocation and so on. Many of these are hard to achieve via the direct renderer-kernel interface. For example, files downloaded through Chrome are by default only readable, while the files imported via the Dropbox local folder interface can be executable. After bypassing the SOP enforcement in Chrome, a memory exploit can leverage the web interfaces (websites) of cloud services to indirectly access the local system.

Bypassing SOP enforcement in Google Chrome. In the presence of intra-process protections including internal address space layout randomization (ASLR) [2] and partitioning [27] in Chrome, as well as data execution prevention (DEP) [41] and control-flow integrity (CFI) [39] defenses, it is difficult to exploit memory bugs to change control flows and further bypass SOP in Chrome. In contrast, we employ data-oriented attacks to corrupt SOP-related critical data to bypass the SOP enforcement. In Google Chrome, we observe that the renderer process is responsible for performing SOP checks. For instance, X-Frame-Options dictates whether a site `b.com` can be iframed by `a.com` [2]. Such a specification is at the discretion of `b.com`, and its enforcement should ideally be done in the browser kernel (or outside `a.com`'s renderer process). We find several instances of such policy enforcement logic that are instead done in the renderer. The consequence of sharing a renderer process between distrusted origins is that all SOP checks must be done by a security monitor within the same process. Protecting the security-sensitive state of the SOP-enforcing monitor is a difficult task in the presence of low-level memory corruption bugs. We show that memory corruption vulnerabilities can be used to corrupt the security-critical data in the renderer process to bypass SOP. These attacks can target purely non-control data, and do not require any foreign code injection or control flow hijacking. That is, they remain agnostic to deployment of DEP and CFI defenses.

Chrome employs several best practices to minimize exploits via memory errors. Chrome separates its heap memory into different partitions, and deploys ASLR on these partitions with its internal random address generator independent of that in the underlying OS. We show that although such defenses are clearly useful, they are insufficient to block SOP bypasses. This is because resource objects of one origin can be manipulated to create pointers to objects of other origins. Since they share the same address space, such references can be used to cross origin-based partitions and bypass memory randomization of partitions.

Concrete attacks and light-weight mitigation. We have verified our attack by exploiting the renderer in Chrome 33 using a memory vulnerability [8]. Further, we have taken a best-effort approach to identify SOP-related critical data, and successfully found over 10 SOP checks and their corresponding critical data. We have disclosed the details of our attacks to Google Chrome team, and they have acknowledged our findings. They also admit that a massive refactoring of Chrome is required to isolate origins into different processes, which prevents our attacks. We discuss the defenses both on cloud services, and on web browsers. As the first step to protecting the same-origin policy in Chrome against data-oriented attacks, we have implemented a light-weight prototype based on address and data space randomization of security-critical data for SOP. Our evaluation shows that our mitigation introduces negligible overhead.

Contributions: In conclusion, we make following contributions:

- **Threat to Web/Local Isolation.** We highlight the importance of SOP enforcement in maintaining the "web/local" separation in light of emerging web-based cloud services. Once SOP is bypassed, cloud services can be abused by a renderer attacker via the web interface to access the local system.
- **Security Analysis of Chrome.** We take a closer look at Chrome's design and implementation for the SOP enforcement. We find that there are several important differences between Chrome's deployment and the ideal design, e.g., misplaced security checks and unprotected security-critical states.
- **Concrete Data-oriented Attacks for Chrome.** We show how to construct reliable data-oriented attacks that target the security-critical states in the renderer address space to bypass the SOP enforcement. Our attack demos are available online [33].
- **Light-Weight Mitigation against Data-oriented Attacks.** As a first step, we propose the use of address randomization and data pointer integrity as a light-weight mitigation to protect security-critical data of SOP checks.

2. BACKGROUND & THREAT MODEL

In this section, we give a brief overview of Chrome's security mechanisms, namely, web/local isolation and SOP enforcement in Chrome, as well as our threat model.

2.1 Process-Based Sandboxing in Chrome

Chrome's multi-process architecture. For a complex system like a web browser, bugs are almost inevitable. A good approach to enhance reliability and security is to isolate bugs in different processes. Specifically, in Chrome, there are two types of processes:

[2] `a.com` loading `b.com` in an iframe.

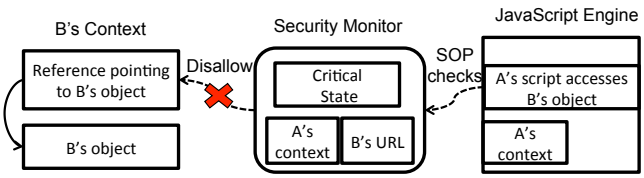

Figure 1: Overview of the SOP enforcement in Chrome's renderer.

there is one process to act as the browser kernel module that handles the interaction with the local system, while the other constitute the renderer module representing the web [24]. They are referred to as the browser kernel process and the renderer processes, respectively.

The browser kernel process is in charge of issuing network requests, accessing persistent storage such as cookies and history, and displaying bitmaps on the user's screen. A renderer process is responsible for parsing web documents including HTML and CSS, interpreting JavaScript, and rendering the web content into bitmaps [43]. The browser kernel process is trusted by the operating system, whereas the renderer is assumed to be untrusted because it includes untrusted resources from the web. To enforce the web/local isolation, each renderer process is restricted within a sandbox (described below), As such, it can only access limited local resources such as network and files from the browser kernel process through IPC calls, but cannot directly read/write the user's file system.

Sandbox mechanisms for web/local isolation. Chrome's sandbox [31] leverages the security assurance provided by the operating system. The specific assurances depend on the operating system. For example in Linux, Chrome has two layers of sandbox, i.e., Setuid and user namespaces for layer-1 sandbox as well as Seccomp-BPF [62] for layer-2 sandbox [23]. The layer-1 sandbox creates different network and PID namespaces for renderer processes. The layer-2 sandbox is designed to protect the browser kernel from malicious code executing in the user space. Consequently, renderers can only run with a limited set of privileges and access resources from the browser kernel process through IPC calls.

2.2 SOP Enforcement in Chrome

An origin is defined as the combination of protocol (or scheme), host and port. In Chrome, enforcement of the same-origin policy is the responsibility of the renderer process. This prevents scripts from accessing content between sites unless they share the same origin.

Figure 1 demonstrates the design of the SOP enforcement in Chrome's renderer. Consider the two scripts, hosted at `a.com` (or A) and `b.com` (or B) respectively. When A's script accesses objects in B's origin, a security monitor will check if the access is permitted by the rules of SOP. For instance, if A's script references the `contentDocument` field of B's window object using JavaScript, the security monitor will check whether obtaining such a reference is allowed under SOP. Once the reference is obtained by A's script, all objects transitively reachable via that reference are implicitly allowed, as JavaScript follows an object capability model for security checks [44, 60]. Therefore, the key to protecting B's object resources from cross-origin attacks is to interpose on all references to root objects such as `contentWindow` which contain references to B's other objects. Chrome inlines calls to the security monitor in the renderer to perform these checks comprehensively.

Intra-process protections. In-memory partitioning is used to separate logically unrelated pieces of data into different compartments. For instance, Chrome deploys its in-memory partitioning mechanism (called *PartitionAlloc* [27]) to separate the heap memory in each renderer into different partitions. With its own internal random address generator [2], Chrome randomizes the starting location of each partition and places unreadable/unwritable guard pages between partitions. These protections make memory bugs in the renderer difficult to exploit. However, we will show that our attack can successfully bypass all these protections in Section 4. Further, our attack can corrupt the critical state of the security monitor, and bypass the SOP enforcement in Chrome.

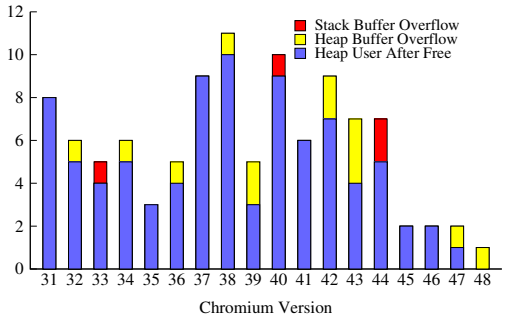

Figure 2: The memory vulnerabilities in the renderer module.

2.3 Threat Model

We assume a memory vulnerability in the renderer which can be exploited when the browser visits a website owned by the web attacker. The web attacker can entice the victim into visiting her website via spam emails, advertisements, search engine optimization (SEO) and other techniques. We also make a realistic assumption that some popular web-based cloud services (e.g., Dropbox, Google Drive and GitHub) are integrated into the user's local system. These services claim that over a billion users use them [14, 18]. We assume that the user has active web sessions (e.g., cookies have not expired) of these sites or the vulnerable browser remembers the credentials for them. The goal of our attack is to demonstrate that the intra-process protections are insufficient to prevent a memory vulnerability exploit from bypassing the SOP enforcement in the renderer. This enables the exploit to subvert the promise of the process-based sandboxing, which attempts to protect the local system from malicious sites.

Our attack requires a memory vulnerability that 1) it is located in the renderer process; 2) it can be used to read/write data within a certain non-negligible range. We analyzed 599 security vulnerabilities for Chrome listed in Chrome Issue Tracker [5] and CVE [17], from version 31 (Feb 2014) to 48 (Feb 2016). Based on the labels and descriptions for these vulnerabilities, we identified 104 memory vulnerabilities in the renderer module. They belong to three categories: use-after-free, heap buffer overflow and stack buffer overflow as summarized in Figure 2. The average number of vulnerabilities in the renderer module per major version is about 6.

We make no assumption on the configuration of the HTTP servers used by the web services. The servers can set any HTTP headers, e.g., X-Frame-Options and cross-origin resource sharing, on their websites. We will show in Section 5 how our attack bypasses SOP checks that enforce these policies.

3. WEB/LOCAL INTEGRATION

The development of cloud services has been blurring the boundary between web and local. For example, services like Dropbox and OneDrive synchronize a cloud drive with several local devices. Additionally, web services are also used to remotely control traditional desktop based OS (such as Amazon EC2) or mobile devices

	Channel	Example Apps	# of Users	Impact
1	Cloud Storage	Google Drive, Dropbox and OneDrive	400 M+	Leak private data and place malware on the user's local system
2	Code Hosting/Editing	GitHub, GitLab and Bitbucket	10 M+	Inject malicious code into the user's code repository
3	Email Clients	Thunderbird and Outlook	500 M+	Place malware into the user's email repository
4	Remote Control Service	TeamViewer, NoVNC and Device Manager	200 M+	Remotely control the user's device
5	App Store	Google Play	1000 M+	Install malware on the user's mobile system stealthily
6	Cloud Computing Service	OpenStack and Amazon EC2	1 M+	Control the user's VMs on the cloud
7	Video/Audio Recording	vLine, Talky and Skype	300 M+	Record the video/audio from the device's camera and microphone
8	Location Tracking	RunKeeper and Android Device Manager	10 M+	Track the user's location and places
9	Wearable Device	Google Fit	5 M+	Track the user's health information

Table 1: The web-based cloud services and their impact on local systems.

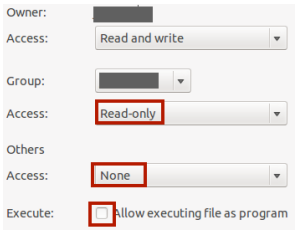

Figure 3: The binary file downloaded by Chrome is only readable.

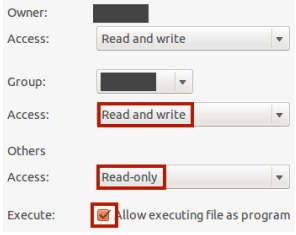

Figure 4: The same binary file synchronized by the Dropbox client is executable.

(such as Google Play). As a result, the web has become an integral extension of the local systems.

To understand the impact of such web-based cloud services, we have investigated over 1000 popular websites, and categorized the cloud service related sites into 9 channels as shown in Table 1. By exploiting the first three channels, the attacker can indirectly read-/write files on the user's local system. The latter three channels can potentially enable the attacker to interact with the local system to install applications or execute code. Using the last three channels, the attacker can leverage the system sensors, e.g., GPS sensor, heart rate monitor, camera and microphone, to collect the user and system's sensitive information. Next, we analyze these channels and the potential damages caused by abusing such channels.

3.1 Accessing Files on the Local System

Cloud Storage Service. Cloud storage services, like Google Drive, Dropbox and OneDrive, are widely adopted by users. For instance, Google Drive has over 240 million users [18], and Dropbox serves more than 400 million users [14]. The most common usage is to integrate such a service into the local file system so that it appears as a local folder. Users usually save documents, multimedia files and even executables in the folder. The files are automatically synchronized with the storage server. In order to improve usability, nearly all storage services provide a web interface for users to manage their files.

Consider Google Drive[3] as an example. If attackers can run JavaScript in Google's origin, they can load the site in an iframe, use JavaScript to operate the DOM elements and trigger actions on the site. On behalf of a user, the attacker can steal any private file by sharing it with herself, create a new file or copy a malware into the folder by accepting the attacker's shared file (as shown in our demo [9]). The attacker can also delete a file permanently, directly update the file or modify it using the shareable link. The Google Drive client will then synchronize all the files after these operations on the Google Drive server. These synchronized files typically keep the original permission, which is hard to achieve via the direct renderer-kernel interface. For example, the normally downloaded files in Chrome by default are only readable as shown in Figure 3; whereas, the files imported via the Dropbox client can have executable permission as shown in Figure 4.

Code Hosting. Web-based code hosting sites, like GitHub, GitLab and Bitbucket, offer distributed version control and source code management. Millions of developers use these services to host their various code repositories and projects [15]. These services support command-line tools for users to operate their repositories, and also allow users to commit updates on their websites. Once the attacker hijacks any of these sites in the user's browser, they can embed bugs or backdoors in the user's private project repositories by injecting malicious code into the files.

3.2 Interacting with the Local System

Remote Control Service. Remote control services allow users to remotely operate their computers. A popular example is Virtual Network Computing (VNC). VNC uses a simple frame buffer-based protocol with input events called the Remote Frame Buffer (RFB) protocol to communicate between the server and clients. Clients such as NoVNC and TeamViewer allow users to remotely control their computers using a web client, providing the convenience of accessing their remote device using nothing but a browser. Once authenticated, the user has an active session to the remote computer, which allows for any arbitrary action through interaction with the server using RFB. For example, NoVNC implements the RFB protocol as a JavaScript library that the web client interacts with. Hence, an attacker is able to control the remote host by accessing the RFB object of the web client from another origin using the memory vulnerability as described in Section 4.

Cloud computing services, like OpenStack and Amazon EC2, often provide control panels for users to remotely perform operations on virtual machines. OpenStack even supports remote shells (VNC) in the website for users to execute commands. As a result, an attacker can utilize a compromised renderer to run arbitrary commands/software on the user's remote VMs as shown in our demo [12].

Meanwhile, several mobile apps provide the service on their website to perform actions on the user's mobile device. For example, Android Device Manager's site allows the user to locate, ring, lock and even erase her phone. Thus, the compromised renderer can utilize these sites to remotely control the user's device.

App Store. To be convenient for mobile phone users, app stores (e.g., Google Play) usually provide installation synchronization across multiple devices. In Google Play, after selecting the targeted app, the user can choose one of her Android devices to install the app. Once the "Install" button is clicked, the Google Play's app on the user's chosen device will install the app without the user's explicit consent, e.g., popping up the permission prompt. This feature exposes to the renderer attacker a channel to stealthily install apps

[3]The site of Google Drive is https://drive.google.com.

(e.g., the attacker's malware) on the user's mobile device as shown in our demo [11]. If it is an Android version of the vulnerable Chrome, the compromised renderer can also craft malicious URLs and leverage Android URI intents [3, 54] to launch the installed apps to access the local system[4].

3.3 Misusing System Sensors

Apart from directly affecting the local system, bypassing SOP can be utilized to misuse system sensors. Web-based cloud services can collect the user or system's information (e.g., geolocation) from system sensors. Websites like vLine, Talky and Skype, use new HTML5 features to record media with the user's consent within the browser. Chrome remembers the user's decision for the prompt, especially the "allow" option. Thus the attacker's site can embed these sites into iframes/tabs, run JavaScript (e.g., getUserMedia) to record video/audio on behalf of the site with the media permission. This enables an indirect route to access system sensors. In addition, numerous mobile applications and wearable devices, e.g., Apple/Android watch and wristbands, can assist the user to keep track of her geolocation and health status. These apps record and synchronize the user's information from the user's device to their servers. By accessing the sites for these cloud services, the renderer attacker can obtain the user's sensitive information, e.g., geolocation and heart rate.

3.4 Threats to Web/Local Isolation

The web/local isolation enforced by process-based sandboxing in Chrome makes memory exploits in the renderer significantly more difficult to access local resources. However, through leveraging the web-based cloud service, a memory corruption exploit in the renderer can indirectly access the local system, which makes "web/local" isolation questionable.

The logic and critical data for the SOP enforcement in Chrome are placed in the renderer, thus a renderer exploit can affect the execution of SOP (details in Section 4). Once an attacker bypasses the SOP enforcement with a memory vulnerability in the renderer, the attacker's site like `a.com` can run scripts to load a cloud service's site such as `b.com` into an iframe or tab and perform actions (e.g., trigger buttons) to create/edit/delete files in the site. Although the renderer of `a.com` is sandboxed and cannot directly access the local system, the cloud service's client can synchronize the changes made by `a.com` on the local system and import the malicious files/contents added by `a.com`.

4. BYPASSING SOP ENFORCEMENT

To leverage the various cloud services to access the local system, a renderer attacker first needs to use a memory exploit to bypass the SOP enforcement. In Google Chrome, the consequence of sharing a renderer process across multiple origins is that many SOP checks must be done by a *security monitor* within the same process. However, this is difficult in the presence of memory corruption bugs. In this section, we show how SOP enforcement can be bypassed in Chrome using a memory bug in the renderer. In-memory defenses — including internal ASLR and partitions — can also be bypassed and simple extensions of these defenses are insufficient. We present concrete attack techniques that work on Chrome in our demos [33]. The details of a complete attack on bypassing intra-process protections can be found in Appendix A.

[4]In the newer Android version, Chrome does not launch an external app for a given intent URI, when the intent URI is directed from a typed-in URL or initiated without user gesture [3, 29].

4.1 A Data-Oriented Approach

Chrome employs security monitors in the renderer to enforce SOP as discussed in Section 2.2. The design of the SOP enforcement in the renderer suggests that the security monitor's internal state is critical. By using a memory corruption vulnerability to corrupt such critical monitor states, the JavaScript engine can be confused to allow A's script in accessing B's objects. Note that such corruption involves only data values, and does not require any injection of foreign code or even code reuse attacks. The outcome, though, is that A can run arbitrary computation (e.g., a JavaScript function) in B's security authority, as the JavaScript engine is a Turing-complete language interpreter. Such attacks are agnostic to the deployment of control-flow defenses such as CFI. The security monitor's critical data has to be readable/writable, and requires no execute permissions — therefore, DEP is not an effective defense against such data-oriented attacks. The remaining challenges in constructing such attacks is to identify the security critical monitor state and bypass any additional in-memory defenses that may hinder corruption.

In Chrome's implementation, we find that the security monitor logic is implemented as a set of function calls[5] in the renderer module. These functions use a large set of in-memory flags and data-fields to memorize the results of SOP security checks. These security-sensitive data fields dictate the implementation of SOP rules for various kinds of resources (e.g., JavaScript objects, cookies, and so on). Checks on the data fields can be either invoked from the context of A or B. We have confirmed over 10 different SOP-related checks that could be targeted for corruption, listed in Table 2 in Section 5, which allow bypassing SOP in different ways. Further, there are certain decision-making flags which allow "universal" access, i.e., if such flags are set, no cross-origin checks will be performed. For instance, we find the `m_universalAccess` flag which if set, unconditionally returns `true` in the SOP access control check (see Listing 1).

```
bool SecurityOrigin::canAccess(const SecurityOrigin*
     other) const
{
    if (m_universalAccess)
        return true;
    if (this == other)
        return true;
    if (isUnique() || other->isUnique())
        return false;
    ......
    return canAccess;
}
```

Listing 1: `SecurityOrigin::canAccess` in Chrome's renderer process.

In summary, the in-memory security monitor performing SOP checks for arbitrary scripts is susceptible to data-oriented attacks [47, 55, 56]. These attacks do not store/run shellcode on the data area (heap or stack) and require no control-flow hijacking. Thus, even fine-grained implementations of CFI [39, 78, 79, 48] and DEP are insufficient defenses. Next, we discuss additional defenses that Chrome employs — memory separation/partitioning and ASLR — and discuss why such defenses do not suffice as well.

[5]These functions include `isOriginAccessibleFromDOMWindow`, `canAccessFrame`, `SecurityOrigin::canAccess`, and so on.

4.2 Bypassing ASLR

With a memory corruption bug, attackers can usually achieve the privilege to read/write data within an address range. Chrome creates logic compartments (e.g., partitions) in the memory and utilize address space layout randomization (ASLR) to randomize the base addresses of partitions as discussed in Section 2.1. This makes the location of the vulnerable buffer and the SOP-related critical data unpredictable. Note that Chrome employs an internal ASLR within the renderer process, which randomizes the address even if ASLR in the underlying OS is disabled. Here, we describe an approach to bypass ASLR in a fairly generic way, and this is effective even when fine-grained data randomization is applied to randomize the offsets between objects within one partition.

The basic idea is that the attacker's script crafts an object with a predictable layout pattern — this is called a "fingerprinting" object. The location of the fingerprinting object is randomized by ASLR. However, the attacker's script can use a memory error to linearly scan the memory searching for the location of the object. When the object is identified, its location reveals the randomized address. If the fingerprinting object contains pointers to other objects, like a targeted object of the attacker's choice, these pointers can in turn be used to de-randomize the location of the aforementioned object.

For concreteness, we take the memory bug CVE-2014-1705 [8] as an example. This bug allows an attacker to modify the length of a JavaScript array and thus enables memory access past the array bound. Let us observe how this bug can be used to de-randomize the runtime address of the vulnerable buffer with fingerprinting technique. The attacker's script, running in the randomized process, first creates a fingerprinting object — for instance, an object that points to the string constant "aaa...a" k times. This can be done easily by creating k DOM nodes, like `<div>` elements, with a string attribute referencing the "aaaa...aa" string. Next, with the buffer overrun, the code linearly scans the memory after the buffer to find a four-byte value that repeats the most number of times (that of the k string pointers) as shown in Figure 5. The repeated value is the address of the string "aaaa...aa", which can be read programmatically by the script. Thus, we can use this memory fingerprinting to reliably recover the address of objects.

```
class PLATFORM_EXPORT SecurityOrigin : public
      RefCounted<SecurityOrigin>
{   ......
    String m_protocol;
    String m_host;
    String m_domain;
    String m_suboriginName;
    unsigned short m_port;
    bool m_isUnique;
    bool m_universalAccess;
    bool m_domainWasSetInDOM;
    bool m_canLoadLocalResources;
    bool m_blockLocalAccessFromLocalOrigin;
    bool m_needsDatabaseIdentifierQuirkForFiles;
};
```

Listing 2: The definition of the class `SecurityOrigin`.

The fingerprinting technique can be used to fingerprint objects with unique, predictable layout patterns. For instance, several security-critical data are stored in an object of the `SecurityOrigin` class in Chrome, which has a fixed, unique layout pattern as shown in Listing 2. We use the same fingerprinting technique in a linear scan to identify the security-critical object in our attacks.

This fingerprinting technique is different from memory disclosure attacks commonly used to bypass ASLR, where the address is leaked to an external network. Similar to attacks such as JIT-ROP [67], our attack uses the fact that the adversary is running a powerful script which allows the reuse of in-place, de-randomized addresses as an input to the next-stage of the exploit. We wish to point out that this fingerprinting is fairly robust even against finer-grained data-space randomization, which may randomize offsets between objects in memory. This is because the technique relies on the fingerprint of the object layout, and doesn't make any strong assumptions about fixed relative offsets between objects in memory. However, extremely fine-grained randomization, which randomizes the internal layout of an object may defeat such a technique. Finally, we point out that the technique has been used in data-structure identification for kernels [49], and in some control-flow hijacking attacks in the wild [76, 16]; our goal is to utilize them to construct data-oriented exploits.

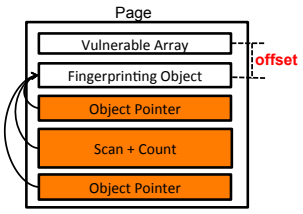

Figure 5: Obtaining the randomized address of an object with fingerprinting.

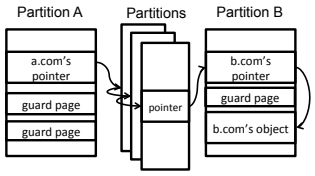

Figure 6: Overview of origin-based partitioning implemented in Chrome.

4.3 Bypassing Partitioning

Browsers (e.g., Chrome and IE) also employ in-memory partition to separate the heap memory into different partitions. The concept of an in-memory partition is fairly generic, and can be used to separate logically unrelated pieces of data in separate compartments. For instance, Chrome uses partitions to separate different types of objects in four different partitions. One can also use partitions to separate the object heaps of two distinct origins, or separate objects within an origin across multiple partitions. Typically, two protections are used to separate partitions: (a) each partition is surrounded by inaccessible guard pages (cannot be read/written), and (b) each partition's location is randomized. Guard pages provide protection against sequential memory corruption (e.g., buffer overruns), since overflowing reads or writes will access guard pages and cause an exception. Partition randomization protects against spatial memory errors (such as buffer overflow) and temporal memory errors (such as use-after-free). Our fingerprinting technique in Section 4.2 is restricted to work only within one partition, since guard pages are likely to disallow reads crossing outside a partition.

Nevertheless, partitioning itself is insufficient in preventing actual memory corruption attacks. The key to achieving such attacks is *cross-partition references* or pointers that link objects in one partition to another. In an environment like a web browser, object references are pervasive and often under the control of scripts. Therefore, the attacker's scripts can dereference pointer values to perform memory reads or writes across partition boundaries — a sort of bridge to cross partitions. For instance, Chrome isolates all buffer objects allocated in a `buffer` partition from other objects, which are not linearly accessed in a `general` partition. However, the buffer partition contains references pointing to data in the `general` partition. In our attack, we fingerprint a cross-partition reference by creating a specific number k of these references pointing to a crafted object. These references, once obtained by the fingerprinting method, can be dereferenced to reach the `general` partition that stores security-critical objects.

Insufficiency of Origin-based Partitioning. It might be tempting to suggest a different partitioning strategy, say one origin per partition. We will explain why such a strategy is insufficient. To support primary communication channels among different origins, e.g., PostMessage, browsers need to provide a handler of the targeted origin B for the requesting origin A. Thus the origin A should have an object containing a reference created by the browser and pointing to an object within the origin B. As Figure 6 depicts, `a.com`'s objects are in Partition A and `b.com`'s objects are isolated in Partition B. An attacker can exploit a memory corruption bug in Partition A to find the address of the object in Partition B by reading its pointer located in A. In this way, the attacker can further read/write the data in Partition B. Therefore, the origin-based partitioning does not prevent a memory exploit from crossing partitions and fingerprinting an object reliably.

5. ATTACK IMPLEMENTATIONS

We have investigated and implemented our attack on Chromium version 33. In this section, we will detail the end-to-end implementations to bypass the SOP enforcement in Chrome.

We take Google Drive as an example to show a concrete attack. When a user visits the attacker's website in Chrome, the script running in the site exploits an unpatched memory vulnerability in the renderer process. With the fingerprinting techniques, the script bypasses Chrome's ASLR and partitioning, finds the critical data for the security monitor (e.g., the `SecurityOrigin` object) and corrupts the critical flag such as `m_universalAccess`. Thus, the script bypasses the corresponding SOP checks like `SecurityOrigin::canAccess`. Then the script obtains the cross-origin access capabilities to load Google Drive's site into an iframe ignoring the X-Frame-Options, and further uses `contentDocument` to access the content in Google Drive. With these capabilities, the script can trigger clicks to share private documents to the public, create/edit or permanently delete files, and perform other privileged actions on behalf of the user. After these actions, the Google Drive's server assists the client-side software installed on the user's local system to synchronize these operations, which may introduce malicious files to the local system or integrate malicious code into the current local files. With that, we have summarized the capabilities an attacker can have into three categories[6]:

- Arbitrary cross-origin reads (1 - 4)
- Arbitrary cross-origin reads/writes (5 - 8)
- Getting/setting cross-origin persistent storage (9 - 11)

We have identified over 10 SOP checks and the corresponding critical data (as shown in Table 2). We have verified our attack by exploiting the renderer in Chrome 33 using a memory vulnerability found in the JavaScript engine V8 [8]. We provide the methodology of identifying these critical data in Section 5.4.

5.1 Arbitrary Cross-origin Reads

In this section, we describe four different attacks using arbitrary cross-origin read. The attacks are, reading arbitrary local files using FILE scheme, bypassing frame busting defenses to load cross-origin sites into iframes, using XMLHttpRequest to read the contents of cross-origin sites and reading the data of tainted canvas.

FILE Scheme. Since file paths are treated as part of origins, SOP implies that the local file should not be able to access the content of other files, e.g., `file:///Download/a.html` cannot load another file like `file:///etc/passwd`. Further, the browser kernel ensures that a renderer process created for HTTP scheme is unable to load resources under FILE scheme no matter what the origin is. However, this check is not enforced for FILE scheme. That is, scripts from FILE scheme are able to load resources from HTTP scheme. Given this design decision, we will show how an attacker is able to exfiltrate any local file to an attacker-controlled server given a memory vulnerability in the renderer process [10].

The first step of the attack is to entice the user into opening a downloaded malicious HTML file. As a result, the rendered HTML file will have FILE scheme instead of HTTP scheme. The memory vulnerability can be used to set the `m_universalAccess` flag as described previously in order to bypass the security monitor deployed in the renderer. This allows us to bypass the SOP check to read the content of any local file. Finally, the content can be exfiltrated by simply constructing a GET/POST request to an attacker controlled remote server and appending the content as a parameter to the request.

X-Frame-Options. This header prevents one site from being iframed in a cross-origin site. If Chrome does not load the content of a cross-origin site into the renderer, the attacker cannot read its content even with arbitrary memory reads. By modifying `m_protocol`, `m_host`, and `m_port` in the `SecurityOrigin` object to that of the targeted site, the attacker can bypass the SOP check[7] gaining the ability to load any sites without X-Frame-Options or with X-Frame-Options set as "SAMEORIGIN" into iframes. Currently, the majority of sites using X-Frame-Options set it as "SAMEORIGIN" [58]. For the sites with "DENY" setting, we can utilize `window.open` to load them instead.

Cross-origin Resource Sharing (CORS). This mechanism relaxes the SOP restriction by allowing one origin's resources to be requested from another origin. The resource provider can set `Access-Control-Allow-Origin` as the requesting site's domain or "*" (any domain) in the response header to explicitly instruct Chrome to enable the requesting domain to access the resources via XMLHttpRequest. Two of the functions are invoked by the CORS check[8]. Setting `m_universalAccess` makes these two functions return true. This causes the CORS check to always pass and therefore, the attacker's site can fetch the resources (e.g., HTML and JavaScript) from arbitrary sites via XMLHttpRequest.

Tainted Canvas. The `getImageData` [4] interface is used to obtain an `ImageData` object representing the underlying pixel data for the specified rectangle on a canvas. Once a canvas is mixed (or tainted) with a cross-origin resource (e.g., image), Chrome does not allow scripts to obtain the data from the tainted canvas. The access to the tainted canvas results in a call to `SecurityOrigin::taintCanvas`. Similar to that of the CORS check, the access check invokes both `canAccess` and `canRequest`. By corrupting `m_universalAccess`, the attacker will be able to remotely read data in the canvas.

5.2 Arbitrary Cross-origin Reads/Writes

Beyond the capability to arbitrary cross-origin reads, we can also obtain the cross-origin arbitrary read/write capabilities. With these capabilities, we can further use JavaScript to manipulate operations on the DOM elements in the cross-origin sites in iframes or tabs/windows, e.g., trigger clicks to create/edit/delete private files in

[6]We demonstrate another category of bypassing discretionary controls in Appendix B.

[7]Implemented as a flag `isSameSchemeHostPort`.

[8]The two functions are `SecurityOrigin::canRequest` and `SecurityOrigin::canAccess`.

	Features/APIs	Descriptions	Example functions to perform checks (using `blink` or `content` namespace)	Bypass with overwriting critical data
1	FILE scheme	Blocking to read arbitrary local files in the user's system	canAccessFrame, SecurityOrigin::canRequest and SecurityOrigin::canAccess	Set `m_universalAccess` as true; Set `m_protocol`, `m_host`, `m_port` same with the targeted site in the `SecurityOrigin` object
2	X-Frame-Options	Preventing loading cross-origin websites into iframes	FrameLoader:: shouldInterruptLoadForXFrameOptions, parseXFrameOptionsHeader, SecurityOrigin::isSameSchemeHostPort	Set `m_protocol`, `m_host`, `m_port` same with the targeted site in the `SecurityOrigin` object
3	CORS for XMLHttpRequest	Blocking XMLHttpRequets to retrieve the data from cross-origin sites	XMLHttpRequest::createRequest, SecurityOrigin::canRequest and SecurityOrigin::canAccess	Set `m_universalAccess` as true; Set `m_protocol`, `m_host`, `m_port` same with the targeted site in the `SecurityOrigin` object
4	getImageData	Blocking to access the tainted canvas's data	SecurityOrigin::taintsCanvas, SecurityOrigin::canRequest and SecurityOrigin::canAccess	Set `m_universalAccess` as true; Set `m_protocol`, `m_host`, `m_port` same with the targeted site in the `SecurityOrigin` object
5	contentDocument/ contentWindow property	Blocking to access the content (e.g., DOM elements) of cross-origin websites in iframes	BindingSecurity::shouldAllowAccessTo-Frame, canAccessFrame, SecurityOrigin::canAccess, SecurityOrigin::isSameSchemeHostPort	Set `m_universalAccess` as true; Set `m_protocol`, `m_host`, `m_port` same with the targeted site in the `SecurityOrigin` object
6	window.frames property	Blocking to access the content of cross-origin sites within frames in the frameset	isOriginAccessibleFromDOMWindow, SecurityOrigin::canAccess, SecurityOrigin::isSameSchemeHostPort	Set `m_universalAccess` as true; Set `m_protocol`, `m_host`, `m_port` same with the targeted site in the `SecurityOrigin` object
7	window.parent /parent, window.top/top	Blocking cross-origin sites in iframes/frames to read/write the content of the parent window	isOriginAccessibleFromDOMWindow, SecurityOrigin::canAccess, SecurityOrigin::isSameSchemeHostPort	Set `m_universalAccess` as true; Set `m_protocol`, `m_host`, `m_port` same with the targeted site in in the `SecurityOrigin` object
8	window.open	Blocking to access a new opened tab/window of a cross-origin site	isOriginAccessibleFromDOMWindow, SecurityOrigin::canAccess, SecurityOrigin::isSameSchemeHostPort	Set `m_universalAccess` as true; Set `m_protocol`, `m_host`, `m_port` same with the targeted site in the `SecurityOrigin` object
9	Cookies	Blocking to get/set cookies of cross-origin sites	WebCookieJar::cookies/WebCookieJar:: setCookie and Document::cookie/Document:: setCookies	Set `m_cookieURL` as the targeted site's URL in the `Document` object
10	Localstorage	Blocking to get/set localstorage of cross-origin sites	DomStorageDispatcher:: OpenCachedArea, and WebStorageNamespaceImpl:: createStorageArea	Set `m_protocol`, `m_host`, `m_port` same with the targeted site in the `SecurityOrigin` object
11	IndexedDB	Blocking to get/set indexedDB of cross-origin sites	IndexedDBDispatcher:: RequestIDBFactoryOpen, and WebIDBFactoryImpl:: open	Set `m_protocol`, `m_host`, `m_port` same with the targeted site in the `SecurityOrigin` object

Table 2: The critical data for security-related policies in Chrome.

the cloud storage sites. Next, we present the instances to bypass the security monitor for different interfaces such as `contentDocument` and `window.open` to achieve these capabilities.

The `contentDocument/contentWindow` interface is used to access the DOM elements of an origin within an iframe. Similarly, the `window.frames` DOM API is used to read/write the content of frames in the frameset; and the `window.parent/top` API can be utilized by frames/iframes to access the content of the parent window. Based on SOP, Chrome disallows one site to access the content of another cross-origin site in a frame/iframe via these JavaScript APIs. The security monitor (e.g., `canAccessFrame`) in the renderer is used to determine whether one origin can access the origin in an iframe/frame or not. By corrupting the critical security monitor state — enabling `m_universalAccess` or setting `https` for `m_protocol`, `b.com` for `m_host` and 0 for m_port, we can obtain the capabilities for arbitrary reads/writes in `b.com`. This results in the ability to manipulate the DOM elements and mimic actions on behalf of the user in any sites within iframes/frames.

Since Chrome is designed as process-per-site-instance, a site opened by JavaScript (i.e., `window.open`) in a new tab/window still belongs to the opener's process. By default, the opener can only access the content of the page in the new tab if they have the same domain. However, if the renderer attacker modifies the decision-making data for the security monitor such as `isOriginAccessibleFromDOMWindow`, the checks can be bypassed to access the content of an arbitrary site in the opened tab/window. With this feature, the attacker can access the data of a cross-origin site with X-Frame-Options as "DENY" in the opened tab.

5.3 Cross-origin Persistent Storage

Websites usually use cookies [19], localstorage [35] and indexedDB [20] to store persistent data to maintain the states for web sessions. We find that these storage APIs strictly follow SOP, which indicates that only the origin itself can access its data stored in the storage. The browser kernel handles the implementation and updates of these storages for various sites. However, in Chrome's current process model, one renderer process may contain different origin sites in iframes, and the security monitor for the check of these storage is deployed in the renderer. Considering cookies an example, we find that by setting `m_cookieURL` as the targeted site's URL in the `Document` object used in security-check functions[9], the attacker can get/set cookies of arbitrary origins. Similarly, setting `m_protocol`, `m_host`, and `m_port` used in the functions[10] allows the attacker to access data in localstorage and indexedDB of the targeted site.

5.4 Identifying Critical Data

Chrome has over 5 million lines of code. It is quite challenging to find the security monitors and identify the critical data for the SOP enforcement. We have taken a best-effort approach to identify the decision-making data for the SOP checks as shown in Figure 7. Our methodology is described below.

1) Test generation. We set up an experimental website, say `https://a.com`. The site's page contains JavaScript code for various SOP-related functionalities (e.g., loading sites into iframes and

[9]Such as `WebCookieJar::cookies` and `WebCookieJar::setCookie`.

[10]Implemented as `WebStorageNamespaceImpl::createStorageArea`.

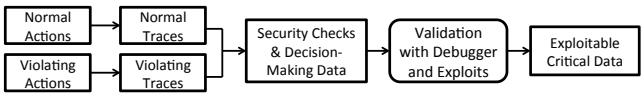

Figure 7: Overview of the critical data identification. We trigger the normal/violating actions in the experimental site, and record the invoked functions/return values as traces. By comparing the traces, we identify the decision-making data in the security-check functions. Finally, we modify the data to verify whether they are exploitable to bypass SOP or not.

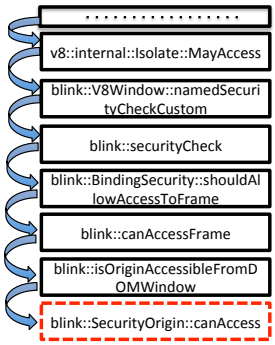

Figure 8: The invoked functions when accessing an iframe.

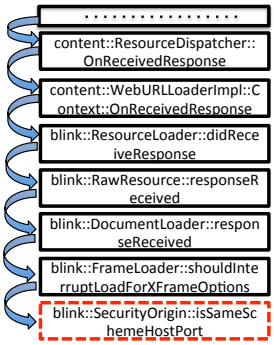

Figure 9: The invoked functions when loading a site into an iframe.

accessing their contents), which can be triggered to perform normal actions (following SOP) and SOP-violating actions. For example, we run the scripts using `contentDocument` to access to `a.com` in an iframe as a normal action or to access a cross-origin site `b.com` as a violating action. Typically, Chrome throws an error for this violating action.

2) Generating execution traces. We generate the execution traces for both actions. For instance, we attach the renderer process for the experimental page with a debugger like GDB and load symbols for the renderer module. For one functionality such as the `contentDocument` interface, we find a related function (e.g., canAccessFrame) in the source code[11] and set a breakpoint for this function. Next, we trigger the normal action for the functionality, the process will stop at the breakpoint. By backtracing and stepping into invoked functions, we record the invoked functions and variables/return values as the trace for the normal action. Analogously, we generate the trace for the SOP-violating action. We repeat this trace-generating step till we have adequate information to differentiate the trace of the normal action from the violating one's. Figure 8 and 9 demonstrate the invoked functions for accessing an iframe and loading a site into an iframe.

3) Identifying the deviations in data/function ways. The difference between the normal action and the violating one is that the latter one violates the specific SOP check, e.g., the check for `contentDocument`. Thus the corresponding traces also reflect this difference. By analyzing the traces for one SOP-related functionality, we find the functions that in different traces contain different return values or different values for the same variable. For example, the function `SecurityOrigin::canAccess` returns true in the normal trace, but it returns false in the violating trace. We label these security-check functions with red color in Figure 8 and 9. Within these functions, we can also identify the variables called decision-making data, which influence the return values. In addition to the variable having different values in two traces, we also find several decision-making data (e.g., `m_universalAccess`) that have the same value in different traces but can actually affect the function's return.

4) Validating the corruption attack concretely. After identifying the decision-making data, we first verify them with the debugger. We set breakpoints at the security-check functions, and trigger the violating action. After modifying the decision-making data (e.g., setting `m_universalAccess` as true), we resume the execution of the process. If we receive the same result as the normal action's, as well as Chrome does not throw an error message, we successfully bypass the security-check functions; otherwise, the decision-making data do not work. In addition, we validate the critical data that pass the first phase with real memory corruption exploits. By exploiting memory corruption vulnerabilities [8] (e.g., heap buffer overflow), we now have the privilege to arbitrary read/write data in the memory space in the renderer process. Using that, we can verify if the security checks can be bypassed by corrupting the respective critical data associated with it. In cases where the lifespan of the critical data is not within the influence of the memory corruption exploit, we categorize it as a non-exploitable critical data. The results passing the two-phase verification are summarized in Table 2.

6. MITIGATING WEB/LOCAL ATTACKS

As the web/local attacks discussed in the paper involve both client-side exploits and server-side integration services, we can mitigate the attacks from both sides.

Defense on the side of cloud services. As the web/local integration is a necessary component of web/local attacks, to mitigate these attacks, cloud service providers should restrict the privileges of their web interfaces:

- Distinguish the request of its website from the one of its native client, which can be achieved by setting headers with different random strings for two types of requests.
- Restrict the privileges for the web interface. For instance, the files uploaded from the website can be added into the local folder, but they are only readable without preserving the original permissions such as executable.
- Require the user's consent when accessing the local system. For example, before the web interface instructs the cloud service to install an app or run remote commands on the local system, its native client on the system should prompt the user and request for her consent. If she agrees on the operation, the web interface can continue accessing the local system; otherwise, the request for the operation is denied.

Defense on the side of web browsers. Another perspective of defenses is to thwart our data-oriented attack for bypassing SOP enforcement on the browser side. One obvious solution is to partition each origin into a separate process, including sub-resources, as proposed by previous research prototypes, e.g., OP [52], Gazelle [72], IBOS [69] and OP2 [53]. In fact, Google Chrome has experimented with such a prototype called "Site Isolation" [32] including "Out-of-Process iframes" [26]; however, the design does not attempt to partition sub-resources other than iframes and has not yet transitioned to the release versions. We have contacted Google's Chrome team and have verified that it is still a work in progress

[11] In Chrome, we can locate the majority of the renderer's implementation code in the directory third_party/WebKit/Source and content/renderer.

with a massive refactoring and challenging open problems. The hurdles include balancing the performance overhead with isolation and retrofitting the new design on the legacy codebase while maintaining compatibility with the web. The prohibitive performance overhead of isolating all origins is consistent with previous conjectures. [50]. If all these resources are isolated by different processes, around 1 GB memory overhead (one browser process requires nearly 20 MB memory [40]) will be incurred for loading the single page. It is clear that more work has to be done in this area for a complete solution. Hence, we discuss here simpler, incremental mitigations against specific attacks that we presented.

The mitigations are based on the following observation: in order for attackers to corrupt critical data, they have to (a) locate the data objects in memory and (b) change their values. In addition, (c) the values set by attackers must be used by the application. All three conditions are necessary for a successful attack. As such, we propose to combine address space randomization (ASLR) with data-flow integrity (DFI) as the basic mechanism for the mitigation.

6.1 Proposed Browser-Side Mitigation

Let the set of sensitive data which we aim to protect be D. In our attacks, the sensitive data identified is in Table 2. Our defense works by storing D in a sensitive memory segment R, the location of which is randomized at load time. The attacker is assumed to have access to memory; hence, we protect the base address of R by always keeping it in a dedicated register XMM_R and never leaking it to memory. Legitimate access to R are always accessed by indexing with the XMM_R as the base via register operations only, and all such accesses are confined to the range of R. Finally, we ensure that no legitimate pointers to R are located outside of R, that is, R is the transitive closure of pointers referencing to the sensitive data D. With these invariants, formalized below, we can ensure that the attacker cannot directly find the location of any data in R with significant probability, without using a legitimate XMM_R-based indexing. Note that this eliminates our fingerprinting attack in Section 3, as well as precludes our use of cross-partition references since such references are eliminated by construction. We have implemented this mitigation in Chromium 33, and report on the results in Section 6.3.

This mitigation significantly raises the bar for the attacker, since the attacker is forced to use XMM_R-based indexing to access sensitive instructions. As the attacker cannot violate CFI, this reduces the number of instruction where it can mount an attack from drastically. The indices used in such instructions could still be forged, allowing the adversary to swap an access $XMM_R[i]$ with $XMM_R[j]$ where $i \neq j$, from those handful of legitimate instructions. A final defense could be used to tighten the protection of indices used in XMM_R operations. Specifically, we use a form of DFI for this. DFI ensures that data writes to a location x is bound to a small set of statically identified instructions I_x, significantly narrowing down the points in the program execution that could be used to corrupt indices into R. Our enforcement for DFI is cryptographic, similar to CCFI [61] except that it protects indices rather than code pointers. Specifically, x is authenticated-encrypted with its address as the key. The instructions statically determined to legitimately modify x seals it after modification while the instructions which read it verify the integrity of the authenticated-encrypted x. This locks down the set of instructions that could be used to mount corruption of indices into R to a small set. Though not a comprehensive defense, we believe this mitigation significantly reduces the attack surface, and eliminates our specific attacks presented.

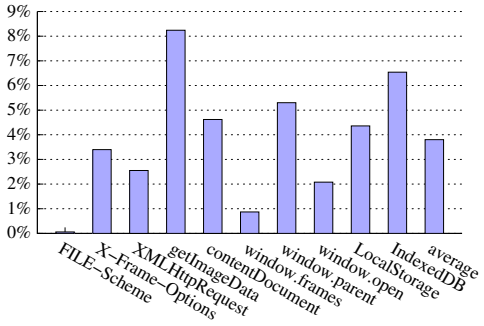

Figure 10: Performance overhead of the implemented mitigation.

6.2 Implementation

We have implemented a prototype of the proposed mitigation to enforce requirements described in Section 6.1. Although the current implementation only protects the identified data shown in Table 2, other critical data identified in the future can also be protected with the same method. The address of this secret memory region should be stored in `XMM` registers. However, this requires the recompilation of all libraries used by Chromium. To reduce the programming difficulty, our prototype does not reserve `XMM` registers for the whole program. Instead, when the critical object is visited, we save the `XMM` registers and load the secret address from browser kernel process with an inter-process call (IPC). Once the data is accessed, we restore the original value of `XMM` registers. In total we modified 485 lines of code, mainly focusing on file `SecurityOrigin.cpp` and `SecurityOrigin.h`.

With our prototype, attackers cannot directly corrupt the critical objects or the index to bypass SOP policy. However, attackers may try to change pointers of the index to achieve similar attacks [55]. Although possible in theory, we figure out that this attack is not practical in Chromium. Due to the complicated connection between different objects, attackers have to collect a large amount of accurate information to forge a "legitimate" critical object. Otherwise, the process will crash due to invalid pointer dereference.

6.3 Evaluation

To check the effectiveness of our implementation, we run our attack again on the modified Chromium. The attack fails to locate the security-critical objects from the system heap partition, as they have been moved to the dedicated secret memory region. This implies that the current implementation can prevent such data-oriented attacks.

We also measure the performance overhead introduced by the extra operation for critical objects access (e.g., register saving and restoring). We run several micro-benchmarks with the original Chromium and the modified one. Each benchmark performs one SOP-related operation in Table 2 100 times. Results of the micro-benchmarks are shown in Figure 10. As we can see, our protection with randomization introduces 3.8% overhead on average to each SOP-related operation. Considering the load time of the whole webpage, we believe such overhead will not introduce perceivable delay to user experience. We have also tested the modified Chromium with standard browser benchmarks [22, 25, 13], and do not notice any statistically significant difference from the original version. This is reasonable as most of the benchmarks do not trigger cross-origin accesses, thus the extra protection code is only triggered a few times during the critical object creation.

Our evaluation shows that the current implementation nullifies

our attacks with a negligible performance penalty. It is straightforward to extend current implementation to protect other critical objects (like the pointers of the known critical objects).

7. DISCUSSION & RELATED WORK

In this section, we discuss alternative full defenses and the challenges in adopting them in existing code bases.

Deploy XFI ubiquitously. XFI [51] combines software-based fault isolation (SFI) [71, 74, 66] with control-flow integrity (CFI)) [39, 38], and supports both fine-grained memory access control and memory isolation guarantees. We can deploy XFI to isolate each origin into a separate SFI-enforced fault domain (or partition), and to *completely* remove all direct cross-partition references. To do this, the renderer could allocate and isolate all objects belonging to origin A to its own fault domain. A's security monitor, running in its own separate fault domain, is responsible for setting up the segment base and offset for the origin A's partition. SFI-enabled code (e.g., the JavaScript interpreter) is guaranteed to execute within A's partition unless the segment base / offset is changed. The security monitor can ensure that only it has access to these SFI-critical registers. It is necessary to ensure the complete defense against code injection and reuse attacks, in addition, to ensure the guarantees. When A wishes to access objects either in the security monitor or the origin partition B, it must explicitly jump into a designated point into the security monitors (enforced by CFI checks). The security monitor can marshall data back-and-forth between these A's partition and B's.

This solution is plausible, but it raises several concerns. First, we still need to trust the security monitor to be free from vulnerabilities; otherwise, even SFI would allow corruption of security-critical state of the monitor. Second, there is a strong assumption about the attacker's ability to perpetrate control-oriented exploits in the presence of CFI. Though CFI is gaining deployment in commodity systems [70, 34], the security it offers is not perfect. Third, the performance impact of SFI-enforced solution in a browser environment could be high, because of the need for frequent context-switches to the security monitor partition. A careful implementation and empirical analysis of the performance of such a design is a promising future work. Finally, applying such a defense to an existing codebase is an independent challenge in its own right.

Fine-grained Data Layout Randomization. We have shown in Section 4.2 that simply using commodity ASLR and even randomizing the offsets between objects are not sufficient, since fingerprinting techniques can still work. Further, objects may hold references to other objects, which can be used to traverse objects in a non-linear fashion bypassing spatial memory error defenses. One way to bypass fingerprinting techniques is to add "honey" objects with the same layout structure as other objects. These objects are designed to mislead attackers and will crash the browser if it is modified. Another approach is to randomize the internal layouts of objects, so that it may be hard for the attacker to devise a deterministic fingerprinting strategy [46, 59]. This is again a plausible approach, but such fine-grained randomization may incur a runtime cost for field lookups with randomized mappings. Secondly, for most objects the amount of entropy in the layout structure is small, and may not provide a robust defense against multiple attack attempts. Finally, changing the layout structure of objects may have a high compatibility cost with legacy code of the browser, and external interfaces (e.g., plugins) it may have. We believe a solution that addresses performance, compatibility and a high entropy to data objects layouts is an interesting open problem.

Memory Safety. In addition to the defenses discussed above, we can enforce memory safety to thwart data-oriented attacks, i.e., enforcing data-flow integrity (DFI) [68, 73, 45]. By performing dynamic information flow tracking [73, 75] or analyzing the memory modification instructions [45, 77], we can figure out the illegitimate memory read/write operations and detect the memory corruption attacks before the SOP bypassing occurs. However, DFI can cause large performance overheads and requires lots of manual declassifications, which makes it presently impractical.

Alternatively, enforcing memory safety can mitigate the memory corruption attacks at the beginning. Numerous research approaches, e.g., Cyclone [57], CCured [65], SoftBound [63] and CETS [64], are proposed to provide memory safety for type-unsafe languages, e.g., C. To prevent data-oriented attacks we used in this paper, a complete memory safety is required for the browser's renderer. Nevertheless, to guarantee the memory safety has been a systemic challenge for over a decade. Though Google Chrome is continually advancing, e.g., address-sanitizer [1], the memory vulnerabilities for Chrome as shown in Figure 2 suggest that the complete memory safety may still be challenging.

8. CONCLUSION

Chrome's process-based design does not isolate different web origins, but only promises to protect "the web" from "the local system". In this paper, we demonstrate that existing memory vulnerabilities in the renderer can be exploited to bypass the SOP enforcement, and leverage the web-based cloud services to further access the local system, e.g., drop executables/scripts in the local file system. Our main results show concrete attack implementations, which explain the dangers of sharing process address space across origins in the browser, and its impact on the local system. As the first step, we employ address and data space randomization, and implement a prototype to protect SOP-related critical data in Chrome, which introduces negligible overhead and raises the bar against data-oriented attackers.

9. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their helpful feedback. We also thank Charlie Reis, Adrienne Porter Felt and Google Chrome team for useful discussions and feedback on an early version of the paper. This work is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

10. REFERENCES

[1] AddressSanitizer (ASan). https://www.chromium.org/developers/testing/addresssanitizer.

[2] AddressSpaceRandomization. https://chromium.googlesource.com/chromium/blink/+/master/Source/wtf/AddressSpaceRandomization.cpp.

[3] Android intents with chrome. https://developer.chrome.com/multidevice/android/intents.

[4] Canvasrenderingcontext2d.getimagedata(). https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/getImageData.

[5] Chromium Issue Tracker. https://code.google.com/p/chromium/issues/list.

[6] CSP (content security policy). https://developer.mozilla.org/en-US/docs/Web/Security/CSP.

[7] CSP policy directives. https://developer.mozilla.org/en-US/docs/Web/Security/CSP/CSP_policy_directives.

[8] CVE-2014-1705. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1705.

[9] Demo: Dropbox. https://youtu.be/P-oX0wEasz4.

[10] Demo: FILE Scheme. https://youtu.be/IPWJzzpvJdA.

[11] Demo: Google Play. https://youtu.be/nKyvCo5cn6c.

[12] Demo: VNC. https://youtu.be/dYSTxmNVgxI.

[13] DROMAEO, JavaScript Performance Testing. http://dromaeo.com.

[14] Dropbox Announces User Base Exceeds 400 Million, with Eight Million Business Users. http://www.cloudcomputing-news.net/news/2015/jun/26/dropbox-announces-user-base-exceeds-400-million-eight-million-business-users/.

[15] Github Press. https://github.com/about/press.

[16] Google Chrome Exploitation - A Case Study. http://researchcenter.paloaltonetworks.com/2014/12/google-chrome-exploitation-case-study/.

[17] Google Chrome Vulnerability Statistics. http://www.cvedetails.com/product/15031/Google-Chrome.html?vendor_id=1224.

[18] Google Drive Has Passed 240M Active Users. http://thenextweb.com/google/2014/10/01/google-announces-10-price-cut-compute-engine-instances-google-drive-passed-240m-active-users/.

[19] HTTP State Management Mechanism. http://tools.ietf.org/html/rfc6265.

[20] Indexed Database API. http://www.w3.org/TR/IndexedDB/.

[21] Inter-Process Communication. https://www.chromium.org/developers/design-documents/inter-process-communication.

[22] JetStream Benchmark,. http://browserbench.org/JetStream/.

[23] Linux and Chrome OS Sandboxing. https://code.google.com/p/chromium/wiki/LinuxSandboxing.

[24] Multi-Process Architecture. https://www.chromium.org/developers/design-documents/multi-process-architecture.

[25] Octane Benchmark. https://code.google.com/p/octane-benchmark.

[26] Out-of-Process Iframes. https://www.chromium.org/developers/design-documents/oop-iframes.

[27] Partitionalloc. https://chromium.googlesource.com/chromium/blink/+/master/Source/wtf/PartitionAlloc.h.

[28] Process Models. https://www.chromium.org/developers/design-documents/process-models.

[29] Redirecting to intent from manually entered url gives Unknown URL Scheme Error. https://bugs.chromium.org/p/chromium/issues/detail?id=477456.

[30] Same Origin Policy for JavaScript. https://developer.mozilla.org/En/SameoriginpolicyforJavaScript.

[31] Sandbox. https://www.chromium.org/developers/design-documents/sandbox.

[32] Site Isolation. https://www.chromium.org/developers/design-documents/site-isolation.

[33] SOP Bypass Demos. https://youtu.be/fIHaiQ4btok.

[34] Visual Studio 2015 Preview: Work-in-Progress Security Feature. https://blogs.msdn.microsoft.com/vcblog/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature/.

[35] Web Storage. http://www.w3.org/TR/webstorage/#the-localstorage-attribute.

[36] Window.postMessage(). https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.

[37] Market Share Reports. https://netmarketshare.com/, 2015.

[38] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. A Theory of Secure Control Flow. In *International Conference on Formal Engineering Methods*. 2005.

[39] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. In *ACM Conference on Computer and Communications Security*, 2005.

[40] D. Akhawe, P. Saxena, and D. Song. Privilege Separation in HTML5 Applications. In *USENIX Security Symposium*, 2012.

[41] S. Andersen and V. Abella. Memory Protection Technologies, Data Execution Prevention. Microsoft TechNet Library, September 2004.

[42] M. Andreessen. NCSA Mosaic Technical Summary. *National Center for Supercomputing Applications*, 1993.

[43] A. Barth, C. Jackson, and C. Reis. The Security Architecture of the Chromium Browser. http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf, 2008.

[44] A. Barth, J. Weinberger, and D. Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security Symposium*, 2009.

[45] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-Flow Integrity. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[46] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu. A Practical Approach for Adaptive Data Structure Layout Randomization. In *European Symposium on Research in Computer Security*, 2015.

[47] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, 2005.

[48] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *IEEE Security & Privacy*, 2014.

[49] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust Signatures for Kernel Data Structures. In *ACM Conference on Computer and Communications Security*, 2009.

[50] X. Dong, H. Hu, P. Saxena, and Z. Liang. A Quantitative Evaluation of Privilege Separation in Web Browser Designs. In *European Symposium on Research in Computer Security*. 2013.

[51] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[52] C. Grier, S. Tang, and S. T. King. Secure Web Browsing with the OP Web Browser. In *IEEE Security & Privacy*, 2008.

[53] C. Grier, S. Tang, and S. T. King. Designing and Implementing the OP and OP2 web Browsers. *ACM Transactions on the Web*, 2011.

[54] B. Hassanshahi, Y. Jia, R. H. Yap, P. Saxena, and Z. Liang. Web-to-Application Injection Attacks on Android: Characterization and Detection. In *European Symposium on Research in Computer Security*, 2015.

[55] H. Hu, Z. L. Chua, A. Sendroiu, P. Saxena, and Z. Liang. Automatic Generation of Data-Oriented Exploits. In *USENIX Security Symposium*, 2015.

[56] H. Hu, S. Shinde, A. Sendroiu, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *IEEE Security & Privacy*, 2016.

[57] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, 2002.

[58] S. Lekies, M. Heiderich, D. Appelt, T. Holz, and M. Johns. On the Fragility and Limitations of Current Browser-Provided Clickjacking Protection Schemes. *Workshop on Offensive Technologies*, 2012.

[59] Z. Lin, R. D. Riley, and D. Xu. Polymorphing Software by Randomizing Data Structure Layout. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2009.

[60] S. Maffeis, J. C. Mitchell, and A. Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *IEEE Security & Privacy*, 2010.

[61] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *ACM Conference on Computer and Communications Security*, 2015.

[62] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *USENIX*, 1993.

[63] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[64] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *ACM International Symposium on Memory Management*, 2010.

[65] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Principles of Programming Languages*, 2002.

[66] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium*, 2010.

[67] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Security & Privacy*, 2013.

[68] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Network & Distributed System Security Symposium*, 2016.

[69] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[70] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[71] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *ACM SIGOPS Operating Systems Review*, 1994.

[72] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security Symposium*, 2009.

[73] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *USENIX Security Symposium*, 2006.

[74] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Security & Privacy*, 2009.

[75] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving Application Security with Data Flow Assertions. In *Symposium on Operating Systems Principles*, 2009.

[76] Yu Yang. ROPs are for the 99%, CanSecWest 2014. https://cansecwest.com/slides/2014/ROPs_are_for_the_99_CanSecWest_2014.pdf, 2014.

[77] B. Zeng, G. Tan, and G. Morrisett. Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *ACM Conference on Computer and Communications Security*, 2011.

[78] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Security & Privacy*, 2013.

[79] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, 2013.

APPENDIX

A. BYPASSING INTRA-PROCESS DEFENSES

Chrome implements its own heap allocator called *PartitionAlloc* to separate the heap memory for each renderer process [27]. Within a renderer, PartitionAlloc separates allocations on the renderer's heap into four partitions and avoids allocating the metadata and user-input data in the same partition. Chrome deploys its built-in ASLR to assign randomized addresses for different partitions with its internal random address generator [2]. At present, the four partitions are as follows:

- Object Model Partition. This partition stores objects inherited from the Node class, which is from the DOM tree.
- Rendering Partition. This partition stores objects from the render tree.
- Buffer Partition. The objects from the Web Template Framework (WTF), e.g., ArrayBuffer, are allocated here.
- General (System Heap) Partition. The allocations from `WTF::fastMalloc` are stored in this partition[12].

Each partition is made up of a sequence of superpages of 2M bytes, which are divided into buckets based on the size the requested allocation. The superpage contains guard areas (or guard pages) that are reserved and inaccessible, which prevent the memory exploit from sequentially reading/writing (or overflowing) the memory across superpages.

To demonstrate the feasibility of bypassing intra-process defenses, we have verified our attack by exploiting the renderer in Chrome 33 using a memory vulnerability found in the JavaScript engine V8 [8]. This vulnerability allows an attacker to modify the length of a JavaScript array to any value. It enables the attacker to read-/write the data relative to the base pointer of the vulnerable array. To corrupt the security-critical data, we use four steps to exploit the vulnerability as shown in Figure 11.

Finding the address of the fingerprinting object (①). We first create a vulnerable array using JavaScript (e.g., ArrayBuffer) and

[12]We discuss the four partitions based on the PartitionAlloc's implementation in Chrome 33. In Chrome 45, Object Model Partition and Rendering Partition are replaced by Node Partition and Layout Partition respectively.

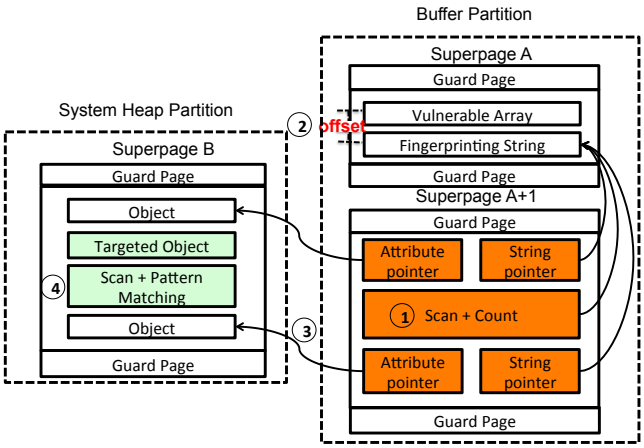

Figure 11: Overview of how to obtain the base address of the vulnerable array and find the fingerprinting object.

it will be allocated in one superpage in Buffer Partition. Thus we create a fingerprinting object (e.g., a string with value 'aa...a') after the vulnerable array. In order to obtain the address of the string, we create multiple (e.g., 0x200) DOM elements (e.g., DIV) with different attributes and each attribute points to the same string. Chrome allocates the elements in DOM Model Partition. The attribute objects, on the other hand, are created as elements of a WTF::Vector object and is allocated in the Buffer Partition too, as shown in Figure 11. Note that we want to force the creation of these attribute objects in a new superpage in the buffer partition. We achieve this by spraying the current superpage of the buffer partition with objects in order to fill it up. Next, we leverage the vulnerable array to perform a linear scan of the memory in the subsequent superpage to count the number of the occurrence for any 4-byte value (e.g., the size of a pointer). Statistically, we can infer that the non-zero value with the highest count is the address of the fingerprinting object (e.g., $object\_address$).

Finding the address of the vulnerable array (②). By conducting a stepwise search for the pattern of the fingerprinting object (i.e., a sequence of 'a') using the vulnerable array, we can obtain the offset between the array and the object. By subtracting the offset from the base address of the fingerprinting object, we obtain the vulnerable array's base address (i.e., $base\_address = object\_address - offset$).

Leveraging the cross-partition references (③). When loading a page into an iframe, a `SecurityOrigin` object is created together with a `Frame` object. This security object is allocated on the system heap using the system allocator, rather than the custom partitions. Due to the presence of guard pages, we need to identify the address range that is currently used as the system heap. Here, we will utilize the attribute object we created previously again. Each attribute object contains a pointer to a `QualifiedNameImpl` object and a pointer to an `AtomicString` object. Note that `QualifiedNameImpl` is allocated using the system allocator and hence the pointer will be a cross-partition reference to the system heap. By obtaining the largest `QualifiedNameImpl` address, we are able to obtain an estimate on the maximum legitimate address for the system heap.

Obtaining the address of the targeted object (④). Take the SOP enforcement for `contentDocument` as an example. To bypass this SOP enforcement, we need to overwrite `m_universalAccess` to bypass SOP checks. Since we have an estimate on the maximum heap address, we can search from a high address to a low address for the object's pattern in the memory and obtain the address of this targeted object (i.e., $target\_address$). Finally, we can calculate the index with the formula: $index = (target\_address - base\_address)/4$, and use $array[index]$ to overwrite the critical data (e.g., set `m_universalAccess` as true) to bypass SOP checks. After this modification, we can bypass the SOP check for `contentDocument` and access the content of arbitrary origins within iframes. With the same method, we can bypass X-Frame-Options and other SOP enforcements. Combining the steps ①, ②, ③ and ④ as shown in Figure 11, we can have an end-to-end exploit to bypass DEP, CFI, ASLR and partitioning, as well as corrupt the SOP-related critical data and achieve the four cross-origin access capabilities.

B. BYPASSING DISCRETIONARY CONTROLS

In addition to the capabilities to directly access cross-origin data, there are numerous discretionary controls in Chrome, e.g., CSP, and user prompts for geolocation. Bypassing them, the attacker can achieve the capability to run inline code in the targeted sites, and directly obtain the geolocation information. For concreteness, we demonstrate how to bypass these controls.

Content Security Policy (CSP) [6] provides directives [7] in HTTP response headers to instruct browsers to restrict the sources for different types of resources. For example, the "script-src" directive specifies valid sources for JavaScript. By setting proper CSP directives, the site can instruct Chrome to only load the permitted documents (e.g., scripts under the same domain) and execute them in the page. The strict CSP directives in a site could prevent the attacker's site from injecting scripts or directly manipulating DOM elements on the site in iframes. In Chrome, we find that the CSP-check functions are in the renderer. By enabling `m_allowSelf`, `m_allowStar`, `m_allowEval` and `m_allowInline` in the `CSPSourceList` object, we can bypass the corresponding checks. Therefore, we can use inline resources, e.g., inline scripts , JavaScript URLs, and inline event handlers, in the targeted site in an iframe or tab. In this way, the renderer attacker can bypass the CSP directives to run arbitrary code on the targeted sites in iframes.

HTML5 provides JavaScript APIs to enable sites to obtain the browser's geolocation, which requires the user's explicit approval. Chrome pops up with a permission prompt (controlled by the browser kernel) for the user to determine whether to allow the site to access the GPS sensor or not. We find that though the prompt is controlled by the browser kernel, the attacker's script can modify `m_geolocationPermission` as PermissionAllowed in the `Geolocation` object to bypass the check of `Geolocation::isAllowed` in the renderer. Then the attacker can obtain the user's geolocation using JavaScript.