박 사 학 위 논 문
Ph.D. Dissertation

새로운 웹 기술에 대한 보안 결함과 클라이언트 측 방어 기술 연구

Browser Evolution and Web Security: Security Defects and Client-side Countermeasures on Emerging Web Technologies

2021

이 지 연 (李 智 娟 Lee, Jiyeon)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

박 사 학 위 논 문

새로운 웹 기술에 대한 보안 결함과 클라이언트 측 방어 기술 연구

2021

이 지 연

한 국 과 학 기 술 원

전산학부

새로운 웹 기술에 대한 보안 결함과 클라이언트 측 방어 기술 연구

이 지 연

위 논문은 한국과학기술원 박사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2021년 6월 7일

심사위원장 신 인 식 (인)

심 사 위 원 강 민 석 (인)

심 사 위 원 손 수 엘 (인)

심 사 위 원 허 재 혁 (인)

심 사 위 원 김 효 수 (인)

Browser Evolution and Web Security: Security Defects and Client-side Countermeasures on Emerging Web Technologies

Jiyeon Lee

Advisor: Insik Shin

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Daejeon, Korea
June 7, 2021

Approved by

Insik Shin
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DCS 이지연. 새로운 웹 기술에 대한 보안 결함과 클라이언트 측 방어 기술 연구. 전산학부 . 2021년. 62+iv 쪽. 지도교수: 신인식. (영문 논문)
Jiyeon Lee. Browser Evolution and Web Security: Security Defects and Client-side Countermeasures on Emerging Web Technologies. School of Computing . 2021. 62+iv pages. Advisor: Insik Shin. (Text in English)

초 록

웹은 세계에서 가장 큰 플랫폼으로 설치 없이 어디서나 접근 가능하며 쉽게 검색 가능한 막강한 장점을 지닌다. 웹의 광범위한 사용을 지원하기 위해, 현대의 웹은 초기에는 디자인되지 않았던 새로운 기술을 끊임없이 흡수하며 확장되고 있다. 본 논문에서는 새로운 웹 기술이 사용자에게 끼치는 보안 영향을 깊이 있게 탐구하여 웹의 성장을 방해하지 않고 검색 환경을 보다 안전하게 만드는 것을 목표로 한다. 이를 위해 최근에 웹에 소개된 프로그레시브 웹 앱과 웹 가상현실이라는 두 혁신 기술에 초점을 맞춘다. 프로그레시브 웹 앱은 웹에서 네이티브 앱과 같은 기능을 가능하게 해주는 기술로 서비스 워커라는 백그라운드 실행 컴포넌트를 이용해서 웹에 푸시 알림과 오프라인 모드 등의 네이티브 앱 기술을 제공한다. 웹 가상현실은 웹에서 가상현실 기술을 가능하게 하는 기술로 웹에서 가상현실 실행을 위한 주변 장치(예: 디스플레이, 컨트롤러)를 관리하고 가상현실 컨텐츠를 렌더링하는 인터페이스를 제공하여 웹 상에서 몰입감있는 3차원 경험을 가능하게 한다. 하지만, 이와 같은 신흥 웹 기술들이 기존의 웹 보안 원칙과 대치되어 웹 보안을 약화시키는 상황이 존재한다. 예를 들어, 프로그레시브 웹의 백그라운드 실행은 사용자가 현재 이용 중인 웹사이트를 쉽게 인지할 수 있었던 웹의 기본 특성을 모호하게 만든다. 또한, 2차원 공간에서 다중 출처를 안전하게 실행하는 웹의 샌드박싱 기술이 3차원 공간에 단순 적용되지 않아 웹 가상현실 환경에서 출처 간의 분리된 렌더링 영역을 제공하기 어렵다. 이러한 이해를 바탕으로, 첫 번째 세부 연구에서는 실행중인 프로그레시브 웹 앱의 출처 판별의 어려움으로 비롯한 새로운 피싱 위협을 소개하고 시용자의 민감한 개인 정보(예: 웹사이트 방문 기록)가 유출될 수 있음을 보인다. 또한 사용자의 컴퓨팅 자원을 은밀하게 탈취하는 크립토재킹 공격을 제시한다. 두 번째 세부 연구에서는 악의적인 광고 제공자를 가정하여 웹 가상현실에 특화된 네 가지의 광고 사기 공격을 소개하고 3차원 공간에서 다중 출처의 리소스를 안전하게 렌더링 할 수 있는 솔루션을 제안한다. 이처럼 새로운 웹 기술에 대한 깊이 있는 조사와 실용적인 대책 마련을 통해 안전한 웹 브라우징 환경을 제공하는 데 도움을 줄 수 있음을 보인다.

핵 심 낱 말 웹 보안, 프로그레시브 웹 앱, 피싱, 웹사이트 방문 기록 유출, 크립토재킹 공격, 웹 가상현실, 온라인 광고 사기, 자바스크립트 격리 기술

Abstract

The Web is the largest platform in the world. It has the powerful advantages of being accessible everywhere and easily searchable without installation. To cope with such widespread adoption, the Web is rapidly evolving, absorbing new technologies that were not initially designed. This dissertation aims to explore the security implications of emerging Web technologies and make browsing environments more secure without hindering the evolution of the Web. To this end, we focus on two recent Web technologies, *Progressive Web App* and *WebVR*. A Progressive Web App (PWA) is a new generation of Web applications designed to provide native app-like browsing experiences. It provides native app features such as push notification and offline mode to the Web by using a background execution component called Service Worker. WebVR is a new technology that enables Virtual Reality (VR) on the Web, providing websites with interfaces to manage VR peripherals (e.g., HMDs, controllers) and render VR content thus enabling an immersive 3D experience on the Web. However, there are situations in which the emerging technologies are confronted with the conventional Web safety principles, thereby undermining Web security. For instance, the new execution pattern of PWAs (e.g., background execution) obscures the underlying property of the Web in which users could easily recognize the currently running site. Furthermore, the Web's sandboxing technique which securely executes multiple sources in 2D space is not simply applied to 3D space, making it difficult to provide isolated rendering execution contexts between multiple sources in the WebVR environment. Based on these observations, we first introduce new phishing threats exploiting the difficulties in identifying the source of running PWAs and demonstrate how users' sensitive information (e.g., browsing history) can be leaked. We also present a cryptojacking attack that covertly hijacks users' computing resources from visited PWAs. Second, we introduce four new advertising fraud attacks unique to WebVR by assuming a malicious advertising provider and propose a defense solution that can safely render multi-origin resources in a 3D world. In this way, we show that conducting in-depth investigations of the risks on emerging Web technologies and presenting practical countermeasures can help provide a secure Web browsing environment.

Keywords Web security, progressive web application, phishing, browsing history sniffing, cryptojacking, WebVR, online ad fraud, javascript sandboxing

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Motivation

The Web is the largest open application platform in the world. While its origins as a simple document delivery system, it has evolved into the most common method of delivering applications to users. The Web has the powerful advantages of being accessible everywhere with a single link, easy to search, and no installation required. Together with these features, the Web connects billions of devices, running a plethora of clients, and serves billions of users every day. As a result, it has become the key access point to plenty of security-sensitive services, which we use on a daily basis.

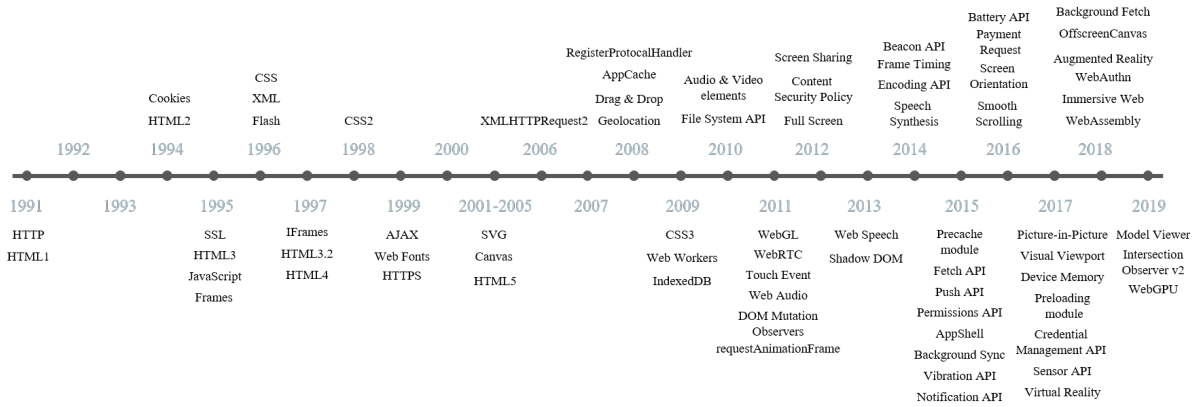


Figure 1.1: A timeline of the emergence of Web technologies

To cope with such widespread adoption, it is constantly evolving. As shown in Figure 1.1, the modern Web is rapidly developing, absorbing new technologies that were not originally designed. In particular, HTML5 proposed in 2004 has changed many aspects of the browser by introducing numerous new concepts, including new elements and JavaScript APIs to enhance storage, multimedia, and hardware access. With the advent of HTML5, the Web is able to provide rich content across a wide variety of computing environments beyond the desktop. Along with its increasing popularity, the Web has picked up more capabilities over time; it incorporates technologies from other areas such as Mobile and Virtual Reality (VR), offering immersive and native app-like browsing experiences over the traditional two-dimensional (2D) environments.

However, as the Web expands, there are situations in which emerging technologies encounter the unique nature of the Web, resulting in violation of security principles. For instance, many recent Web APIs that allow native app-level functionalities have combined with the advantage of high accessibility of the Web, removing the difficulty of installing malwares and facilitating attacks [28, 120, 140]. In addition, because the Web is no longer bounded to a single origin and provides multiple sources in complex forms, Same Origin Policies (SOPs) that protect resources between multiple origins is often bypassed or cannot accommodate new Web features [51, 58, 123, 160].

Such undesired situations in contrast to existing Web safety principles neutralize existing security mechanisms, consequently violating users' privacy and hindering the advancement of the Web. Therefore, there is a clear need for the security community to conduct in-depth investigations of the risks introduced

by emerging browser features.

1.2 Claim of the Dissertation

This dissertation aims to work on the intersection of browser evolution and Web security towards a better upcoming Web. Our goal is to explore the security implications of emerging Web technologies, how we can protect users, and make browsing environments more secure without hindering the evolution of the Web. Overall, our work serves as caveats about the potential risks of browsers implementing new technologies without an in-depth assessment of their security and privacy consequences.

To this end, we specifically focus on two innovative Web techniques called *Progressive Web App* and *WebVR*. Progressive Web App (PWA) is a new generation of Web applications; it provides a seamless native-app experience when visiting a website with PWA features. A PWA, in particular, uses cache [89], push notifications [55], and service workers [56] to provide offline Web browsing experiences as well as interactive customer services. The compatibility of these HTML5 features blurs the boundary between native and Web apps, supporting rich Web experiences with low latency.

WebVR is an open specification that enables websites to offer a VR environment by means of browser supports. Rendering a virtual world entails high demand for graphics processor unit (GPU) resources and the frequent loading of large-sized graphic textures and images. These requirements make it the only viable means for native applications that require installation and frequent software updates to deliver the VR world, resulting in hindered their widespread adoption. The advent of WebVR addresses these core limitations makes it easier for everyone to get into VR experiences, no matter what device users have. WebVR provides interfaces for managing VR peripherals such as HMDs and controllers and working in tandem with WebGL to render VR content on an HTML5 canvas element, thus enabling an immersive 3D world experience on the Web.

It is important to address two topics. This is because the advent of these technologies has resulted in significant improvements on the Web such as the abilities of *background execution* and *3D browsing*, rather than minor updates to bridge the gap between the Web and other domains. However, there are situations where those extended features contrast the conventional security related principles and techniques of the Web; (1) In 2D space, contents from multiple origins could be safely rendered on the same webpage via *iframe* which acts as an independent execution container. However, such iframe-like primitives are not simply extended to 3D space, making it difficult to provide separate rendering areas between multiple origins in a WebVR environment. (2) The Web based on foreground execution in the browser tab now has a new execution paradigm that allows the Web server to trigger execution at any time with the background execution persistence of PWAs. However, this new execution pattern obscures users' perceptions of the sites currently in use or where they come from compared to the traditional Web which is easily recognizable to the running sites.

The brief analysis of two technologies presented above suggests the following thesis statement:

Thesis statement. When new technologies such as WebVR and PWA are introduced on the Web, it needs to investigate whether the emerging technologies violate the conventional Web security principles (e.g., cross-origin sandboxing, perception of the current running site), which can help provide a secure Web browsing environment.

Chapter 1.3 summarizes our findings on considerable vulnerabilities and their remediation of both techniques.

1.3 Outline and Contributions

The main contributions of this thesis are situated in the two studies introduced above. The successive chapters consist of individual works published in top security conferences with the following contributions:
Abusing Native App-like Features in Web Applications (Chapter 3): We perform the first comprehensive review of the security and privacy aspects specific to PWAs as our first contribution. We identify security flaws in popular browsers and design flaws in popular third-party push services, that exacerbate the phishing risk. We introduce a new side-channel attack that infers the victim's history of visited PWAs. The proposed attack exploits the offline browsing feature of PWAs using a cache. We also present a new attack method that abuses service workers to perform a cryptocurrency mining attack. Defenses and recommendations to mitigate the identified security and privacy risks are suggested with thorough understanding.
WebVR Ad Fraud and Practical Confinement of Third-Party Ads (Chapter 4): As the second major contribution, we present four new ad fraud attack methods on the WebVR environment with assuming an abusive ad service provider. Our user study demonstrates that success rates of our attacks range from 88.23% to 100%, confirming their effectiveness. To mitigate the presented threats, we propose *AdCube*, which allows publishers to specify the behaviors of third-party ad library and enforce this specification. We show that AdCube is able to block the presented threats with a small page loading latency and a negligible frame-per-second (FPS) drop for nine WebVR official demo sites.

Before presenting each Chapter, Chapther 2 introduces previously well-known Web attacks and defenses related to this thesis. In Chapter 5, we conclude the dissertation with summarized results for two studies that solidly support our claim and in the direction of future work.

Chapter 2. Related Work

To the best of our knowledge, no research has analyzed the security and privacy risks of PWAs and WebVR. Several studies focused on inspecting the usability and efficiency of PWA features across different environments [77, 78, 129]. T. Steiner [129] examined Web Views support on PWA features in Android and iOS, different from stand-alone browsers. The author evaluated feature supports across different devices and operations systems. I. Malavolta *et al.* [78] assessed the impact of service workers on the energy efficiency of PWAs and demonstrated the energy efficiency of these entities on selected mobile devices.

There also have been some studies of security and privacy on VR [4, 50, 144]. Vilk *et al.* [144] addressed new privacy threats posed in immersive environments. They revealed the privacy risks of using raw camera data or user gesture information, which could expose users' private data, such as room information or people around them. To address these threats, Adams *et al.* [4] established standards for the ethical developments of VR content by carrying out extensive user studies. George *et al.* [50] investigated information leakage that could occur when a bystander observes VR users. In particular, for password authentication, they showed that PIN and pattern types of passwords are comparatively safe from outside observers through a user study.

Unlike these studies, our work offers a better understanding of the security and privacy risks brought by PWAs and WebVR. In the rest of this Chapter, we present existing Web attacks (phishing and push attacks, side-channel leaks, online ad fraud) and techniques for sandboxing third-party ad libraries.

2.1 Phishing and Push Attacks

Phishing has been one of the most serious security problems for decades [26, 40, 59, 66, 86, 137, 158]. Phishing attacks share a basic form in which the attacker crafts a fake website that mimics the appearance of an authentic website. Due to its effectiveness and technical simplicity of conducting these attacks, phishing attackers utilize a tool to develop phishing sites for numerous phishing campaigns. Such a phishing tool is called a *phishing kit*.

Numerous studies has investigated phishing kits [26, 59, 137]. M. Cova *et al.* [26] focused on analyzing various methods, used by phishing kits, while X. Han *et al.* [59] proposed sandboxing live phishing kits to completely protect the privacy of victims. Thomas *et al.* [137] showed that 12.4 million people are potential victims of phishing kits, and 1.9 billion usernames and passwords are exposed via data breaches.

In a similar, but different context, phishing attacks using customized push notifications on mobile devices was studied [157], but not on Web push notifications. The authors have shown that abusing the notification customization may allow installing a Trojan application to launch phishing attacks or to anonymously post spam notifications. On the other hand, a secure Web push system was suggested by G. Saride *et al.* [113]. The authors strengthen the authenticity of web push messages with additional components between content providers and applications. However, we note that our push attacks which derived from the careless implementation of Web push protocols still hold under their proposed system.

2.2 Side-channel Leaks

Side-channel attacks have also posed a great threat to various Web applications [2, 20, 21, 45, 51, 69, 115]. Obtaining leaked sensitive information via a side-channel has been extensively studied. S. Chen *et al.* [21] took an advantage of the size distributions of transmitted packets to infer highly sensitive information (i.e. healthcare, taxation, web search queries), despite the presence of HTTPS protection. Another recent study [115] made use of packet burst patterns on encrypted video streams to fingerprint a video being streamed. P. Chapman *et al.* [20] proposed a way to measure the severity of information leakage in Web apps automatically.

On the other hand, using timing information has been a traditional mean of conducting side-channel attacks. It has been shown that the timing information of a user's browser that exploits Web caching [94], allows revealing the browsing histories [45]. However, timing information can be error-prone due to unreliable page fetch latency affected by a number of error sources, such as network condition, web server loads, and client loads. Recently, exploiting cross-origin HTML5 AppCache was proposed [69], which allows identifying cross-origin resource statuses such as determining the login status of a victim browser. Similarly, T. Goethem *et al.* [51] showed that a Web attacker can uncover users' identification such as Twitter accounts by inspecting the cross-origin resource size stored in AppCache. Our work exploits the cache which is a new attack vector to uncover a victim's browsing history on PWAs when the victim is offline.

2.3 Online Ad Fraud

Ad fraud refers to an operation that generates unintended ad traffic involving ad impressions or clicks. Numerous studies have explored methods of ad fraud [61, 80, 125, 136]. Springborn *et al.* [125] investigated abusive pay-per-view networks that expose fraudulent impressions via pop-under or invisible ads to increase the number of served ad impressions. Thomas *et al.* [136] identified ad networks that replace existing ad impressions to swindle advertising income from benign publishers. Huang *et al.* [61] suggested new clickjacking attack methods, such as cursor spoofing and white-a-mole, for click fraud. We introduce four new ad fraud attacks that harness features unique to WebVR (§4.5). We also investigate the feasibility of each attack and its effectiveness with carefully designed user studies (§4.6).

2.4 Third-party Ad Sandboxing

There have been extensive studies on sandboxing third-party libraries of publisher sites [3, 44, 63, 74, 82, 87, 104, 117]. AdJail [74] provides an isolation technique that inserts an ad script into another hidden shadow page that has a different origin than that of the publisher, and it adds the ad content to the host page via an inter-origin communication mechanism [97]. Third-party ad libraries in Mobile environments have also been sandboxed using a similar concept. AdSplit [117] provided a secure ad service by separating an ad library from a host app by running the library in another process. AdSentry [44] achieved the same goal using a different technique that modified the JS engine in the browser to prevent third-party code from interfering with the host's context. Politz *et al.* [104] proposed a language-based sandboxing technology, called AdSafety. They defined a subset of JavaScript and provided related safeguards through type-based verification. Instead of devising our own sandboxing system, our defense system is built on top of Caja [33], a mature open source project.

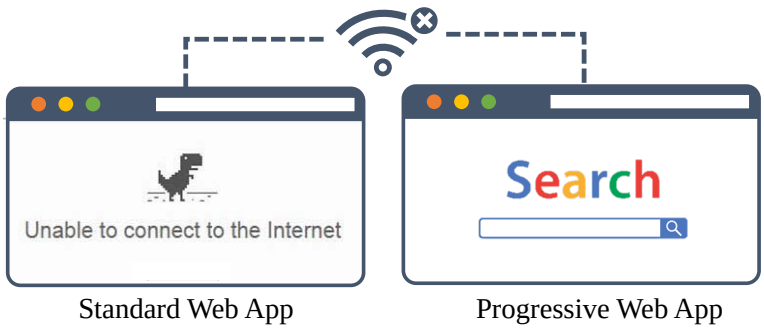
Chapter 3. Abusing Native App-like Features in Web Applications

3.1 Motivation

Progressive Web App (PWA) is a new generation of Web applications. It offers a seamless native-app experience when browsing a website that employs PWA features. Specifically, a PWA provides offline Web browsing experiences as well as interactive user services by making full use of cache [89], push notification [55] and service worker [56]. The harmony of these new HTML5 features blurs the boundary between native and Web applications particularly in mobile devices, promoting short-latency rich Web experiences.

Figure 3.1 illustrates two representative PWA features, push notification and offline browsing. A PWA site can send a Web push message, and a user's browser shows the push notification to notify the user as shown in Figure 3.1(a). Figure 3.1(b) shows a unique PWA feature offering an offline browsing experience, whereas a standard website supports no functionality when a browser is offline. Both push notification and offline usage features are built on the key technical component of service worker. A service worker is an event-driven Web worker that runs in the background. PWAs implement their native app-like features in various event handlers of service workers.

(a) An example of a Web push notification



(b) An illustration of offline usage

Figure 3.1: Representative features of PWAs

Google introduced PWA in 2015 and has encouraged website owners to migrate into PWAs [34]. Instantaneous installation, offline browsing experience, and user notification features attract website owners and motivate them to implement their sites with PWAs. Numerous Web services have promoted their PWA deployment success stories along with their technical advances [38]. Representatively, AliExpress and Flipkart, two large e-commerce sites, attested that their PWAs contributed to significant increases in the conversion rates and customers' shopping times [31, 32].

Despite the vast attention that PWAs have gained, to our knowledge, there has been no research that analyzed the security and privacy risks unique to PWAs. Previous research investigated spam and phishing campaigns [110, 131] and stolen credentials via Web phishing kits [137, 158]. Many researchers have also assessed the privacy risk of side-channel attacks, which allows network and Web attackers to learn visited websites and privacy-sensitive information [2, 20, 21, 45, 51, 115, 122]. However, all of these analyses were based on HTML5 features on standard websites and not on the unique risks brought by PWA features including push notification, cache, and service worker.

Our contributions. We conducted the first systematic analysis of the security and privacy risks on new HTML5 features unique to PWA. Furthermore, we addressed malpractices in third-party push services that expose PWA users to new phishing risks.

We carried out an empirical study on the prevalence of PWA websites on the Internet. By analyzing the front pages of the Alexa top 100,000 domains, we found 3,351 PWA sites using push notifications and 513 sites providing offline services. We collected a dataset for further analysis of PWA sites in the wild.

We started by analyzing the phishing risk via push notification, which has been overlooked in the context of PWAs. Based on the observed push notifications from the collected PWAs, we determined that 56% of PWA sites use their corporation or brand logos for their push notifications. This trend opens a door for a phishing attacker to imitate well-known brand logos for phishing via push notifications, causing users to misunderstand message senders. We found that several PWA websites have already conducted phishing attacks by exploiting WhatsApp and YouTube icons. Our finding assures that the domain name shown in a push notification is the only component that tells its recipient the origin of the notification sender.

Despite the importance of the domain name in a push notification, we found that popular browsers including Firefox for Linux-based desktop and Samsung Internet for Android do not show the domain name in a push notification, but only the thumbnail icon and message. Firefox for Android also shows no domain when the push notification panel is full of other notifications.

Furthermore, the current malpractice of prevalent third-party push services has been leading users not to check push notification domains. Based on the collected PWAs and third-party push services, all such services support push notifications on HTTP sites. Because all browsers allow only an HTTPS site to employ push notification, a third-party push service redirects a user to its own HTTPS site, then asks the user to grant push notification permission for the redirected website. However, the user usually has no clue how the redirected HTTPS site is associated with the HTTP site which the user visited in the first place. The user thus makes an uninformed decision based on the redirection and not based on its domain.

We also investigated eight popular third-party push services and their library scripts. While analyzing how they ensure the integrity of their push messages, we discovered a security flaw that allows a network attacker to spoof the domain of push messages. The addressed vulnerability is caused by their inherent design flaws, which expose PWA users to new phishing risks.

Cache is another core HTML5 feature that enables the browsing of PWA sites offline. A PWA site caches Web contents when a device is online, and uses the cached contents later when the device is offline. We propose a new side-channel attack that exploits the inherent PWA feature of offline browsing. The attack allows a Web attacker to learn the visited PWA sites of their victim. The attacker lures a victim to visit the PWA site, causing the instantaneous installation of the attacker's service worker on the victim's device. The PWA then loads other offline PWA sites within its iframes, the origins of which

differ from that of attacker's PWA site. The successful loading of a PWA within an iframe when the device is offline represents that a user has visited the PWA site before. We experimented on side-channel attacks on the collected PWA sites with diverse kinds of desktop and mobile browsers. We found that the Firefox Android and desktop browsers are vulnerable to our side-channel attack.

We introduce a way of abusing the persistency of a service worker. Because a service worker is able to perform arbitrary computations in the background even after a user leaves the PWA site, a Web attacker is able to abuse such a condition to complete their choice of computations. To demonstrate the practical usability of such an attack, we implemented a PWA site that mines cryptocurrencies with its service worker. We used push messages to distribute transactions and let the service worker verify each cryptocurrency transaction by finding the proper hash value. Thus, the attacker is able to abuse the computation resources of user devices that visit the attacker's PWA site. As a proof of concept, we mined Monreo coins [134] for 24 hours and verified 225,000 transactions by using one service worker.

We concluded with our proposed defenses that mitigate the identified security and privacy risks to guide PWA developers.

In summary, our contribution is as follows:

- We present the first systematic study on the security and privacy risks of PWAs from the Alexa Top 100K sites.
- We analyze the phishing risk via push notifications by inspecting 4,163 PWAs in the wild. We discover a security flaw by which the Firefox desktop/Android and Samsung Internet Android browsers show no push notification domain, thus exacerbating the phishing risk.
- We conduct an in-depth security analysis of eight popular third-party push services that cover 69.9% of PWAs from the Alexa Top 100K domains. We point out a malpractice that exacerbates the phishing risk and a design flaw that results in push domain spoofing.
- We introduce a new side-channel attack that abuses a cache. The attack allows a Web attacker to learn the victim's browsing history on PWAs. We demonstrated that our attack works on the Firefox browser.
- We present a new abusive attack that takes advantage of service workers. We implement a cryptojacking attack that abuses the computation power of each page visitor's service worker.
- We suggest mitigations for the addressed security and privacy risks.

3.2 Background

Progressive Web App (PWA) generally refers to a website that utilizes a list of new HTML5 features including the service worker, Web push, and cache features. Majchrzak *et al.* defined a PWA as a website that provides offline usage and a new user interface [75]. Because the definition is based on the execution behaviors of a website and depends on the completeness of feature implementations, we provide a simple technical definition of a PWA. Throughout the paper, we define a PWA as a website that registers a service worker at the browser of a page visitor. Because the service worker is a key technical component that enables native-app experiences including offline usage and push notifications, our definition captures all PWAs designed for various purposes.

3.2.1 Service Worker

A service worker is a new technology component that facilitates the main PWA functionality. It is an event-driven Web worker implemented in JavaScript [56]. An HTTPS website registers a service worker at a browser, binding the service worker to the HTTPS website origin defined by the HTTPS protocol, domain, and port. Thus, each service worker has its own Web origin that bounds internal resources through the same-origin policy (SOP).

A unique feature of a service worker is that each registered service worker runs in a thread that differs from the browser's main thread. Therefore, it runs in the background, independent of the main thread of the associated HTTPS website. In particular, the thread of a service worker runs persistently in the background even when a user closes the website associated with the registered service worker.

A service worker has an event-driven execution model, which requires implementing event handlers for various events exclusive to the service worker. For instance, *fetch* and *push* events are triggered when initiating an HTTP(S) request and receiving a push message, respectively. By leveraging these events and their event handlers, a service worker is able to intercept network requests from its main website, to receive push messages, and periodically to sync cached local contents with a server in the background.

Because the service worker is a fundamental component, a PWA site first registers its service worker when a user visits the website by calling the `navigator.serviceWorker.register` function. The service worker is then installed and activated in the browser of a page visitor without any disruption of granting permissions. The service worker becomes idle when all event handler operations are over, but it continuously wakes up every time when events for the service worker are invoked.

A service worker requires browser support. Currently, major browsers including Chrome 45+, Firefox 44+, Opera 32+, and Edge 17+ support service workers. For security concerns, the service worker is only supported on HTTPS websites. It indicates that each registered service worker script is delivered over TLS,—thus preventing a script injection from a man-in-the-middle (MITM) attacker who attempts to abuse the service worker functionality.

3.2.2 Web Push

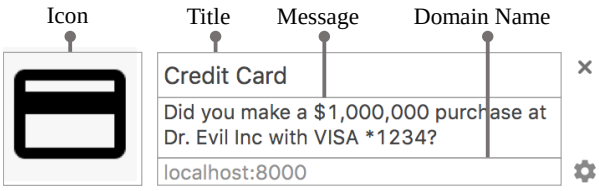


Figure 3.2: A general appearance of a Web push notification

A Web push notification is a fundamental PWA feature, designed to re-engage users with customized content [35]. Unlike mobile push notifications managed by user-installed applications, Web push notifications are controlled by desktop or mobile browser instances. Therefore, PWA site owners do not require users to install applications to show push notifications. In this paper, we focus on Web push notifications and use the term *push notification* interchangeably.

A push notification is a browser window alert that contains a push icon, a push message, and its sender's domain. There has been no standard UI for push notification, however WHATWG has specified a list of required elements including title, body, and origin [54]. Figure 3.2 shows the general appearance

of a push notification that most browser vendors implement. Many Web services, including Gmail, Facebook, and Twitter, have already deployed push notifications that inform users of important notices, or display interesting icons for users to click, thus re-engaging the users by redirecting them to particular web pages.

PWA visitors generally go through the following steps to receive a push notification. First, when a user visits a PWA site, the browser automatically registers a service worker of the site. The website then asks the user for permission to receive a push message. If the user approves, the website owner becomes able to send push notifications. The registered service worker running in the background receives a push message from the PWA site and shows a pop-up push notification to the user. The PWA site owner can still send push messages even after the user closes the PWA website tab or the browser window, as long as the browser process continues running.

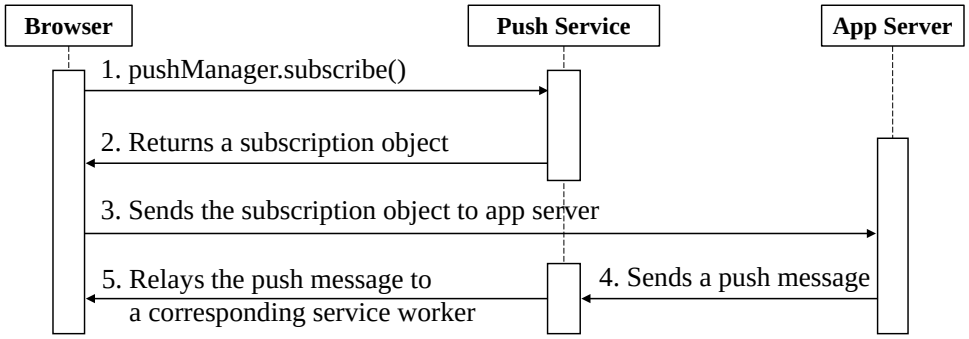


Figure 3.3: The basic procedure of a Web push notification

Figure 3.3 illustrates the basic procedure of how a Web push works. There is a new entity called *push service*, a sub-system that each browser vendor manage to support push notification services. Push service serves as a broker that receives push messages from a PWA website server and delivers them to the subscribed users.

1. When a user grants the push notification permission for a website, the user's browser is subscribed to its push service after the client-side script calls `pushManager.subscribe()`.

2. The push service then returns a *subscription object* that includes an *endpointURL* over TLS. The *endpointURL* is a capabilityURL, composed of the address of the push service and a unique identifier. This identifier represents the user's service worker, a recipient of push messages originated from the website.

3. The script at the client-side browser sends the subscription object to the website server.

4. With the subscription information, the website owner can send push messages to the subscribed users.

5. When the push service receives a push message from the website server, the push service resolves the unique identifier from the *endpointURL* and relays the push message to the corresponding service worker at a user's browser.

6. The user's browser wakes up the service worker, which is responsible for displaying the push notification by invoking a *push* event.

VAPID. The integrity of a push message depends on the secrecy of an *endpointURL* in the Web push protocol above. Consider that a website leaks an *endpointURL* at Step 3 in Figure 3.3 when the client-side script sends it. An adversary who obtains this *endpointURL* becomes capable of sending push notifications to the subscriber with the valid domain name of the website. Because the basic push protocol does not bind an *endpointURL* to its creator, a PWA owner, anyone with a valid *endpointURL* can send a valid push message to the subscriber that the *endpointURL* indicates.

VAPID, a Web Push protocol extension, is designed for a push service to authenticate an application server that sends a push message [39]. When VAPID is employed, the push service blocks push messages from entities without proper authentication.

Specifically, VAPID utilizes an asymmetrical key pair. The public key, termed *applicationServerKey*, is passed to a push service when a service worker subscribes to push service (see Step 1 in Figure 3.3). When the PWA owner sends a push message, the owner signs the push message with the private key and sends it to the push service. The push service checks the validity of push messages with the stored public key and relays the push message with their valid signatures.

Unfortunately, using the VAPID protocol is not a requirement. It is optional for each PWA developer to check the authenticity of push message senders via VAPID.

Push Message Encryption. The Web Push protocol also supports encrypting a push message payload so that a push service is unable to see its content while relaying the push message.

When a client's browser sends a subscription object from a PWA website to its server (see Step 3 in Figure 3.3), the browser appends the two keys *auth* and *p256dh* to the subscription object and sends it to the server. *p256dh* is a client public key that the PWA server uses to encrypt a push message payload. *auth* is a shared authentication secret between the PWA server and the client. Thus, a subscription object that consists of *endpointURL*, *p256dh* and *auth* should not be tampered with or directly inspected by any entity except for the PWA server from which a user elects to receive push notifications.

3.2.3 Cache

The network dependency of Web applications has hindered browsing experiences. The offline Web Application (or AppCache) is one of the attempts to free Web applications from inherent network dependency. AppCache [152] enables a Web application to cache resources in local storage for offline access. However, it is error-prone, and also hard to provide a complete offline experience because of the overhead of managing numerous manifest-typed resources. It is being deprecated by most browser vendors [88].

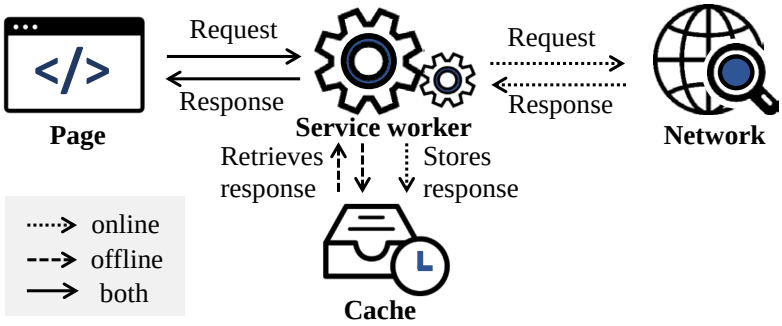


Figure 3.4: An illustration of cache usage

Recently, a new HTML5 feature, termed *cache*, was introduced. Cache [89] is an origin-bounded

local storage that is accessible regardless of the network status. This new feature becomes more powerful when combined with a service worker. For example, as shown in Figure 3.4, a service worker can either load resources from the cache storage or fetch them through the online network according to the network conditions. These programmable interfaces dramatically improve the online and offline browsing experiences of Web application users.

Cache is supported by most major browsers including Chrome 46+, Firefox 44+, Opera 33+, Safari 11.1+, Edge 16+, and also Samsung Internet 4+ for mobile environment. We cover all of these browsers in our experiments.

3.3 A Methodology of Collecting PWAs

Despite the wide attention that PWA has gained, there is little information on the current deployment of PWAs on the Internet. This lack of information hinders understanding the security and privacy impacts brought by vulnerable PWAs.

We investigated the front pages of the Alexa top 100,000 domains and collected PWAs in the wild. Recall that our definition of PWA is a website that registers a service worker (see Chapter 3.2). For each main page of the 100,000 domains, we checked whether a website registers a service worker of its own.

We ran a script that forces a Firefox desktop browser to visit 100K websites sequentially. We then extracted all registered service workers shown in the `about:debugging#workers` page, and crawled the JS files that registered service workers.

We observed that scripts from several third-party push services often registered their service workers only after certain user interactions such as clicking on the *allow* button placed in the css-styled permission dialog as shown in Figure 3.8 (a). To cover such websites with third-party push services, we first identified third-party push services among the Alexa top 100K websites.

For each of the crawled JS files, we checked its source domain and found prevalent domains appearing across the crawled JS files. We then performed keyword search at Google with such prevalent domains to check if the domains are third-party push vendors. We found 2,911 websites with third-party push services. For those websites, the authors manually visited them and clicked buttons that grant push permission.

We conducted a further analysis to check whether a PWA uses a cache. We modified the Firefox browser to emit the logs when accessing any cache object. With the modified Firefox, we visited each PWA identified from the previous step and decided whether the PWA uses a cache.

Our collection method has limitations. It may miss PWAs that require certain user events to register service workers. Such events may include clicks on certain DOM elements or keyboard events. However, the missed PWAs pose less of a threat because it becomes more difficult for an attacker to exploit their service worker, push notification, or cache.

Table 3.1 shows the statistics of our collected PWAs. Among the Alexa top 100,000 domains, 4,163 are PWAs that install service workers at the browser. Among the 4,163 PWA websites, 3,351 (80.5%) use push notifications and 513 (12.3%) use the offline cache functionality. Others represent websites with service workers that uses neither push notifications nor cache.

We observed that 2,911 sites (69.9%) of the PWAs implement the push notification functionality by deploying scripts from third-party push services. These services offer script libraries so that a standard website is able to support a push notification by embedding one of their libraries. The top eight most prevalent services are OneSignal [102] (2,046 sites), SendPulse [116] (364 sites), Pushcrew [106] (126

Table 3.1: PWA statistics for the Alexa top 100,000 sites

Features Used	**# of Websites (% Percentage)**
Push	440 (10.6%)
Push with library	2,911 (69.9%)
Cache	513 (12.3%)
Both	196 (4.7%)
Others	495 (11.9%)
Total	**4,163 (100%)**

sites), Izooto [64] (65 sites), Pushengage [107] (53 sites), Pushwoosh [108] (47 sites), Foxpush [48] (28 sites), and Urbanairship [9] (20 sites). They cover 86.9% of PWAs out of 3,351 sites that support push notification. Our analysis on PWAs in the wild confirms the prevalence of third-party push services, which also pose security and privacy risks caused by their potential vulnerabilities.

3.4 Threat Model

We assume two attack models: *PWA attacker* and *Network attacker*.

PWA Attacker. PWA attacker is a classic *Web attacker* [14]. The attacker controls his/her own PWA website and entices users into visiting the website. The attacker's service worker is instantly registered at a victim's browser once a victim visits the site as explained in Chapter 3.2.1.

We additionally extend the Web attacker model and assume that a user may grant permission for push notifications on an attacker-controlled PWA. As a result, the attacker has the ability to *send* and *customize* push messages which notify their visitors even when they are not on the attacker-controlled PWA. Furthermore, the attacker has no limitation of abusing their own service worker and cache.

Network Attacker. We assume an active network adversary who is capable of monitoring, intercepting and modifying network traffic over the HTTP protocol. Specifically, the attacker can eavesdrop on messages as well as alter HTML or JS code sent over the HTTP protocol. In previous research [20, 21, 81, 115, 118], an active network adversary has shown to be a practical threat to Internet users, exfiltrating passwords and inferring online behaviors. We assume that a network attacker can monitor or selectively revise an HTTP website with a third-party push library.

3.5 Phishing via Push Messages

Phishing is one of the most effective and devastating Web threats that harvest users' credentials as well as privacy-sensitive information [137]. A PWA attacker can launch a phishing campaign by abusing push notifications. The attacker entices users with innocuous Web content and requests push permissions on the attacker-controller PWA. Later, the attacker crafts a push message with her choice of destination URL to redirect victims, and then sends it to all past visitors. All past visitors with service workers from the attacker's PWA receive the phishing push notifications that redirect the victims once clicking the notifications.

From the perspective of a phishing attacker, a Web push is a juicy content delivery system. Phishing

Table 3.2: Push icon usage statistics for 694 PWAs

Push Icon Category	**# of Websites (% Percentage)**
Company/Brand Logo	390 (56.2%)
Article Thumbnail	226 (32.5%)
Default (Bell-shaped)	22 (3.2%)
None (Blank)	56 (8.1%)
Total	**694 (100%)**

via push messages has two advantages over classic email phishing: (1) the attacker can actively show a push notification at a time of her choice, and (2) it is difficult for a push message recipient to determine the origin of a received message. Because a push notification pops up even when a victim is not on the attacker-controlled PWA, a phishing attacker can effectively show a push notification at the time that the victim is most likely to click the push notification.

The only information for a push message recipient to know the message origin is the domain appearing in the push notification dialog. However, its portion in the dialog is relatively small compared to other visible components (See Figure 3.2). Note that the previous research demonstrated that users paid little attention on a small display in the peripheral area of a browser, compared to the large main window [150, 156]. It is also highly likely for users to place little attention on a push notification domain.

In this Chapter, we introduce a phishing method via push message that exploits the current trend of using company and brand logos for push notification icons. We also present browser security flaws of showing no domain name in a push notification, thus exacerbating the phishing risk.

3.5.1 Phishing by Manipulating Push Notification Icons

Generally, a push notification has a domain name component that indicates where the push message originated. We argue that besides a domain name, a push icon contributes to the user's understanding of the origin of a received push message. We collected push notification icons from 3,351 PWAs from our dataset (see Chapter 3.3). Because we have no control over enforcing such PWAs to send push messages, we collected icons from the received push messages for three days.

Among the 694 websites that showed push notifications, 390 (56%) sites used their corporation logos for push icons as shown in Table 3.2. 32% of the domains use push icons for summarizing articles, or advertising products. Thus, it is natural for users to educate themselves to infer a push message sender based on its push icon.

While examining push icons, we came across real-world push notifications that attempted phishing as well as, two domains that imitate popular brand logos including WhatsApp and YouTube for their push icons. Figure 3.5 shows such captured instances. megafilmesonlinehd.org uses the YouTube icon to welcome their subscribers. pornkino.to promotes online-dating opportunities in German with the WhatsApp logo. We note that the Chrome logo displayed in the third push notification appears in the Chrome browser under the MacBook environment.

We also received a push notification claiming "New IPhone X is reserved for you. Delivery to your doorstep for 1$ only!" with an IPhone image as a push icon. Another phishing example with the Chrome icon says "Google Chrome Premium," enticing users to click on the "DOWNLOAD" button, which leads

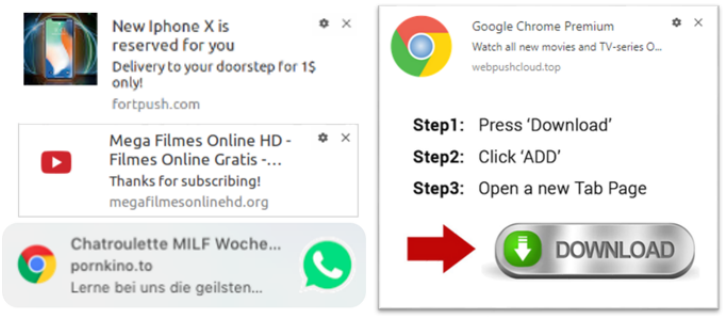


Figure 3.5: Real-world push examples that imitate popular brand logos and phishing attempts

to installing a Chrome extension.

Our findings confirm that phishing via a push message targets Web users and leads users to misplace their trust by manipulating push notification icons. Therefore, the only way for users to know the authentic sender of a push notification is to check its domain name.

3.5.2 Domain Name in a Push Notification

A domain name in a push notification should be visible because it is the only component for users to check the origin of a received push message. We conducted a comprehensive study of investigating how the domain in a push notification is shown under various execution environments.

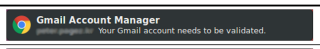
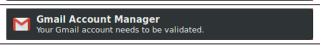
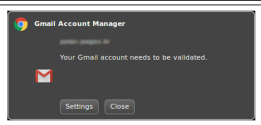
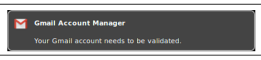
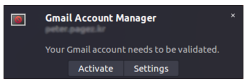
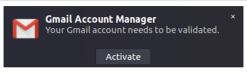
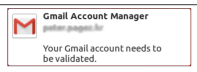
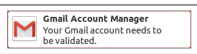
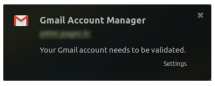
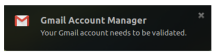
We examined the Firefox, Chrome, Opera and Edge browsers under the Windows 10, Ubuntu 16.04, and MacBook Sierra 10.12.2 operating systems. For mobile browsers, we checked UC Browser, Opera, Brave, Firefox, Chrome and Samsung Internet on Android. Note that the Apple push notification service is revoked recently [29]. Therefore, we excluded the Safari browser and mobile browsers in iOS environment from our study.

We found that the Firefox desktop browser under five Linux-based environments shows no domain in the push notification (Figure 3.6(a)). Because the desktop browsers in Linux-based environments use an external OSD (On-Screen Display) to show push notifications, they make use of the D-Bus (Desktop Bus) to pass a push notification message to the external OSD. We intercepted RPC calls from browsers to the external OSD varying different desktop environments. We found that Firefox under *GNOME, Ubuntu MATE, Cinnamon, Budgie*, and *Pantheon* doesn't pass the site URL on a push notification message while Chrome and other browsers do. As a result, Firefox desktop browsers in such Linux-based environment do not show the domain information in their push notifications.

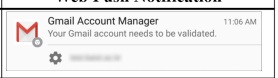
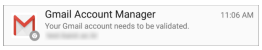
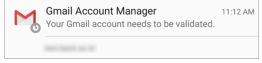
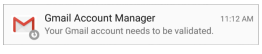
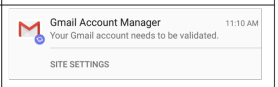
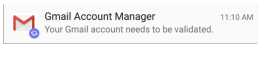
For Android browsers, we found that the Firefox browser shows no domain in certain cases and the Samsung Internet browser always shows no domain in their push notifications (Figure 3.6(b)). When an Android device is locked or the notification panel is full, Android abbreviates push notifications. Otherwise, it displays notifications with more details such as a settings button. The Firefox browser on Android shows no domain in the first case. The Samsung Internet browser never shows a domain in their push notification.

Figure 3.7 shows our phishing message displayed on Firefox and Samsung Internet on Android. We implemented the phishing push message to induce a victim to change the password of their Gmail account. The notifications show the Gmail logo without its domain origin, which would reveal the attacker's domain when displaying them in these browsers.

We reported these security flaws to Mozilla and Samsung, and they were both fixed correctly as a

Desktop Environment	Browser	Web Push Notification
GNOME	Chrome	
	Firefox	
Cinnamon	Chrome	
	Firefox	
Budgie	Chrome	
	Firefox	
Pantheon	Chrome	
	Firefox	
MATE	Chrome	
	Firefox	

(a) Push notifications on three different browsers on Android

Browser	Notification Panel Condition	Web Push Notification
Chrome	Not busy	
	busy	
Firefox	Not busy	
	busy	
Samsung Internet	Not busy	
	busy	

(b) Push notifications on five different desktop environments

Figure 3.6: Push message display difference

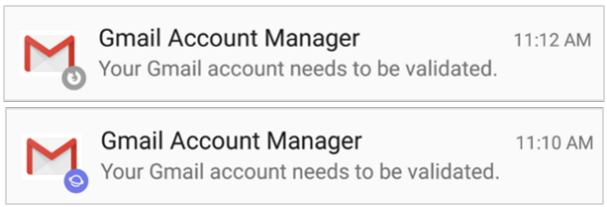


Figure 3.7: Crafted push notifications in the Firefox and Samsung Internet Android browsers

result.

We acknowledge that a phishing victim who already clicked a phishing push notification may still see the full URL of a redirected website before entering sensitive information. However, we argue that phishing via a push message is a critical threat. Thomas *et al.* showed that popular Web phishing kits harvest 230K credentials every week [137]. Phishing websites emulating Gmail, Yahoo, and Hotmail logins have managed to steal 1.4 million credentials despite the victims' browsers not showing any valid service domain. A well-crafted phishing push message with no message origin certainly favors the chance of a successful phishing attack.

3.6 Risk of Third-party Push Services

In this Chapter, we address security risks that arise from a third-party push service. Such a service provides a convenient and fast way of enabling push notifications at their client's websites. Generally, a website owner includes a script from a third-party push service, which automatically performs a series of procedures that enable push notifications. The site owner sends a push message to their subscribers by

utilizing the Web interface provided from the third-party push service. The site owner can also customize push message titles, message, and icons by utilizing a handy interface provided by the third-party push service.

We analyze the current practice of enabling push notifications on HTTP websites by third-party push services. Chapter 3.6.1 explains that the unhealthy practice of redirecting users from a client HTTP site to a third-party HTTPS website has been leading a user to misunderstand the valid origin of a push message that the user wants to receive. A phishing attacker is certainly able to exploit such misunderstandings against innocuous users.

We also investigated how third-party push services handle a push subscription object to preserve its secrecy. Chapter 3.6.2 describes two security design flaws that allow spoofing a push notification domain by a network attacker.

3.6.1 Prejudice against Third-party Domains in Push Notifications

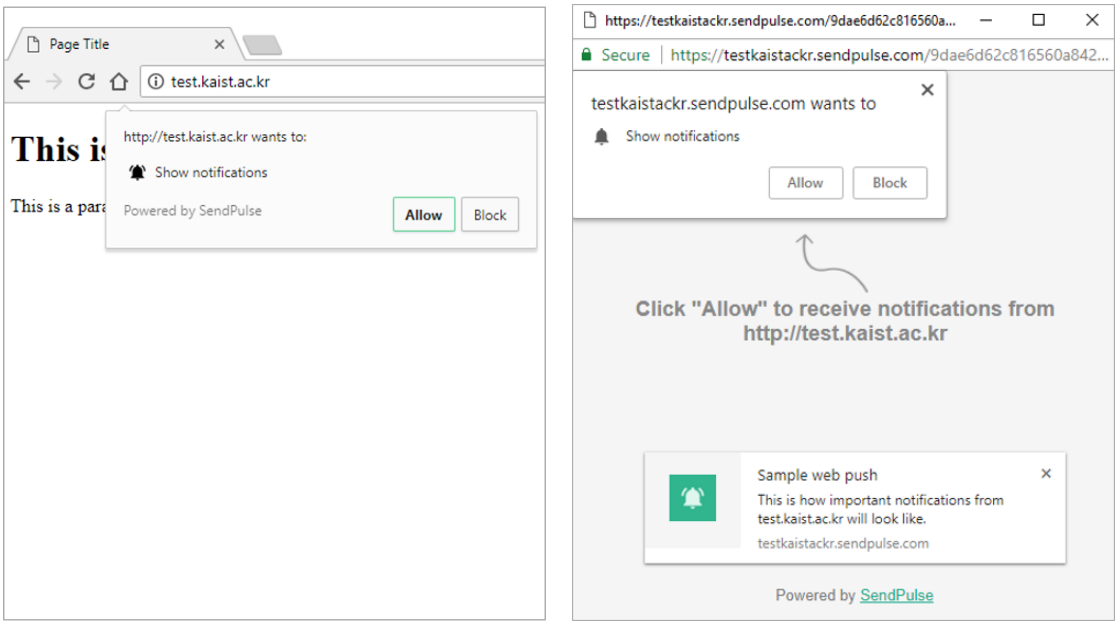
Popular third-party push services provide various services including sending a push notification, scheduling a push notification, and reporting the statistics of subscribers. One of the most common supports is to enable push notifications for HTTP websites.

As mentioned in Chapter 3.2.1, only HTTPS sites are able to register their service workers. Because the presence of a service worker is mandatory to show a push notification, HTTP websites are intrinsically unable to show a push notification. Third-party push services bypass this restrictions by placing a service worker for their own HTTPS domain. For each HTTP site that embeds a script from a third-party push service, the third-party push service assigns an HTTPS domain, a subdomain of their HTTPS domain. The third-party push service then enables the HTTP site visitors to receive push messages from this HTTPS subdomain.

Figure 3.8 demonstrates this trust transition in two steps. (1) A user visits an HTTP website that implements push notifications using a third-party script from a third-party push service. The script shows a css-styled dialog that asks the user to accept push messages from the HTTP website. It is noteworthy that the css-styled dialog is not a browser dialog asking for push permission, but a notifying window to inform the user. (2) If the user clicks on "allow", the script redirects the user to the subdomain of the third-party push service HTTPS domain, assigned to the HTTP website. The redirected HTTPS website then pops up a browser dialog asking the user to grant push permission for the HTTPS domain. For HTTP websites with third-party push services, a user who seeks push notifications should give her/his consent twice.

Risk. The problem arises from users' ignorance of the relationship between an HTTP website and the third-party push service that the HTTP website uses. Users may understand the first consent request because the consent seeks the push permission for the visited HTTP website domain. However, the HTTPS domain name that appears in the second permission dialog partially matches the prefix of the HTTP website domain or uses a random domain prefix with the third-party push service HTTPS domain suffix. Such domain relation between an HTTP website and its third-party push service domain is chosen by the HTTP website owner and not by the website visitors. It is natural for HTTP website visitors to be ignorant. Based on the redirection from the HTTP website to its corresponding HTTPS domain, users should decide whether to accept push notifications from the third-party push service HTTPS domain, of which they may be unaware.

We argue that the current practice of getting a push consent by redirection contributes to the trend of not checking a domain name for granting the push notification permission. Normal Internet users



(a) A css-styled permission dialog on a HTTP website that user visited

(b) A push permission dialog on a HTTPS websites that library provided

Figure 3.8: An example of the two-step push permission granting procedure

have no way to understand this complicated trust transition chosen by a HTTP site owner, but make an uninformed decision based on the redirection and not on the HTTPS domain in the permission dialog.

Furthermore, a network attacker can take advantage of this trust transition from an HTTP domain to an HTTPS domain. Consider that the network attacker changes the redirection URL after the first consent window from a valid third-party HTTPS domain to the attacker's HTTPS domain. A page visitor should decide whether to receive messages from the attacker's HTTPS domain. Unless the victim who visited the website knows the valid third-party push service domain in advance, the victim naturally trusts the attacker's HTTPS domain based on the fact that the first push permission consent redirects the victim to the attacker's domain. A phishing attacker who seeks the push permission consent on the attacker-controlled HTTPS domain can exploit this malpractice by changing the redirection URLs of popular HTTP websites with third-party push services.

We demonstrated the attack of changing the redirection URL on websites with popular push services in Chapter 3.9.1

3.6.2 Domain Name Spoofing in a Push Notification

We investigated the VAPID protocol deployment in popular third-party push services. Based on the occurrences of third-party script sources in the collected PWA (see Chapter 3.3), we selected the eight most prevalent third-party push libraries and checked whether they use *applicationServerKey* when they subscribe to push service (see Chapter 3.2.2). Unexpectedly, among the eight third-party push libraries, only two (OneSignal and Urbanairship) implement their Web push systems with the VAPID protocol. One explanation for its low adoption rate is that the VAPID protocol requires an additional step of performing the ECDSA p-256 signing on push messages, which brings performance overheads [133] on vendors' push servers.

When no VAPID protocol is present, the only required component for a phishing attacker to send

a forged message with a spoofed domain is a subscription object leaked from the target domain (see Chapter 3.2.2). Thus, we further investigated a possible leakage of subscription objects accessible to a network attacker. We analyzed in/outbound network payloads from/to PWAs with the six third-party push services that do not deploy the VAPID protocol. We used mitmproxy [84], an open-source interactive HTTPS proxy, to inspect and modify Web traffic to mock the capability of a network attacker.

Because a third-party push service internally uses a browser-provided push service[1] underneath a curtain, the subscription object created at the client-side should be delivered to a third-party push service by any means. Therefore, we focused on the subscription object transmission channel from a service worker at the client-side to a third-party push server.

We found two leakage paths that allow a network adversary to obtain the complete subscription information: (1) the transmission of a subscription object over HTTP, and (2) the reflected transmission of a subscription object over HTTPS.

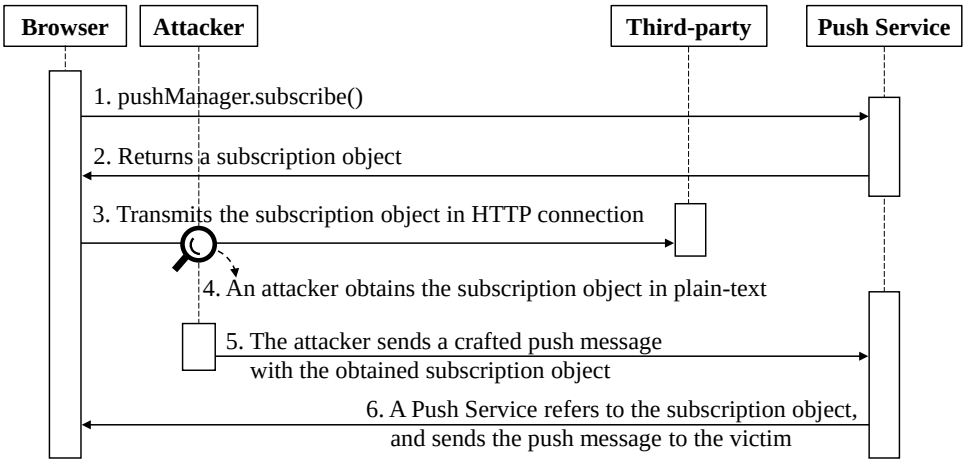


Figure 3.9: An exploitation of a subscription object leaked over HTTP for a push domain spoofing attack

Transmission of Subscription Objects over HTTP. The first leakage path is where a subscription object is sent over HTTP. Any network adversary is capable of harvesting such a subscription object in plain-text. We found that the *Izooto* [64] push service corresponds to this case. Figure 3.9 describes the overall process of how a network adversary sends a push message with a spoofed domain in a push notification. Consider a vulnerable website with the Izooto library. After the Izooto script at the client-side generates a subscription object after Step 2, it sends the subscription object to its push service server over HTTP. A network attacker inspects this transmission and extracts *endpointURL* in the subscription object. She can send a push message through Steps 5 and 6 and the recipient will see a push notification, the domain of which shows *subdomain.izooto.com* assigned to the target HTTP domain.

Reflected Transmission of Subscription Objects over HTTPS. We present a new attack that exfiltrates subscription objects over HTTPS. The attack exploits a design flaw in popular third-party push libraries. According to our analysis of the six third-party push libraries, the SendPulse and Pushwood third-party libraries use a variable to hold the destination HTTPS URL for a subscription object to be sent, as shown in Listing 3.1. However, the problem is that the script that holds this variable is sent over HTTP so that the network adversary changes this variable.

[1] https://fcm.googleapis.com/ and https://updates.push.services.mozilla.com/ are the addresses of push services for Chrome and Firefox respectively.

```
var n="https://pushdata.sendpulse.com:4434";
```

Listing 3.1: A script of defining the subscription object destination URL from SendPulse

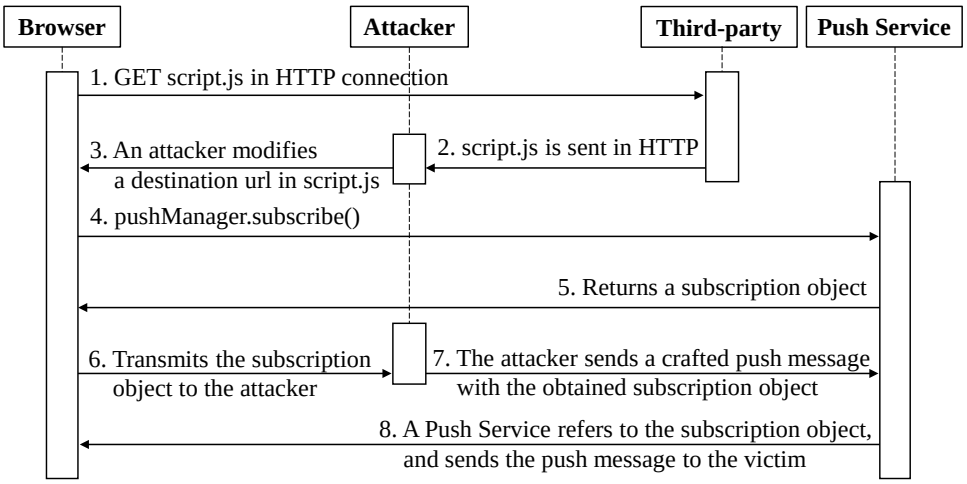


Figure 3.10: An exploitation of a reflected subscription object sent over HTTPS for a push domain spoofing attack

Figure 3.10 illustrates the overall network flow of our attack. During Step 6, the network attacker obtains the subscription object, the result of a reflected request originated from script.js altered by the attacker in Step 3. Steps 7 and 8 show that the attacker sends a push message by abusing the obtained subscription object.

We conducted the experiments which exploited both leakage paths on real-world PWAs and successfully sent push messages with a spoofed domain name. Chapter 3.9.2 explains the details of our attack and its results on third-party push libraries.

3.7 Side-channel Attack on browsing history

History sniffing attack that leaks a Web user's browsing history has been considered a critical privacy threat [45, 122, 149]. The inferred browsing history can reveal its owner's personal interests, political preferences, medical history, dating preferences, and so on.

In this Chapter, we present a new method that a PWA attacker can use to infer the browsing history of PWAs where his/her victim visited in the past. This new side-channel attack takes advantage of the cache, which a PWA uses to support offline browsing usage. In this attack, we assume that a victim already visited the attacker-controlled PWA and that its service worker automatically stores the attack code in the cache for its offline usage.

Attack. When a victim opens the attacker-controlled PWA in offline, the attack PWA prepares multiple iframes whose sources are the HTTPS URLs of the target PWAs. The attacker also registers an *onload* event handler for each iframe so that the top attacker-controlled PWA knows the loading completion of a cross-origin target website in each iframe.

If the victim visits a target PWA that supports offline usage, an *onload* event handler will be called. Otherwise, an *onload* event handler will not be invoked. We tested our attack against Chrome, Firefox, Safari, Edge, Internet Explorer, UC Browser, Opera and their Android versions as well.

We confirmed that our side-channel attack is effective on the Mozilla Firefox 59.0.2 (Windows 10, Ubuntu 16.04, and High Sierra 10.13) browser. Fortunately, unlike Firefox and Safari, all other browsers invoke their *onload* event handlers regardless of whether the loading of a target PWA is successful or not. In the case of Safari browser, it manages cache storage separately when loading inside an iframe.

The difference in the handling the *onload* event stems from each user agent's event handling policy, and not from simple implementation bugs [18, 23]. The living HTML standard describes that *load* event should be fired when a *Document* in an iframe is completely loaded [151]. It also states that it is up to user agents to implement a strict cross-origin policy of firing the event when loading cross-origin resources within an iframe. However, such a policy may not aligned with existing Web content. Our attack exploits this subtle policy difference in the context of PWA offline usage.

The proposed side-channel attack has several limitations. Because of its dependency on the cache, the attacker can only infer visited PWAs that offer offline usage. Frame busting techniques, X-Frame-Options header [111], and Content Security Policy [53] also make our attack ineffective.

The proposed attack also has unique advantages over previous history sniffing attacks [45, 149]. (1) *Accuracy*: Our attack is more accurate than a sniffing attack that exploits the load time differences on cached resources [45]. It is well-known that exploiting the loading time differences is not practical because the loading time is greatly affected by network environments [69]. On the contrary, our attack is deterministic due to its simplicity of checking for offline usage support from a target site. (2) *No outgoing requests*: Because the attacker conducts the attack in the offline mode, there is no outgoing network request toward a target PWA with any referer header that reveals the attacker's domain. This makes the detection of our attack difficult. (3) *Coverage*: As the offline usage prevails among PWAs in the wild, the coverage of our attack becomes larger.

Above all things, our side-channel attack is a brand new category of history sniffing attacks unique to PWAs.

3.8 Abusing Service Worker Persistency

A service worker persists in performing event handlers until they are complete even after a user closes or leaves its website. This persistency is a key requirement when syncing local Web contents in the background and showing push notifications in time. At the same time, a PWA attacker is able to abuse such persistency to perform arbitrary computations. The attacker entices a victim to visit an attacker-controlled PWA and thus installs a service worker onto the victim's host. At this point, the attacker is able to perform arbitrary computations on the victim's machine by triggering registered event handlers in the service worker.

Fortunately, there are limitations to abusing PWA service workers. Major browsers such as Chrome and Firefox provides limited built-in browser objects and API for a service worker to access. For instance, Web socket [153] , GPS, and, gyro sensors are inaccessible from a service worker. The *SetTimeOut*, *SetInterval*, and *XMLHTTPRequest* built-in methods are also unavailable.

In this Chapter, we demonstrate a crypto-jacking attack that abuses service workers regardless of how limited built-in objects and APIs are provided to them. The proposed attack is designed to exploit computation resources of victims who once registered a service worker from an attacker-controlled PWA.

Cryptocurrency Mining. Cryptocurrency mining has become a popular way of utilizing surplus computing resources [70]. It also becomes an alternative way of monetizing a popular website instead of exposing advertisements that can annoy the website visitors. The website assigns each visitor a list of

cryptocurrency transactions to verify and the visitor's browser then finds valid hash values that validate the assigned transactions by performing numerous trial-and-error hash computations. CoinHive [24] is a popular JavaScript cryptocurrency mining service for website owners who seek mining opportunities from their website visitors.

Once a website embeds a CoinHive cryptocurrency mining script, a host browser that renders the website becomes a cryptocurrency miner. The miner initially connects to a central CoinHive mining pool and then receives a list of transactions to validate via WebSocket [153]. It then runs multiple Web Workers [57] that validate the received transactions. CoinHive also requires browser supports for WebAssembly [99] to make full use of computation resources. If the miner finds a valid hash value, the miner script sends the hash value to claim its reward for the performed computation.

Attack. A PWA attacker is capable of abusing a service worker with push messages when validating cryptocurrency transactions, thus mining coins. The benefits of using service workers for mining cryptocurrencies are two-fold. (1) The attacker has no need to compromise the user's local machine, but requires a victim to visit her PWA and gets the consent for a push notification. (2) The attacker is able to continuously mine cryptocurrency coins even after the victim leaves the website.

To demonstrate the feasibility of mining coins via service workers, we implemented a service worker that mines Monero coins [134]. We refactored the CoinHive mining script to make it workable by a PWA service worker. Instead of WebSocket to fetch transactions from a CoinHive server, we used a cross-origin *fetch* API to make a HTTP request to our proxy server where communicating with the CoinHive server via WebSocket.

The technical challenge of using a service worker for cryptocurrency mining is to keep the service worker running for a long time. Once the service worker registration completes, it lives in a browser "indefinitely" and the browser instantiates a new service worker process when there is an associated event including push event. The process runs continuously in the background even if the tab on the corresponding website is closed. The Chrome browser terminates this service worker process if it has been idle for 30 seconds [42].

Due to the nature of cryptocurrency mining, a service worker cannot start with a long list of transactions to work with because other miners may validate those transactions before the service work completes the task. Therefore, we use a push messages to distribute cryptocurrency transactions as well as to wake idle service workers.

An unfortunate downside of exploiting push messages is that push messages trigger displaying push notifications, which is undesirable for a stealthy mining operation. We thus investigated how not to show a push notification when a service worker receives a push message.

A straightforward way is not to purposely call any Notification API (i.e., `showNotification()`) upon receiving a push message to hide its push notification. We tested our method against all browsers supporting Web push: Whale, Edge, Brave, UC Browser, Samsung Internet for Android, Chrome, Firefox and Opera. We confirm that only UC Browser, Firefox and Edge allow receiving push messages without displaying any push notification. The other browsers show a default warning notification. Chrome shows the message: "*This site has be updated in the background.*"

We observed that Firefox, and Edge revoked their push subscriptions if a service worker ignored displaying a push notification 15 and 3 times, respectively upon receiving a push message. UC Browser did not revoked its subscription as well even when showing no push notification for 100 push messages. Therefore, to maintain continuous stealthy mining operations, we periodically renewed the subscription objects after receiving several consecutive transactions via push message. However, we found that Edge

Table 3.3: Monero mining rewards for 24 hours by a single service worker

Monero price(Apr 23, 2018, close): $283.30

Browser	Execution Environment	Number of Solved Hashes		Amount of Monero	
		Total (24h)	Average (1h)	Total (24h)	Average (1h)
Chrome 65	Windows 10 Desktop	225,024	9,376	0.00001266 ($0.00358657)	0.00000053 ($0.00014944)
Firefox 59	Windows 10 Desktop	195,840	8,160	0.00001119 ($0.00317013)	0.00000047 ($0.00013209)
Chrome 65	Android 8.0 Google Pixel Phone	50,176	2,091	0.00000282 ($0.00079891)	0.00000012 ($0.00003329)
Chrome 65	macOS High Sierra 10.13.4	138,496	5,771	0.00000778 ($0.00220407)	0.00000032 ($0.00009184)

does not allow re-subscription on the background and UC Browser does not support WebAssembly, which the CoinHive mining script requires. Therefore, our mining attack works against Firefox for a stealthy mining operation.

A PWA attacker is not necessarily limited to conducting her mining attack against victims with Firefox. She is able to conduct a cryptocurrency mining campaign at the time when victims are not likely to be present such as 3:00 AM.

Table 3.3 shows the experimental result of mining Monero cryptocurrency for 24 hours only by using one service worker. The experiments are performed on *MacBook Air* with 1.3 GHz Intel Core i5 processor (4250U) and 8 GB memory machine, *Windows10* desktop with 3.6 GHz Intel Core i7 processor (7700) and 16GB memory, and Google Pixel Phone. The CoinHive mining algorithm is not optimized in ARM architecture [112], thus resulting in poor performance in the Android 8.0 Pixel device. Using one service worker for mining coin is not as efficient as using multiple Web workers. However, the service worker persists even if a user leaves its website. The more victims visit the website, the more computation capability the attacker has. The attacker is capable of building her/his own service worker botnet, designed to mine cryptocurrencies, neither compromising victims' machines nor letting victims install malwares.

3.9 Attacks on PWA in the Wild

In this Chapter, we demonstrate the feasibility of the push permission delegation attack in Chapter 3.6.1, the push domain spoofing attack in Chapter 3.6.2, and the side-channel attack via cache in Chapter 3.7 against real-world PWAs.

3.9.1 Push Permission Delegation Attack

In this Chapter, we demonstrate a push permission delegation attack that redirects a user to an attacker-controlled site. The presented attack exploits the ignorance of a victim about the relationship between a visited HTTP website and its redirected HTTPS website.

As explained in Chapter 3.6.1, a user should give consent twice to grant push permissions on an HTTP website that uses a third-party push library. We checked whether a redirection URL is spoofable by a network attacker.

We investigated the eight most popular third-party push libraries (See Chapter 3.3). We confirmed that a network attacker is certainly able to manipulate the redirection URLs from Foxpush, SendPulse, Pushwoosh, and Izooto since these library scripts are delivered through HTTP. However, our attack is not necessarily limited to vulnerable third-party push libraries. Because a network attacker has the

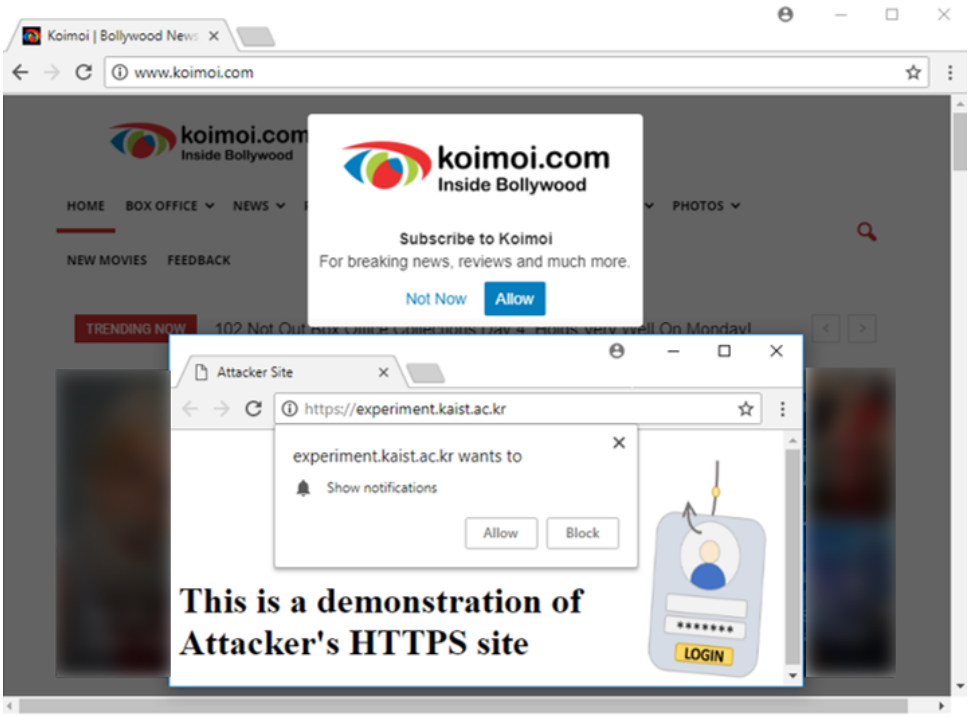


Figure 3.11: A demonstration of a push permission delegation attack

ability to change the intended semantics of an HTTP website, the attacker can block the consent dialog shown by any third-party push library and display their own consent dialog with the choice of redirection URL.

Figure 3.11 shows a successful attack launched against http://www.koimoi.com that deploys the Pushwoosh library. An attacker takes advantage of the blind trust transition of users from http://www.koimoi.com to https://a756c-03273.chrome.pushwoosh.com[2]. The attacker can modify the redirection destination from https://a756c-03273.chrome.pushwoosh.com to https://experiment.attacker.com so that victims will grant push permission to the attacker-controlled domain.

3.9.2 Push Domain Spoofing by EndpointURL Hijacking

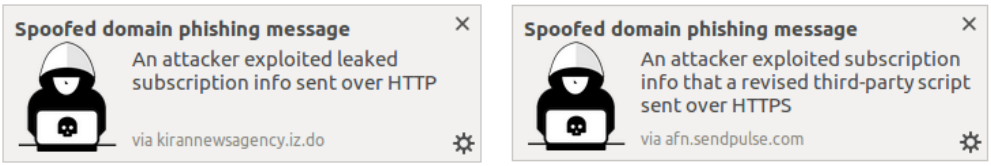
We undertook push domain spoofing attacks that leverage the two leakage paths described in Chapter 3.6.2. We assumed the presence of an active network attacker, capable of altering scripts sent over HTTP.

Subscription Object Transmission over HTTP. Among the six third-party push libraries with no VAPID protocol, Izooto is the only library that sends a subscription object over HTTP. Listing 3.2 shows in-plain text delivered over HTTP with all *endpointURL*, *p256dh* and *auth* information. Any network attacker with such subscription information is capable of sending a phishing message to the victim corresponding to the leaked *endpointURL* with a spoofed domain name.

```
http://events.izooto.com/api.php?s=0&...&bKey=ehWb8IzgwUo:APA91bGoUSAve14Oc...&auth=
  GkyeYQIFEnLnLg...&pk=BPoN_JEpU-oYXmbGle_Q-EoEB...
```

Listing 3.2: A subscription object instance sent over HTTP

[2]https://a756c-03273.chrome.pushwoosh.com is an HTTPS domain assigned to http://www.koimoi.com.



(a) A push message with spoofed domain "kirannewsagency.iz.do"

(b) A push message with spoofed domain "afn.sendpulse.com"

Figure 3.12: Demonstrations of push permission delegation and domain name spoofing

We conducted an experiment with the http://kirannewsagency.com/ PWA website, where the vulnerable Izooto [64] library was used. When an author grants the push permission for http://kirannewsagency.com/, another author exfiltrates a subscription object by inspecting deployed *mitm* proxy logs. We use this subscription information to send a phishing push message to the author who grants the push permission. Remember that the attacker is able to control all visible components in a push notification including its title, message, push icon image, and even the landing URL that redirects a recipient when clicking the push notification. Figure 3.12(a) shows our crafted push notification with the spoofed domain of https://kirannewsagency.iz.do.

Reflected Transmission of Subscription Objects over HTTPS. We found that the JS libraries fetched over HTTP from SendPulse, PushWoosh and Izooto contained a variable that holds the destination HTTPS URL (see the Listing 3.1 in Chapter 3.6.2). We changed this value to an attacker-controlled HTTPS domain. Note that SendPulse and PushWoosh have been sending subscription objects over HTTPS. However, their JS libraries enabling push services have been delivered over HTTP.

We conducted an experimented of a push domain spoofing attack against http://afn.az with the SendPulse [116] push service. The successful attack on http://afn.az changed a variable that holds https://pushdata.sendpulse.com:4434/ to have our HTTPS domain. This change causes a victim to hand over their subscription objects via a POST HTTPS request to our server. Listing 3.3 shows a retrieved subscription object, delivered to our HTTPS server. We used this subscription object to send a phishing push message with the spoofed domain. Figure 3.12(b) shows our push notification with the spoofed domain of https://afn.sendpulse.com.

```
{ action: 'subscription',
  subscriptionId: 'f4_4mOef9gY:APA91bHeQyjOvtsV ... ',
  appkey: '5b0b85c4dd9d4ded16c73d9436fa494e',
  browser: { name: 'Chrome', version: '65' },
  lang: 'en',
  url: 'http://afn.az/',
  sPubKey: 'BOMfTTU/13bEPy1FXf ...',
  sAuthKey: '8moW+qAXsAKjsOBR3F ...',
  sPushHostHash: '7c977009d5861eebb711656eb7d87a74' }
```

Listing 3.3: A subscription object delivered due to the spoofed destination URL in a target library

Table 3.4 summarizes the feasibility of our attacks against eight third-party push HTTP services. The four libraries fetched their script over HTTP, which makes the websites with these libraries are vulnerable to push permission delegation attack. Also, little or no effort has been committed to protecting the secrecy of a subscription object (which is the Izooto case). Even transmitting a subscription object over HTTPS is not enough to protect users from phishing via push messages with spoofed domains as shown in Figure 3.12(b). The VAPID protocol blocks the domain spoofing attacks. However, only two

Table 3.4: A feasibility of push permission delegation and domain spoofing attacks across third-party HTTP push services

Library	# of Affected HTTP Sites	VAPID	Push Permission Delegation	Domain Name Spoofing	
				Subscription over HTTP	Subscription over HTTPS
OneSignal	528	✓	×	×	×
SendPulse	93	×	✓	×	✓
Pushcrew	31	×	×	×	×
Pushengage	19	×	×	×	×
Izooto	18	×	✓	✓	✓
Pushwoosh	4	×	✓	×	✓
Urbanairship	2	✓	×	×	×
Foxpush	1	×	✓	×	×

Table 3.5: A feasibility of a side-channel attack using the cache on PWAs in the wild

Offline Cache Attack		# of Websites (% Percentage)
Vulnerable		187 (36.5%)
Not Vulnerable	Frame Busting	10 (1.9%)
	CSP	22 (4.3%)
	Corrupted Content	20 (3.9%)
	X-Frame-Options	132 (25.7%)
	Bad Cache	142 (27.7%)
Total		513 (100%)

vendors place the VAPID protocol, which exposes visitors on 166 HTTP websites to push domain spoofing attacks. The domain spoofing attack is critical. In the perspective of a push message recipient, there is no way of knowing that the message actually comes from the attacker because the push notification shows its valid domain. We recommend several mitigation to address the push attacks in Chapter 3.10.

3.9.3 Side-channel Attack on Browsing History via Cache

We implemented a new side-channel attack in which a PWA attacker can learn the PWA browsing history of a victim. As explained in Chapter 3.7, the attack code loads a target PWA website within an iframe on the attacker-controlled page, then checks the *onload* event callback corresponding to the target iframe is called.

To check the feasibility of our attack against various browsers, we experimented the side-channel attack against the Chrome, Firefox, Safari, UC Browser, Edge, Internet Explorer, and Opera browsers. We confirmed that our side-channel attack is effective on Mozilla Firefox 59.0.2. Fortunately, unlike Firefox and Safari, all other browsers invoke their *onload* event handlers regardless of whether the loading of a target PWA is successful or not. Also, Safari browser manages cache storage separately when loading inside an iframe.

Against the 513 collected PWAs that use the cache (see Chapter 3.3), we conducted the side-channel attacks on inferring visited PWAs. As Table 3.5 shows, 187 (36.5%) PWAs were identifiable by the side-channel attack. The attack did not work for 164 PWAs (31.9%) because of their frame busting techniques

(10 PWAs), Content Security Policy [53] including the *frame-ancestors* [90] directive (22 PWAs) and X-Frame-Options header [111] (132 PWAs).

3.10 Defense

In this Chapter, we propose defenses and recommendations to mitigate the security and privacy risks of PWAs addressed earlier. We suggest practical defenses for third-party push library providers and PWA developers to act on immediately, while recommending a guideline for PWA users.

To library providers. Third-party push library providers should manage sensitive push subscription information with care, not to leak such information by any means. A simple but powerful defense against a push domain spoofing attack (see Chapter 3.6.2) is to place the VAPID protocol. The VAPID protocol prevents any unauthenticated entity from sending a push message to a browser push service.

Another defense to block leaking subscript objects via reflected channels is to prevent a network attacker from modifying the library script. HSTS [92] header can achieve this by enforcing a JS library to be delivered over HTTPS. We observed that OneSignal, Urbanairship, Pushcrew and Pushengage set up HSTS, providing a safer service than others.

We believe that the current practice of obtaining the push permission from a redirected website is unhealthy (see Chapter 3.6.1). Unless a user is aware of the explicit relation between his visited website and its redirected website, a user is compelled to grant push permission based on the redirection, and not on the explicit domain name.

Note that push notification is designed to support only HTTPS websites. Third-party push vendors have expanded their services to HTTP websites by blindly asking a user to grant push permission for a redirected website domain. This brings unfortunate consequences such that a user makes a permission granting decision based on the redirection, which is rooted at an untrustworthy source, a HTTP website. A practical solution is that a user's browser whitelists certain third-party push service domains, and only allows the permission requests from their subdomains. The Chrome browser provides the `contentsettings.notifications` property for an extension to specify whether the listed domains are allowed to show any notifications [30]. To compute such a whitelist, users can reference the reputation of websites collected via social clouding such as Web-of-Trust [100] or Google Safe Browsing [52].

To PWA developers. The practical defense against our history sniffing attack via cache (see Chapter 3.7) is to prevent being framed by cross-domain websites. Stock *et al.* demonstrated that X-Frame-Options adopted 53% of the Alexa top 500 sites in 2016 [130], which demonstrates the security awareness on the prevention of being framed. However, X-Frame-Options, CSP, frame busting techniques are known for blocking Clickjacking attack [61], not the side-channel attack on PWAs. We thus recommend PWA developers to actively place the *frame-ancestors* directive of CSP or X-Frame-Options header that prevents the websites from being framed by other PWAs.

Applying HTTPS is a powerful defense against our push attacks as shown in Chapter 3.6. Developers should fetch their third-party library scripts and send subscription objects over secure channels so that any network attacker cannot interfere with them. The recent dedications of security communities toward secure Web have been helping seamless migrations into HTTPS websites [37, 47]. We believe that applying HTTPS has become easier and cheaper on the modern Web.

To users. Users should be aware of the phishing risk incurred by push notifications. Because push domain spoofing and push permission delegation attacks are feasible as a consequence of security flaws in third-party push libraries, users should carefully check the domain appeared in a push notification

and a push permission granting dialog.

Several previous research suggested interesting ideas applicable for mitigating our attacks. As D. Florencio *et al.* [46] proposed, users may choose a trustworthy auditing service and send their push messages to this service. This auditing service aggregates phishing push messages from different users and informs users and phishing target websites on any suspicious activities. To address the crypto-jacking attack in Chapter 3.8, monitoring of fine-grained browser behaviors [142] can identify abnormal resource consumption from a specific website and its service worker.

More practically, we recommend to regularly check the browser settings to unregister unnecessary service workers who can be abused for performing arbitrary computations. Also, cleaning the cache frequently can be an effective defense to protect the side-channel attack as shown in Chapter 3.7.

3.11 Summary

We conducted the first study of analyzing the security and privacy risks of PWAs. We analyzed the phishing risk via push notification, identified security flaws in pervasive third-party push libraries, and proposed the new attacks of abusing cache and service worker. Our findings stem from the inherent PWA features that provide native-app like Web browsing experiences, which make the addressed risks unique to PWAs. We proposed our defense recommendations to enhance the safe use of PWAs in practice. We have also reported our findings to the corresponding vendors. Samsung, Firefox, and some of third-party push service providers involved in our attacks. We view the entire work in this chapter as a step towards a better understanding of emerging native-app like features on Web applications and their security and privacy aspects.

Chapter 4. WebVR Ad Fraud and Practical Confinement of Third-Party Ads

4.1 Motivation

WebVR [147] is a JavaScript programming interface that enables virtual reality (VR) presentation in user browsers. It aims to provide an integrated VR environment for different browser platforms and operating systems. WebVR works in tandem with WebGL [65] and leverages a canvas document object model (DOM) to render VR scenes; this canvas becomes a window displaying a VR world.

WebVR websites provide the unique feature of enabling immersive virtual world experiences. Internet surfers who seek diverse and content-rich experiences are attracted to WebVR websites [127, 135]. Considering that advertisers seek opportunities to expose their ads to large audiences, it is natural for them to search for a means to bring promotional content into VR worlds. StateFarm reported a 500% increase in the click-through rate over mobile ads due to their VR ad campaigns [6], demonstrating the potential of VR ads to attract audience attention. Online VR ad service providers, including OmniVirt [101] and Adverty [7], have provided a means for advertisers to expose their products or services in VR worlds.

A standard website often monetizes its content by renting out its screen estates for ads. For this, the website embeds a JavaScript (JS) library from an ad service provider (e.g., Google or Facebook), and this library leverages an iframe element to display ads and confine the execution of their JS scripts. This iframe serves as an execution container such that the hosting website cannot alter ads within the iframe. Unfortunately, in WebVR environments, there are no iframe-like primitives that isolate the execution of an ad-serving JS script; instead, it shares a portion of the displayed VR scene. This WebVR limitation stems from the usage of a canvas DOM to render VR scenes, thus providing no browser-supported method of sharing this canvas between different web origins [98].

Previous research has demonstrated the presence of abusive ad service providers who perpetrate impression or click fraud campaigns [125, 159]. When an ad service provider with ill intent abuses the absence of iframe-like primitives in WebVR environments, there is no practical method for WebVR websites to sandbox the execution of their third-party ad-serving JS scripts. Furthermore, to the best of our knowledge, there is no previous study that investigates security threats imposed by third-party WebVR ads.

Our contributions. Assuming the presence of abusive ad service providers who conduct impression or click fraud, we introduce four new attack variants that leverage unique WebVR features. We present *gaze and controller cursor-jacking* attacks. Gaze and controller cursors are new input channels from head-mounted displays (HMDs) and VR controllers, respectively. These attacks introduce fake gaze and controller cursors into VR scenes and deceive users into clicking promotional VR entities. We then introduce a *blind spot tracking* attack whereby the adversary places promotional objects, images, and videos in the opposite direction of a user's current line of sight. This attack exploits the limited visual awareness of users when they enable 360-degree immersive views. Lastly, we propose an *abuse of an auxiliary display* attack that exploits the inability of users to view the main display when they enter the immersive mode.

We conducted user studies with 82 participants to measure the efficacy of our attacks. The exper-

imental results show that the gaze and controller cursor-jacking attacks have success rates of 88.23% and 93.75%, respectively, with participants clicking at least two ad entities. The blind spot tracking and abuse of an auxiliary display attacks have success rates of 94.12% and 100%, respectively. These results demonstrate that the adversary is able to readily conduct stealthy ad fraud.

We propose a defense system, AdCube, which is designed to block the four types of attacks presented as well as traditional web threats, including cookie theft [160] and unrestricted private information retrieval [76] by untrustworthy third parties. We define two security requirements to block the presented threats: 1) the visual confinement of three-dimensional (3D) ad entities; and 2) the sandboxing of ad-serving JS scripts according to a given security policy. To address the first requirement, we propose an algorithm confining ad objects as well as 3D models to bounding boxes, called *adcube*. To address the second requirement, we leverage Caja [33], a mature sandboxing technology maintained by Google, to confine the execution of third-party JS code. Specifically, on top of Caja, we design a set of JS APIs that an ad-serving JS script is able to use to create WebVR ads and implement each API. Therefore, a benign WebVR website owner is able to use AdCube to confine the locations and executions of VR ads as the owner specifies.

We evaluated the performance of AdCube in terms of page loading time and frames-per-second (FPS). Compared to the baseline without any defense, AdCube produced a negligible FPS drop when rendering a complex demo site of a virtual art museum and an additional page loading time of 236 msec on average when rendering nine WebVR sites, thus demonstrating the promising efficacy of AdCube in the wild.

4.2 Background

4.2.1 WebVR

VR technology offers an immersive user experience that provides users with a virtual 3D world. Rendering a virtual world scene entails heavy usage of matrix computations, high demand for graphics processor unit (GPU) resources, and the frequent loading of large-sized graphic textures and images. These requirements make native applications the only viable means of delivering a VR world. However, the proper installation and frequent software updates, which native applications often require, have hindered their wide adoption.

The advent of WebVR addresses these core limitations. This new technology enables a website to offer a VR environment by means of browser supports. WebVR specifies a set of browser-supported APIs that enables VR in user browsers [147]. It provides interfaces for managing VR peripherals, such as HMDs and VR controllers, thus enabling an immersive 3D world experience. WebVR works in tandem with WebGL [65] to render VR content on an HTML5 canvas DOM element. WebGL provides a set of interfaces that launch shader programs as well as manage viewports, thus rendering sophisticated 3D entities and models via a large volume of matrix computations empowered by GPUs.

In 2018, WebVR was integrated into WebXR [148], which is designed for both augmented reality (AR) [83] and virtual reality (VR) on the Web. However, the original architecture of WebVR remains the same in WebXR, with only keyword changes. In this paper, we focus on addressing new security threats that involve WebVR APIs in WebXR.

WebVR terminology. Here we clarify WebVR terms that we use throughout the paper. A *VR scene* refers to a view of a VR world in WebVR. In this definition, a scene requires a viewer of the VR world.

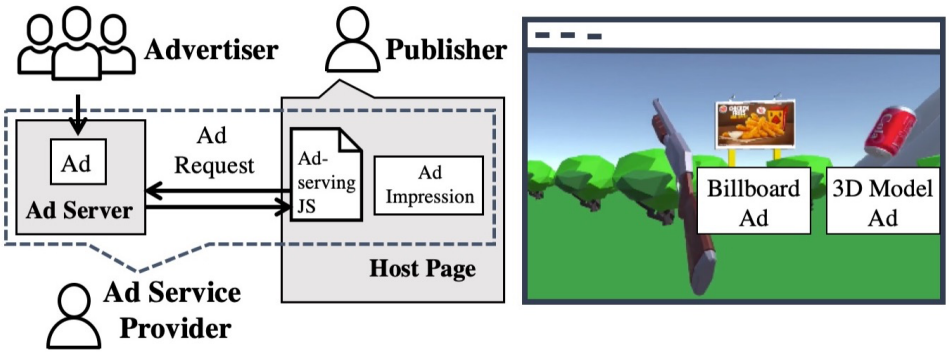


Figure 4.1: A simplified overview of the web ad ecosystem and examples of OmniVirt VR ads: A billboard ad and a promotional 3D model in a VR scene.

A *camera* refers to this viewer, usually represented by the perspective of a user. The *immersive mode* refers to the mode in which a user sees the scene through an HMD. A *viewport* defines a rectangular area where the VR world is rendered. Most WebVR sites offer two viewports onto their VR world; these viewports correspond to the left and right eyes, respectively. An *entity* refers to a visible or invisible object within a scene. To avoid confusion with DOMs and JS run-time objects, we explicitly use the term *entity* to describe objects placed in a VR world. Therefore, an entity that promotes a commercial product is called either an *ad entity* or a *promotional entity*.

3D library. To facilitate usages of WebGL, many JS 3D libraries, including Three.js [139], babylon.js, and React 360, have been proposed. Several vendors have even promoted new WebVR frameworks (e.g., A-Frame [1], PlayCanvas [103], and Sketchfab [119]), which not only provide intuitive interfaces but also establish their own abstraction layers to ease the implementation of rich VR experiences.

A-Frame [1] is a representative WebVR framework introduced by Mozilla in 2015. Its striking feature is that a scene and all entities rendered within the scene can be defined through a markup language, which is accessible via a DOM [91]. For instance, a developer can create a 3D box entity by defining an `<a-box>` tag in HTML and query this entity via JS DOM APIs. This intuitive approach to encoding diverse 3D entity properties into HTML tag attributes has lowered the technical barriers to developing VR content.

4.2.2 Online Advertising

There exist three main types of participants in the web ad ecosystem: publishers, ad service providers, and advertisers. Figure 4.1 depicts how these three participants interact with one another. Publishers are website owners or operators who serve informative, promotional, or intriguing content to their website visitors. Advertisers play a role in planning and bidding on ad campaigns and want to expose those ad campaigns to users who visit publisher websites. Ad service providers connect these publishers and advertisers; they provide publishers with ad-serving JS APIs, which fetch banner, text, and even video ads provided by advertisers.

Web ads have been a prevailing method by which publishers monetize their content. As advertisers seek diverse channels and responsive interactions with their audiences, ad technology has evolved to support not only text banner ads but also various multimedia delivery mechanisms, such as video and native responsive ads integrated into their hosting websites [122, 159]. For instance, news feed ads blended

with other non-ad feeds have become a popular ad technology for social media platforms, including Facebook [19, 159].

VR ad market. The VR market was valued at USD 7.3 billion in 2018 and is expected to reach 20.5 billion by 2026 [154]. The total number of active VR users was approximately 171 million as of 2018 [126]. Thus, it is natural for advertisers to seek new opportunities to promote their products or services in a content-rich VR environment, thereby reaching a large number of VR users.

There exist at least 13 VR ad service providers offering VR ad forms; OmniVirt [101] and Wonderleap [155] have supported options to initiate WebVR ad campaigns. OmniVirt reported 100 million and 1 billion delivered VR/AR ad impressions in 2017 and 2018, respectively [143], which demonstrates the surging demand for VR ads.

Ad fraud. Ad fraud refers to an operation that generates unintended ad traffic involving ad impressions or clicks. Previous studies have described various adversarial models that address ill-intentioned publishers committing click fraud [27, 61], malicious extension replacing ads [136], and abusive ad providers generating unwanted ad traffic [125, 159].

In this paper, we assume an abusive ad provider whose objective is to increase ad traffic that fetches ad impressions or generates click events via deceptive techniques. To the best of our knowledge, there have been no previous fraud studies involving WebVR ads.

4.3 Problem

A typical way of placing web ads is for publishers to copy and paste an ad bootstrapping JS script on their websites. This embedded JS code, which runs with the same origin as its host webpage, creates an iframe [95] of the page that is fetched from a third-party ad service provider. Because the publisher origin differs from the origin of the embedded iframe, the JS script in this iframe can neither alter nor read resources from the hosting page due to browsers enforcing the Same Origin Policy (SOP) [98]. The SOP ensures that the rest of the ad script confined within this iframe is isolated from the hosting page. Thus, publishers only need to check how the embedded bootstrapping JS code performs to prevent potential abuses by advertisers or ad service providers.

Problem. Today's WebVR does not provide an origin separation mechanism that allows a third-party ad script to securely share the same origin as its hosting page to render ad entities, images, or videos within the VR content of the hosting page. This limitation stems from the usage of a canvas element [93] when rendering VR content, which does not provide a way of sharing this element among different origins. This limitation leaves no option for WebVR ad service providers except to run their ad scripts with the origins of hosting pages. Consequently, it is imperative that publishers completely trust these ad service providers.

Unfortunately, previous studies have demonstrated the presence of abusive or malicious ad service providers that victimize visitors to publisher websites [61, 125, 136]. For instance, Springborn *et al.* [125] investigated abusive pay-per-view networks that expose fraudulent impressions via pop-under or invisible ads to increase the number of served ad impressions. Given that an abusive ad service provider is capable of running scripts using its hosting origin, she is able to conduct clickjacking [10, 61], steal cookies [160], and even access the private information of users [76]. However, no previous study has addressed the unique risks entailed in WebVR. Considering that WebVR introduced an immersive mode, in what ways does this paradigm shift favor the attacker?

Sandboxing. Previous studies have investigated how to sandbox the execution of third-party ad scripts

within the same origin as the hosting page [3, 8, 43, 63, 82, 87, 114, 141]. Such sandboxing methods are viable as they require low overhead, which is a key requirement in WebVR environments demanding a robust FPS rate. However, it is not clear how to apply these existing techniques to confine WebVR ad scripts.

What are the security properties that the sandbox technique should guarantee? Which API should the sandbox technique provide to support VR features while achieving security requirements? These questions drive our research into providing a practical method of confining ad scripts in WebVR websites.

4.4 Threat Model

We assume an *abusive ad service provider* who serves 2D/3D ads into WebVR sites. In this scenario, the business imperative is to expose promotional VR entities, images, or videos in the VR worlds of publishers. At the same time, the goal of the adversary is to increase ad traffic by rendering more promotional entities and to generate user clicks via deceptive techniques that increase ad revenue. We emphasize that this adversary model is a real threat. There exist numerous malicious secondary or tertiary ad service providers whose sole motive is to maximize their short-term profit [67, 85]. `Adf.ly` was a notorious abusive ad service provider that modified the link addresses of publisher pages and tricked users into clicking ads [159].

Considering that there exists no practical way of separating origins that share the same canvas that renders VR scenes, we assume that the adversary places her ad-rendering code at the hosting page, which allows the code to access any resources that belong to the hosting page. The adversary victimizes publishers by abusing their website visitors; her ad-serving JS script generates ad fraud traffic by victimizing visitors. These publishers also lose visitors due to providing bad user experiences with fraudulent ads. An advertiser also becomes a major victim who is obliged to pay for those fraudulent ad impressions and clicks.

4.5 Attacks

In this Chapter, we present four new ad fraud attacks that exploit blind spots and new VR peripherals.

4.5.1 Cursor-Jacking Attack

Facilitating WebVR experiences requires two representative IO devices: an HMD and a VR controller. These devices introduce two new input channels: a gaze cursor and a controller cursor, which did not exist in a standard web environment.

Unfortunately, both of these input methods can be altered by a JS script, allowing a malicious ad service provider to control them. Thus, the adversary abuses this capability by creating a fake input source to induce actual clicks on other entities. Specifically, we introduce two attack vectors: gaze and controller cursor-jacking attacks.

Gaze cursor-jacking attack. A gaze cursor is a marker that represents the focal point at which a user looks in a VR scene. Usually, a gaze cursor has a circular appearance, which helps users realize what they are looking at. This gaze cursor supports a *fusing* event that fires when a user locates the cursor on a targeted entity. When the gaze cursor stays on this target entity for 1.5 seconds (default), a browser

(a) Gaze cursor-jacking (b) Controller cursor-jacking

Figure 4.2: An illustration of (a) gaze and (b) controller cursor-jacking attacks: (a) When a user clicks a UI button via the gaze cursor made by the attacker, the authentic cursor clicks the ad. (b) Inserting a fake controller cursor by rotating its z-axis by 180 degrees. When a user clicks the green box with an authentic VR controller cursor, the ad placed in the opposite direction is also clicked.

then fires a *click* event. Thus, the gaze cursor provides a unique way of triggering a "click" event on an entity without involving any mouse or controller events.

Gaze cursor-jacking (GCJ) refers to an attack that creates a fake gaze cursor and hides the original cursor in a target VR scene. This GCJ attack leads its victims to believe that a fake cursor is actually an authentic input cursor and to place the "authentic" cursor at a point where the attacker wants it to be. Figure 4.2(a) demonstrates the implementation of the attack in an A-Frame environment. The attacker is able to make the authentic gaze cursor invisible and insert a fake gaze cursor that triggers click events on different entities placed near the position where the authentic cursor is located. Thus, she is able to hijack authentic clicks that should be attributed to first-party content.

Controller cursor-jacking attack. A VR controller is another input device that enables a user to trigger various events on entities, such as clicks. Usually, a VR site shows a projection line that points to a target, which varies according to the user's controller direction. A user leverages this projection line in a scene to select a target entity at which the user fires various events.

Controller cursor-jacking (CCJ) is an attack that introduces an additional fake VR controller cursor in a target VR scene. When a victim generates a user event on an entity, the same event is also triggered at the target entity that this fake cursor indicates because this fake cursor shares user events with the original controller cursor. The adversary is able to leverage blind spots to hide fake controller cursors and induce clicks on ads whenever a click occurs (Figure 4.2(b)).

In a standard web page, a clickjacking attack [61] performs a similar attack by using another iframe window from a third-party source that actually tricks victims into clicking a target element underneath this iframe window. However, the two attacks presented here differ from the clickjacking attack in that they do not exploit third-party windows due to the WebVR nature of sharing the same scene between first- and third-party scripts. Furthermore, these two attacks abuse new input vectors that only exist in a WebVR environment.

Considering the adversary is already able to fire click events via `dispatchEvent` API invocations, she might not need to induce genuine user clicks with these two attacks to achieve her goal. However, in Chrome, Edge, and Oculus Android browsers, only event handlers invoked via genuine click events are able to open a new window or cause redirection to a different website. Because the goal of the adversary is eventually to redirect victims to an ad-landing page, the attacker has a clear motive to conduct GCJ

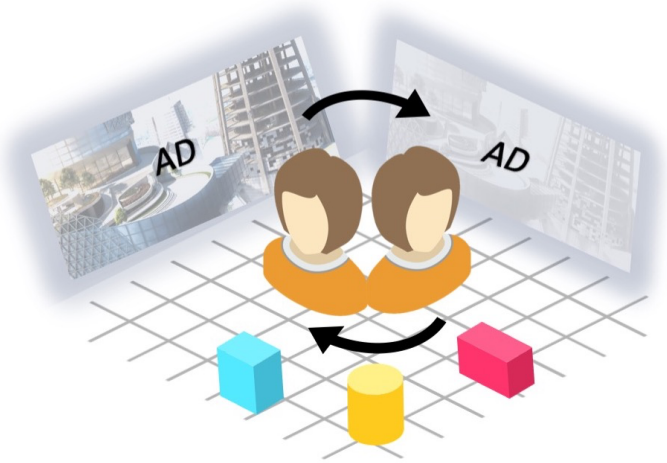


Figure 4.3: An illustration of blind spot tracking ads.

and CCJ attacks.

4.5.2 Blind Spot Attack

A WebVR site offers surrounding 360-degree views through the support of an HMD, thereby enabling a new kind of browsing experience. This results in two types of blind spots that users are unable to see when experiencing the immersive mode with the HMD: 1) one is located in the direction opposite a user's current line of sight, and 2) the other is the main display, such as a desktop monitor or a laptop display, which becomes an auxiliary display when users wear the HMD. The attack is able to place promotional entities in these blind spots, which are invisible to users. We introduce two attack vectors: blind-spot tracking and abuse of an auxiliary display.

Blind spot tracking attack. A blind spot tracking (BST) attack occurs when the adversary hides an ad entity in the opposite direction from the user's current line of sight. She can also move this entity into blind spots whenever the user's gaze changes direction by tracking the camera sight's direction (Figure 4.3). Thus, the adversary is able to increase the number of rendered ad impressions or entities, later charging the respective advertisers for this inflated number of ad views.

The BST attack is a unique variant of ad impression fraud. Ad impression fraud refers to an operation that (1) hides rendered ads underneath other UI elements, (2) makes ads invisible by making them too small, (3) places ads to appear when a user scrolls down a webpage, or (4) simply renders a vast volume of ad impressions [72, 73, 132]. On the other hand, the proposed BST attack leverages blind spots that are inherent in any VR content.

Abuse of an auxiliary display attack. An attacker can abuse the user's limited awareness of the browser on the auxiliary display by displaying diverse ad impressions or videos to maximize ad view counts. We call this attack an abuse of an auxiliary display (AAD) attack.

Furthermore, the attacker can identify the moment when a victim exits the immersive mode when a `vrdisplaypresentchange` event is fired or when the HMD device is taken off; this is achieved by monitoring abrupt gaze cursor changes or scene change events. When identifying such moments, the attacker can remove all ad impressions and stop video ads on the auxiliary display involved in stealth ad campaigns.

4.6 User Study

To measure the efficacy of the presented attacks (Chapter 4.5), we recruited 82 participants and investigated their responses to the four attack scenarios. This Chapter describes our user study designs (Chapter 4.6.1) and experimental results (Chapter 4.6.2).

4.6.1 Experimental Design

From July to October 2019, we recruited a total of 82 university students, consisting of 52 males and 30 females (mean age = 23.69). Among them, 49 had been exposed to VR experiences before. The participants were offered $5 per attack scenario, each of which took approximately 30 minutes to complete. We obtained IRB approvals and consent from every participant. We focused on demonstrating the feasibility of each attack rather than proving its success on general audiences. For participants, we thus targeted primary consumers of VR content, whose ages were between 19 and 30 [128].

Each participant was randomly assigned to one of four attack scenarios. For each attack scenario, we prepared two webpages: one represented the normal case without any attacks (control group); the other was implemented for the corresponding attack. We used a within-subject design; all participants experienced both normal and attack tests in each scenario. To minimize the learning effect, whereby a prior user study experience affects metrics observed during a posterior user study, we shuffled the order of normal and attack cases for each participant, ensuring that the same number of users initially experienced normal and attack cases.

While exploring the two webpages described above, the participants were asked to complete a specific task for each page. At the end of each task, they were asked to complete a survey asking about their awareness of the existence of rendered ads and the differences between the normal webpages and those under attack. For the GCJ and CCJ attacks, we asked the participants of the user study the following questions:

1. *Did you find any promotional products or brands in your VR experience?*
2. *If yes, mark each of the findings in the given examples. (We gave examples of ad objects to choose from.)*
3. *Is there a webpage where you were exposed to more ads between two VR webpages that you explored? If so, why?*

Using the examples provided in Question 2, we could verify whether the ad object that the user claimed to have found was a genuine ad object. For the BST and AAD attacks, we asked the participants the following questions:

1. *Did you hear any sounds of commercial clips? If yes, which sounds?*
2. *Did you find any ad videos in the background of your VR scene or on your monitor screen? If yes, which videos?*
3. *Which of the two VR websites exposes ads? And which ad is exposed?*

In Question 3, "two VR websites" refers to the normal and attack websites for each attack scenario.

We reserved a spacious classroom for the participants to browse the VR websites and prepared a Windows 10 host with an HTC VIVE device. We instructed the participants not to interact with ads

Table 4.1: Experimental results for participants who experienced GCJ and CCJ attacks.

*Half indicates at least three out of seven ad entities for GCJ and at least two out of three ad entities for CCJ.
†In the normal case, the attack is not carried out, so the result is shown as N/A.

Attack Scenario	Treatment Group	Total	Awareness of Ads			Authentic Clicks			Forged Clicks (Attack Success)		
			at least one	half*	all	at least one	half*	all	at least one	half*	all
GCJ	Normal	17	17	17	5	11	4	0	N/A†	N/A†	N/A†
	Attack	17	17	16	11	6	5	1	17 (100%)	15 (88.23%)	0 (0%)
CCJ	Normal	16	6	2	1	0	0	0	N/A†	N/A†	N/A†
	Attack	16	6	2	2	0	0	0	16 (100%)	15 (93.75%)	6 (37.5%)

and notified them that any clicks on promotional entities would be considered interactions. We gave explicit guidance to the participants that they did not need to interact with any ad entities to finish a given task.

4.6.2 Experimental Results

Gaze Cursor-Jacking Attack

We used halloVReen [49], a game of finding hidden animation objects, to test the efficacy of a GCJ attack. The participants were expected to find five animated Halloween ghosts scattered across a VR scene via gaze-clicks. The task was to end after five minutes, regardless of the completion of a given task.

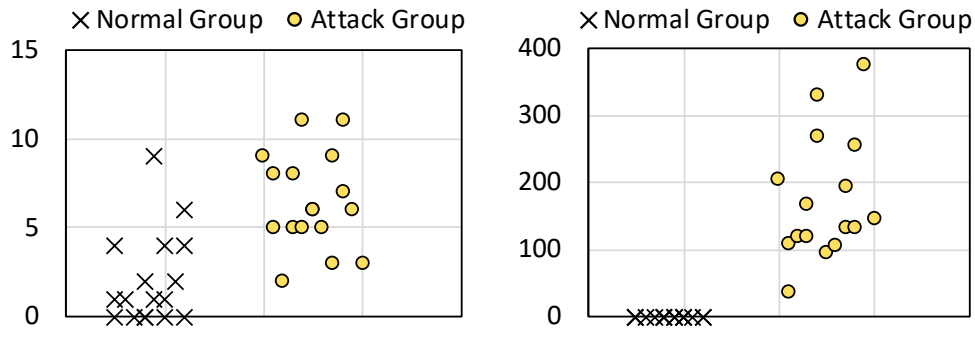
For the normal webpage without the attack, we placed seven ad entities placed near Halloween figures. For the attack page with the GCJ attack, we placed seven different ad entities near Halloween figures. We also created a fake gaze cursor near the actual cursor and made the actual cursor invisible. To minimize the learning effect, we used different Halloween figures and ad entities for each webpage.

When participants gaze-clicked the fake cursor on the Halloween figures, the ad entities were gaze-clicked by the actual cursor. Because a gaze click event is triggered when the cursor stays on a target for at least 1.5 seconds, non-intentional head movements could not have accounted for any of the gaze-clicks. In other words, all counted gaze clicks originate from either users' intentional clicks or the GCJ attack. After a given task, participants were asked on the survey to check which ad objects they had found and whether they had clicked any of them.

Table 4.1 shows the experimental results. The columns below Awareness of Ads represent the number of participants who noticed the existence of ads. The sub-columns at least one, half, and all represent the number of participants who recognized at least one, half, and all of the promotional entities, respectively. The columns below Authentic Clicks show the number of participants who intentionally gaze-clicked promotional entities. Also, the Forged Clicks columns represent the number of participants who gaze-clicked promotional entities with the real gaze cursor due to this attack.

As the first row in Table 4.1 shows, all 17 participants who browsed the normal webpage discovered at least three ad entities, which is about half of the seven ad entities that we placed in the scene. Interestingly, whereas the instructions were given to avoid clicking on promotional entities, 11 and 6 people in the normal and attack cases, respectively, intentionally clicked on at least one ad entity.

As the second row in the table shows, every participant gaze-clicked at least one ad entity due to the GCJ attack, which is a significant improvement over the six participants who intentionally gaze-clicked



(a) GCJ (b) CCJ

Figure 4.4: Total number of participants' clicks on all ads in the GCJ and CCJ attack scenarios.

at least one ad entity in the attack case. Also, 15 participants (88.23%) gaze-clicked at least three ad entities due to the attack; this means that the emplaced attack caused a majority of the participants to gaze-click ad entities.

Figure 4.4(a) shows the total number of gaze-clicks on all ad entities for each participant. The normal group represents the number of intentional clicks on ads in the normal case, while the attack group represents how many gaze-clicks were fired due to the attack. The mean of the clicks due to the attack is 6.51, which is three times greater than 2.61, which is the mean of clicks in the normal case. This statistic demonstrates that the adversary can generate more gaze-clicks on ads than in the normal case by exploiting a GCJ attack.

Controller Cursor-Jacking Attack

For the second user study, we used A-Blast [146], a shooting game in which players shoot flying monsters with two blaster guns maneuvered by two VR controllers. Monsters appear randomly within 120 degrees of the front. Each participant played a game for five minutes, or the game ended early when his/her avatar died.

We prepared two webpages. The normal page implemented the A-Blast game, in which three ad entities were placed in positions where the participants would hardly ever look, which was about 180 degrees away from the front. The attack page had the same ads placed in the same location as on the first page. It implemented the CCJ attack, in which a fake controller was inserted that rotated the z-axis 180 degrees. Thus, the participants unwittingly clicked the back of the scene when they shot at monsters in front of their sights, thus clicking ad entities.

The second row of Attack Scenario in Table 4.1 summarizes the experimental results. It shows that six of 16 participants were aware of at least one ad entity in both cases although they could not see them unless they turned their line of sight around 180 degrees. Also, no participants intentionally clicked ad entities because they were located in the opposite direction of the front area where the game was taking place.

Due to the attack, every participant (100%) clicked at least one ad entity. Also, 15 (93.75%) and 6 (37.5%) participants clicked at least two and all of the ad entities, respectively. This demonstrates that no one intentionally clicked these ad entities but that the attack caused participants to click them.

Figure 4.4(b) shows the total number of clicks on all ad entities for each participant. These results

show that there were no clicks in the normal case. On the other hand, the mean of ad clicks in the attack case was 174.31. This unbalanced metric indicates the effectiveness of the CCJ attack.

Blind Spot Tracking Attack

We revised two VR game websites to implement a blind spot tracking attack: halloVReen [49] and Whack-a-mole [109]. The Whack-a-mole game is designed such that users attempt to grab moles, which appear in the 360-degree scene, via user gaze-click. We asked the participants to finish the games in one minute.

The attack was implemented by placing a video ad for which z-order was set to the behind camera position, thus rendering the video ad at a blind spot of the participant. Considering that a typical video ad is accompanied by music and sound effects, we also tested the degree to which ad sounds enhanced the participants' awareness of ads in their blind spots. The participants were asked to wear earphones connected to the HMD supporting 3D spatialized sounds. Note that the 3D spatialized sounds only reflect the distance from a sound source and not the direction. Regardless of whether sounds were played in the front of or behind the participants in our VR worlds in A-Frame, the participants heard the identical sounds.

We chose two ad videos that advertise a popular supermarket and drink product. They had been well-received by university students due to a heavy volume of commercial marketing. Thus, the participants were highly likely to recognize these brands by just hearing the sounds of these ad videos.

For the user study, we prepared two treatment groups. One group consisted of 17 participants who experienced two VR websites: Whack-a-mole for the normal case and halloVReen for the attack case with the sound enabled. The other group, consisting of 15 participants, experienced two VR websites: halloVReen for the normal case and Whack-a-mole for the attack case with the sound muted.

Table 4.2 presents the experimental results for each treatment group. Of the 32 participants who experienced the normal sites that rendered no ads, only three participants (9.375%) claimed that they heard ad sounds, which were actually the sound effects of the underlying websites. Of the 15 participants who experienced the attack site with the sound muted, only two (13.3%) claimed awareness of ad sounds, which were actually noises in the experimental environment, such as desk-dragging sounds. That is, no one heard genuine ad sounds in the normal and attack cases with the sound muted. In contrast, 14 participants (82.35%) who experienced the attack site with ad sounds claimed that they indeed heard ad sounds and became aware of the presence of ongoing ad campaigns.

Note that no one in the attack group saw the ad video in the muted attack, and only one participant claimed that he saw an ad video in the sound attack. Considering that this participant could not specify the ad video he saw, we concluded that he did not see any video ad playing in the opposite direction of his line of sight. We concluded that the BST attack is capable of concealing ad impressions and videos, rendering users unable to recognize whether ads are rendered.

Abuse of an Auxiliary Display Attack

We implemented an AAD attack on the A-Blast website [146]. Each participant played a game for five minutes. The attack created an iframe that rendered a video ad on the A-Blast webpage in the original desktop display when a participant entered the immersive mode. The attack also deleted this iframe when a participant exited the immersive mode. Therefore, it was improbable for participants to find such ads unless they took off the HMD device before finishing the task.

Table 4.2: Experimental results for participants who experienced the blind spot tracking (BST) and the abuse of an auxiliary display (AAD) attacks.

Note: The *Awareness of the Ads* column indicates that the number of participants who realized the presence of ads. The *Found ads* column shows the number of participants who actually saw the ads.

Attack Scenario	Treatment Group	Total	Awareness of the Ads	Found ads (Attack Success)
BST	normal	32	3	0 (N/A)
	attack (muted)	15	2	0 (100%)
	attack (w/ sound)	17	14	1 (94.12%)
AAD	normal	32	3	0 (N/A)
	attack (muted)	17	1	0 (100%)
	attack (w/ sound)	15	15	0 (100%)

We also measured the effects of ad sounds to measure the participants' awareness of the ads rendered on their auxiliary display, which was the desktop monitor used in this user study. For the user study, we designed two treatment groups. One group consisted of 17 participants, and they experienced the A-Blast website for the normal case and the same website for the attack case with the sound muted. The other group, which consisted of 15 participants, visited the same A-Blast website for the normal and attack cases with enabled sound. For the ad videos rendered, we chose two videos that advertise a popular e-commerce site and a vitamin drink product.

Table 4.2 presents the experimental results for each treatment group. Only three (9.375%) of the 32 participants who experienced the normal site and one (5.882%) of the 17 participants who experienced the muted attack site claimed hearing ad sounds; however, they were the sound effects of the underlying websites. On the other hand, all (100%) of the 15 participants who experienced the attack site with sound were aware of the presence of ongoing ads due to the video ad sounds. However, note that no one explicitly found the ad video, thus demonstrating the feasibility of abusing this attack in a stealthy manner.

4.7 AdCube

This Chapter explains two security requirements to mitigate the presented attacks and a defense model of AdCube (Chapter 4.7.1). We then present the architecture of AdCube (Chapter 4.7.2) and its usage in terms of defining security policies (Chapter 4.7.3). Lastly, we explain how AdCube is implemented to enforce the aforementioned security requirements (Chapter 4.7.4 and Chapter 4.7.5).

4.7.1 Defense Model

We list security requirements that a new defense model should have in order to prevent the four proposed attacks as well as traditional threats [74, 76, 160].

1. Third-party JS code should place ad entities only within the confined area that the first party specifies, and these entities should fit within this area.

2. Third-party JS code should not be able to alter DOM elements and sensitive entities (e.g., camera and controller) if the first party does not permit doing so.

The first requirement aims to block the BST attack and any abusive attempts to place a prohibitive number of ad entities all over the VR scene of a publisher. The second condition is required to block the GCJ, CCJ, and AAD attacks, thereby limiting the adversary's capability of changing gaze cursors, VR controller cursors, and DOM elements belonging to the first party. Note that the defense system in the second requirement also prevents malicious third-party scripts from gaining unrestricted access to sensitive information, such as credential cookies and private information [76, 160].

Previous studies have addressed the second requirement by confining the execution of third-party code [8, 63, 74, 104, 114, 117]. These approaches are categorized according to two objectives: 1) origin-based isolation and 2) code sandboxing. The origin-based isolation refers to a technique that assigns each embedded third-party code with a separate origin (or process) so that SOP (or process isolation by OS) forces the confinement of the third-party code. In contrast, code sandboxing enforces third-party code to interact with its host via specified APIs while sharing the same origin with its host.

Unfortunately, origin-based isolation techniques, including AdJail [74] and AdSplit [117], often demand a heavy volume of cross-origin or process communications, which enable the separate origins of third-party codes to operate as a single app. Such a large volume of communications introduces execution latency, thus impeding a stable frame rate, which undermines rich user WebVR experiences. On the other hand, previous studies of code sandboxing have not explored the confinement of a third-party script in a WebVR environment [8, 104, 114].

To this end, we propose *AdCube*, a client-side defense solution that addresses the aforementioned two security requirements. The defense is designed for benign publishers who wish to prevent third-party scripts from accessing and modifying the host page's DOM elements and VR entities. For the first requirement, AdCube provides a hexahedron, called an adcube, which visually confines the ads. The publishers specify its position to indicate where an ad entity should be rendered. To address the second requirement, AdCube sandboxes a given third-party JS script while providing a limited set of APIs which the third-party codes use to render WebVR ads. Also, it allows the publishers to set a security policy, which defines how specified third-party scripts should interact with host elements. Therefore, the ad service providers should implement their ad-serving scripts in AdCube APIs. To enable AdCube, the publisher embeds an AdCube JS library in their host script.

Publisher's motives. Considering that ad fraud campaigns may not only benefit the adversary but also publishers in the adversary's ad network via inflated numbers of impressions and clicks, the following question arises: *What would motivate publishers to use AdCube?*

Note that an abusive ad service provider may harness the absence of visual confinement of WebVR ads. The adversary emplaces an enormous number of ad entities that visually block the VR content of a publisher, thereby diverting visitors' attention to the invasive ads [60], which conflicts with the publisher's intention. Furthermore, this service can also place eye-grabbing promotional entities that block first-party promotional entities, conducting occlusion attacks [68]. These invasive or spammy ads can eventually contribute to visitors avoiding publisher websites [15].

The FTC states that publishers are responsible for substantiating whether deceptive ads are present [25]. They examine whether publishers have known or participated in serving deceptive and invasive ads. Google penalizes the search rankings of publishers with invasive ads [17]. We believe that these trends necessitate the adoption of AdCube by publishers.

Furthermore, it is known that security vulnerabilities, including cross-site scripting bugs, often arise from third-party JS code [87, 114, 124, 160]. AdCube is able to isolate third-party JS code, thereby preventing the adversary from harming the customer via exploiting security vulnerabilities.

4.7.2 Architecture

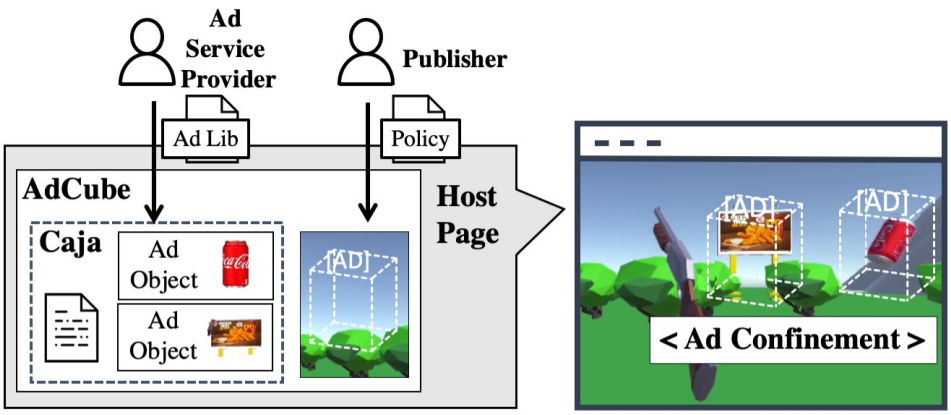


Figure 4.5: AdCube overview.

The overall architecture of AdCube is demonstrated in Figure 4.5. AdCube is a JavaScript library, designed to confine the execution of third-party scripts rendering WebVR ads. A publisher furnishes this JS library with a given security policy that specifies how a third-party script should interact with the resources belonging to the first-party origin. The publisher then embeds a third-party JS ad library in Secure ECMAScript [36], which fetches and renders WebVR ads.

To sandbox this embedded JS ad library, AdCube leverages Caja [33], a seminal sandbox framework from Google. We chose Caja from among the previous studies [3, 8, 43, 63, 82, 87, 114, 141] because it is open source software that has been well-managed for over 10 years. Caja sandboxes the execution of guest code from its host page so that the guest code is only accessible to defensive objects that the host page allows. Caja achieves this sandboxing via dynamically monitoring the execution of transformed guest code. Caja conducts *cajoling* of the original guest code into a transformed version, which adds inline checks to enforce invariants that Caja requires.

By design, Caja guarantees no free variables. Thus, the only way for guest code to modify JS objects or DOMs in the host page is to use the references of defensive objects that the host page explicitly offers. Furthermore, the host page is able to enforce customized access control checks on these defensive objects because the host page can revise the APIs that the guest code uses to access defensive objects.

Therefore, the sandboxing of a third-party ad JS script is enforced by Caja. AdCube is a set of wrapper Caja APIs. For publishers, AdCube offers a security policy language. For ad service providers, AdCube offers a set of JS APIs that enable the programming of WebVR ads while interacting with VR entities in host pages.

Listing 4.1 shows an example of applying AdCube to an A-Frame host page. To enable AdCube, a publisher includes *adcube.js* at Line (Ln) 2. Also, the publisher defines an advertising cube at Ln 9 where a third-party ad-serving script places VR advertising entities. The publisher is also able to specify a security policy that defines which host elements a third-party ad script interacts with. As Lns 5-6 indicate, the third-party is able to *read* the properties of the a-box DOM object and to *write* the properties of the a-sphere object. At last, the ad script embedded at Ln 13 runs in a Caja-enforced sandbox with limited access to the a-box and a-sphere entities. That is, this *load* invocation specifies third-party scripts that should be sandboxed via AdCube.

```
<body>
<script src="adcube.js"></script>
<a-scene>
<!-- part of the host app -->
<a-box can-read></a-box>
<a-sphere can-write></a-sphere>
...
<!-- a new definition for ad -->
<a-adcube position="0 0 -2" width="2" height="2" depth="2"></a-adcube>
</a-scene>
<script>
const adcube = AdCube();
adcube.load("https://3rdparty.com/ad.js");
</script>
</body>
```

Listing 4.1: An example of A-Frame host page with AdCube

4.7.3 AdCube and Security Policy

AdCube asks a publisher to specify two types of specifications: 1) an `adcube` primitive that specifies a third-party ad rendering space in the VR world of a host page; and 2) a security policy that specifies DOMs that interact with a confined third-party ad script.

AdCube primitive. An `<a-adcube>` tag defines an AdCube primitive for A-Frame enabled web pages. It specifies a hexahedron in which to render WebVR ads. This adcube tag has four properties. The `position` property specifies a hexahedron position in the VR world of a host page. The `width`, `height`, and `depth` define the size of this hexahedron. When this `<a-adcube>` tag is placed as a child of a host element, AdCube internally sets the `parent` of the `adcube` to be this host element, and the location of the `adcube` is relative to this parent element. For instance, when specifying the parent element of an `adcube` primitive to be a camera entity, this `adcube` moves as the camera angle of the scene changes.

Security policy. A publisher with AdCube is able to specify access control policies regarding which host entities and DOMs are *readable* or *writable* by a third-party script that AdCube sandboxes. Specifically, the publisher assigns a `can-read` or `can-write` attribute to an A-Frame entity or a DOM. AdCube stores this labeled entity or DOM in a JS object, called `TamedDOM`, which AdCube lets a third-party script access or revise via the `querySelector` API. That is, `TamedDOM` becomes a bridge between the host and a sandboxed third-party script. AdCube implements this functionality by leveraging the `markfunction` API of Caja.

By default, AdCube prohibits a sandboxed third-party script from accessing any entities or DOMs in the host page. This default policy blocks all the attacks (Chapter 4.5) by preventing a third-party script from accessing cameras, gaze/controller cursors, and DOMs whose origin is bounded by the host origin. Moreover, this default policy significantly lightens the burden of specifying a proper security policy for publishers.

Furthermore, AdCube attaches an "[AD]" label at the top of a defined `adcube` area, as shown in Figure 4.6, thus making VR ad content visually distinguishable from host VR entities. In this way, publishers are able to help their visitors easily identify which entities are for ads, which the IAB has been recommending for healthy ad ecosystems [62].

Table 4.3: An API list for advertising.

Creation
`createElement(`*[tag name|URL]*`)`
Creates a new entity and returns the *entity's interfaces* defined by AdCube
`addElement(`*adcube_id, entity*`)`
Appends an *entity* to the adcube which has the *adcube_id*.
Set
`entity.setAttribute(`*key, value*`)`
Sets an `entity`'s attribute with *key* and *value*
`entity.appendChild(`*child entity*`)`
Appends a *child entity* to the `entity` as its children
`entity.addEventListener(`*event name, function*`)`
Sets an `entity`'s event handler with *event name* and *function*
Get
`entity.getAttribute(`*key*`)`
Returns an `entity`'s attribute corresponding to the *key*
`querySelector(`*tag name|ID*`)`
Returns an *entity* corresponding to the *tag name* or *ID*
`querySelectorAll(`*tag name|ID*`)`
Returns *multiple entities* corresponding to the *tag name* or *ID*

4.7.4 Ad Service APIs

AdCube sandboxes a third-party script by providing a confined execution environment with predefined objects and APIs. We designed a set of APIs that a third-party ad serving script is able to use to implement VR content. Instead of defining a long list of all possible APIs, we focused on defining essential APIs for the AdCube prototype. Table 4.3 shows the API list. We designed our APIs similar to JavaScript DOM APIs [91] to make them compatible with common software engineering practices among JS developers.

Caja does not allow any direct access to host DOM elements from Caja's guest context. Thus, AdCube creates a custom JS object called `TamedDOM` that contains the APIs presented in Table 4.3. Any API invocations other than defined APIs result in an execution error.

```
1:  let e = createElement("a-gltf-model");
2:  e.setAttribute("src", "product.gltf");
3:  e.addEventListener("click",onClick);
4:  addElement("adcube-id", e);
5:  function onClick(event){
6:    e.setAttribute("animation-mixer", "clip:animate");
7:  }
```

Listing 4.2: An example of ad-serving JS script

Listing 4.2 is a third-party ad-serving script example that implements VR content. The code creates an ad entity via `createElement()`, which is yet to be added to the scene. By leveraging the returned entity reference, the code sets the attribute that specifies the URL source of a 3D model and

attaches a click event handler that causes the model to animate. The invocation of `addElement()` appends this entity to the `adcube` that the host page defines via the `<a-adcube>` tag. Note that this `addElement()` could be an injection channel to insert DOMs and entities furnished with malicious JS code in their event handlers. Thus, AdCube implements filters that allow appending only A-Frame objects (e.g., `<a-gltf-model>` and `<a-obj-model>`) and forbid altering sensitive sink properties (e.g., `Element.innerHTML` and `Element.insertAdjacentHTML`) [87].

Host entities with `can-read` and `can-write` attributes are converted into `TamedDOM` objects. Thus, a third-party ad script can obtain the references of these objects via `querySelector()` or `querySelectorAll()`.

4.7.5 3D Ad Confinement

AdCube uses a bounding helper box, called a BBox [13], for publishers to confine the locations of ad entities, which addresses the first security requirement (Chapter 4.7.1). When loading or creating an ad entity within a specified BBox, AdCube resizes the entity to fit within the BBox. Note that VR axis scales often differ between the VR worlds of the entity and the underlying publisher's website. Therefore, we decided to resize ad entities that do not fit, instead of rejecting them.

This security enforcement requires AdCube to compute whether a specified BBox is able to contain a target entity. It is straightforward to compute whether primitive entities, such as boxes or spheres, fit within the hexahedron. AdCube simply does this by invoking the Box3 API in Three.js, which internally calculates an axis-aligned bounding box in 3D space.

However, checking whether a 3D model fits within a BBox entails a technical challenge; when the model is designed to animate or move around in a scene, it is necessary to compute the maximum size of the model at the time of loading. That is, AdCube should estimate the maximum size of this model and ensure that its estimated size fits within the specified BBox.

We tackle this challenge by playing a target model one-time before attaching this model to a scene. The idea is to sample frames while rendering the target 3D model and compute the maximum boundary of the shapes in these frames.

To this end, we project the model into 2D space and sample frames during the animation loop, which runs once. We then find the maximum size of the shape by scanning the pixels in the captured frames. Because only information for two axes is obtained in the 2D projection, we then rotate the camera angle (e.g., from front to side) and repeat the operation to retrieve information for three axes. AdCube projects a model onto the x-, y-, and z-axes and overlays the frames rendered during the animation. AdCube obtains the min/max positions of the pixels that are not the same color as a background and calculates the maximum BBox.

It is possible to append multiple ad entities to a single `adcube` space. For this, we use a Three.js Group object [41]. The Group object allows the management of multiple entities, including their children, as a single entity. We update the Group object when a new ad is added and adjust the scale of the entire group to prevent it from escaping the `adcube`.

4.8 Evaluation

This Chapter describes a showcase of WebVR ads enabled by AdCube (Chapter 4.8.1). We then evaluate the security of AdCube (Chapter 4.8.2) and the performance of AdCube (Chapter 4.8.3).

4.8.1 Ad Showcase

We conducted a preliminary study investigating on-going VR ad campaigns offered by OmniVirt [101], Adverty [7], and Admix [5]. They support three kinds of VR ad campaigns: i) billboard ads, ii) entity ads, and iii) image ads. A billboard ad campaign renders its video or image on a billboard in a VR scene. An entity ad campaign places a 3D ad object in a VR scene. An image ad campaign places an image of which the z-depth is zero in a VR scene.

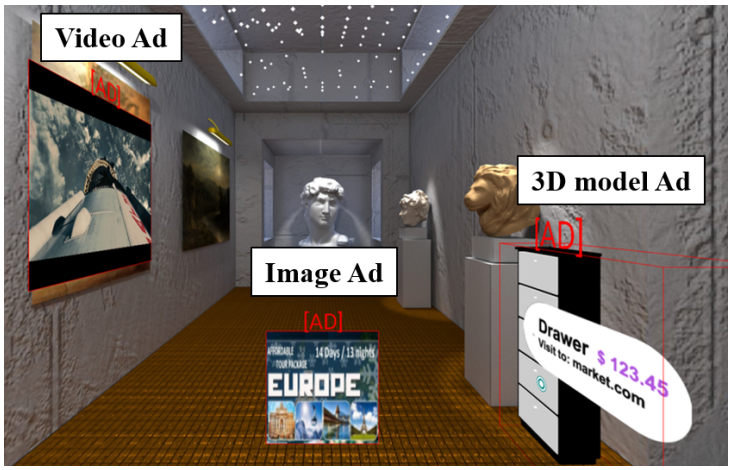


Figure 4.6: A showcase of WebVR ads with AdCube.

To demonstrate that AdCube supports each VR ad type, we implemented an ad showcase on WebVR, as shown in Figure 4.6. In the figure, the underlying VR environment is an art museum where users can experience VR art content [121]. All the entities within the red-bordered hexahedrons are from sandboxed ad-serving scripts. The billboard on the left wall renders a video ad campaign, and the one on the floor renders an ad impression promoting hotels in Europe. This image rendering display is attached to the current camera, thus varying in accordance with the user's current line of sight. On the right side, the third-party script draws a 3D drawer model.

4.8.2 Security

To evaluate the security provided by AdCube, we checked whether it is able to block all four of the presented attacks (Chapter 4.5). We assume that the adversary is an ad service provider that delivers the ad entities in Figure 4.6. For this experiment, we implemented a host website with the default security policy that specifies no `can-read` and `can-write` properties.

The default policy provides no reference point to a confined third-party script so that the script complied via Caja becomes unable to obtain a current camera position, insert new fake cursors, or modify any DOMs in the host page, thus rendering all the presented attacks ineffective. Note that this default policy blocks third-party scripts from reading or writing any first-party elements, including cookies, thus mitigating traditional web threats [76, 160].

A publisher may grant `can-read` and `can-write` access to their host camera and attach an `<a-adcube>` tag to the current camera, which makes this adcube area to move along with the camera perspective. However, to prevent the BST attack, AdCube prohibits the z position value from being a positive value when the adcube tag has the camera as its parent. Furthermore, all fake gaze and controller cursors that third-party scripts generate will be visually distinguishable from their host scene because these cursors

will be confined within a helper box with the "AD" label.

Note that it is feasible to abuse an auxiliary display when the publisher allows a third-party script to revise the host page. However, this can be easily blocked by carefully assigning `can-write` properties to host DOMs. The extension of such a policy can also block a third-party script from accessing private user information and credentials belonging to the host page, which is an original security goal of Caja.

We also emphasize that an `<a-adcube>` tag visually confines VR entities within this adcube area. When AdCube adds or loads VR entities in an adcube area, it ensures that these loaded entities do not escape from this area.

4.8.3 Performance

We evaluated the performance overhead of AdCube and compared it with two other methods: Baseline and Mirroring. The baseline method is to run a third-party script without any underlying security defense, thus running it with the same origin as its host. For the other method, we chose an origin-based execution separation method that runs the third-party script in a separate origin different from its host origin. For this, we referred to AdJail [74], which provides a secure ad service using the origin separation method in a standard web environment. This approach leverages the SOP enforcement of the browser and uses a postMessage API [97] for communications between different origins. The unique feature of this approach is to mirror any entity updates on the host page in a separate third-party iframe to address the scenario in which the first- and third-party contents interact with each other. AdJail only mirrors the static content types of ads that are not necessary to be rendered on the mirror page. However, in a WebVR environment, the mirror page must have a VR scene in order to sync ad behaviors between the two origins; therefore, rendering the scene in both pages is inevitable. We implemented this AdJail approach (denoted by Mirroring) for the comparative study.

Experiment setup. For each defense approach, we measured the page loading time and FPS on a machine with Intel i7 CPU, GeForce GTX 1060, and 32GB of main memory. All experiments were conducted using Firefox 78.0.2.

Loading latency. To understand the overhead of deploying AdCube, we measured the loading time of a WebVR webpage. When a page is requested, all three approaches (i.e., Baseline, Mirroring, and AdCube), request a VR library (e.g., A-Frame) and a third-party ad script and then render the host scene. We assumed that a target website is furnished with a Caja library because this library is a part of AdCube, and AdCube is a defense system for website owners. AdCube establishes an execution environment for Caja. It then parses adcube tags on the host page and renders third-party ad entities into adcube areas after resizing these entities. On the other hand, the Mirroring approach generates a guest page within an iframe and loads the required resources onto both the guest and host pages. It then renders ad entities on the guest page and mirrors these entities on the host page.

For the experiment, we used WebVR showcases on the A-Frame official site [1]. Of 17 showcases, we collected a total of nine open source apps that use the later versions of A-Frame 0.6. These WebVR sites comprise diverse demo purposes (Hello WebVR, Lights, Anime UI), games (A-Blast, Super Says), and utilities (360 Image, 360 Image Gallery, A-Painter, A Saturday Night).

For the guest code to be sandboxed, we created an ad-serving JS script that loads a static 3D chair model and applied three approaches to it. We also specified a security policy for each website that specifies three host entities with the can-write property, with which the ad-serving guest code is able to interact. We measured the page loading time ten times using Firefox with cache enabled and reported the average.

Table 4.4: Comparison of the average page loading times for nine WebVR sites and the average FPS for 12 events on the showcase with Baseline, Mirroring, and AdCube.

Performance Evaluation	Baseline	Mirroring	AdCube
Average Loading Time (s)	0.55	0.95	0.78
FPS (drop rate)	56.70 (-)	53.12 (6.32%)	55.79 (1.60%)

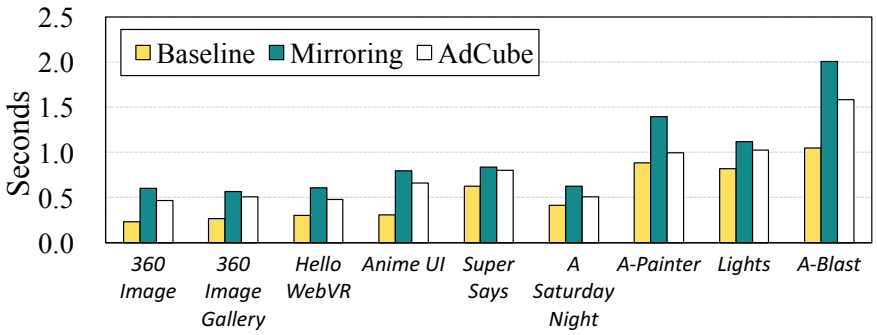


Figure 4.7: A comparison of page loading times between Baseline, Mirroring, and AdCube for nine WebVR sites.

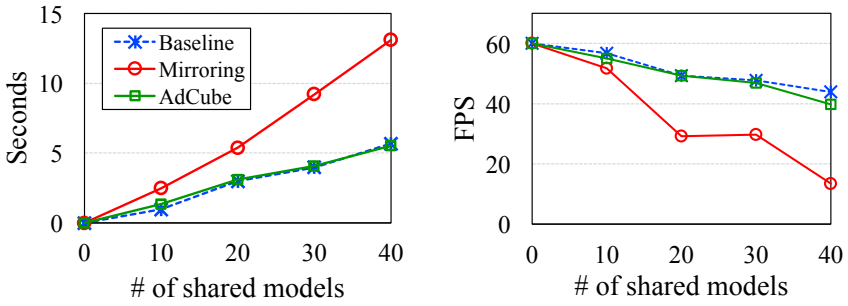
Figure 4.7 represents the page loading time of the nine WebVR demo websites using three approaches: Baseline, Mirroring, and AdCube. The Saturday Night and A-Blast websites exhibited the smallest and largest overheads for AdCube, reporting an additional 95 and 537 msec, respectively. On average, the page loading time of the nine demo sites with AdCube took an additional 236 msec, compared to an additional 406 msec with Mirroring. Furthermore, the page load time for each website with AdCube was consistently smaller than with the Mirroring approach. As Table 4.4 shows, the average loading time of the nine WebVR websites was 0.55 sec (Baseline). When applying the Mirroring and AdCube methods, the average loading times were 0.95 and 0.78 sec, respectively.

To understand this observed loading latency by AdCube, we further measured the rendering time by subdividing steps. The rendering time of AdCube includes the execution time of Caja, which can be divided into three steps: 1) requesting caja.js and connecting with the Caja's server; 2) making the host code accessible to the guest code; and 3) loading the guest code and cajoling the code.

Table 4.5: Overall rendering latencies (msec.) for Lights and A-Blast where having Caja's minimum and maximum execution overhead, respectively.

Website	Caja Execution Time			Total Rendering Time
	Step 1	Step 2	Step 3	
Lights	255.9	1.1	1.6	1016.6
A-Blast	1125.2	1.3	2	1533

Table 4.5 shows the overall loading time for websites with Caja's minimum and maximum execution



(a) Page loading time variations

(b) FPS variations

Figure 4.8: Page loading time of a testing page while varying the number of shared 3D models.

overhead from a total of nine websites. Caja's execution resulted in rendering latencies of 25.44% for Lights and 73.61% for A-Blast. This means that more than 25.44% of the rendering latency for all the nine websites using AdCube is due to the Caja setup. That is, the initialization time of Caja dominated the observed page loading times, whereas the latency for cajoling ad-serving JS scripts was small, which was less than 2 msec.

For the A-painter, Lights, and A-Blast Baseline websites that exhibited relatively longer page loading times, the Mirroring approaches of their corresponding websites also exhibited greater overheads. This is because the Mirroring approach inevitably entails the redundant loading of resources onto the guest page, which means that the more objects rendered, the greater the overhead. On the other hand, the initialization of Caja dominates the page loading time of AdCube, which happens only for the first visit.

Note that we chose an arbitrary number of three host elements that interact with a sandboxed ad-serving JS code for the experiment. Thus, we further measured variations of page loading time as we increased the number of elements shared between a host page and its ad-serving JS code. For the host page, we implemented an empty VR world and added a given number of shared 3D models, which were randomly chosen from 3D static models at Sketchfab [119]. Figure 4.8(a) shows the experimental results, which demonstrate that the page loading time in the Mirroring approach increases significantly with the mirroring of many 3D models. This demonstrates that AdCube is more scalable than the Mirroring approach to cope with an arbitrary number of shared entities.

FPS. To assess the overall performance during the exploration of a WebVR world, we measured the FPS change for each approach while the WebVR website was running. We experimented with the museum site presented in Chapter 4.8.1, including the various types of advertising campaigns. To show FPS variations, we added entities of museum sculptures to the virtual scene to intentionally lower the FPS rate.

Existing VR advertising services [7, 101] provide interaction behaviors for each type of ad. For example, accordance with user interactions, video ads can be loaded, image ads can be resized, or the animation states of 3D models can be changed. Based on this, to measure the impact of user and publisher interactions on FPS, we also defined 12 different user events and implemented their corresponding event handlers. We then forced them to trigger every 10 seconds using `setTimeout()`.

The following list entails nine user events and three publisher events that we used to measure FPS drops in the ad showcase in Figure 4.6:

- e1: Load and play a video ad

- e2: Attach an image ad to a camera entity
- e3: Resize the image ad
- e4: Load a 3D model ad
- e5: Change an animation status of the 3D model ad
- e6: Replace the video ad with another one
- e7: Replace the image ad with another one
- e8: Replace the 3D model ad with another one
- e9: Modify the host entity with permission
- e10: Hide the video ad (publisher event)
- e11: Change the location of the image ad (publisher event)
- e12: Resize the 3D model ad (publisher event)

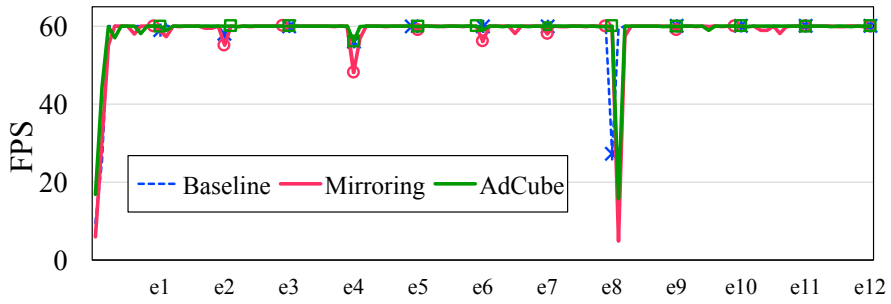


Figure 4.9: FPS drops in three approaches in response to interaction events.

Figure 4.9 shows measured FPS variations over time and across events. Note that Firefox caps its FPS at 60. A targeted museum site loads many entities at the initial time, resulting in a significant FPS reduction in the early stages of all three approaches, including Baseline.

AdCube exhibited a reliable performance that aligned with Baseline's along all timelines. In contrast, Mirroring struggled with unstable performance when the events occurred. Especially, in the eighth event, replacing a 3D model ad, Mirroring resulted in a decrease of FPS that was about 1.5 times that of Baseline.

Table 4.4 shows the average FPS of our showcase website when the 12 events were triggered for Baseline, Mirroring, and AdCube, respectively. Note that Firefox caps its FPS at 60. The average FPS for Mirroring is 53.12, which is an additional 6.32% decrease from the Baseline approach. On the other hand, AdCube exhibited a 1.60% decrease (55.79 FPS), which shows a negligible FPS drop from the Baseline approach. This means that AdCube provides stable performance even when various events occur.

We further measured the FPS variations while increasing the number of objects shared between a host page and ad-serving JS scripts. We used the same empty VR webpage used for measuring the page loading time variations as increasing the number of shared objects. We also measured the average FPS for 20 seconds after the page completes loading. As shown in Figure 4.8(b), when reaching 40 shared objects, the average FPSs for Baseline, Mirroring, and AdCube decreased to 43.86, 13.45, and 39.75, respectively. AdCube exhibited an FPS drop similar to that of the Baseline approach. The experimental

results indicate that AdCube is a practical solution compared to Mirroring in WebVR, in which FPS drops are critical. Note that an abrupt FPS decrease reduces the user's sense of immersion in the VR mode and may cause a poor user experience [16].

4.9 Discussion

This Chapter discusses other possible defense methods against the proposed WebVR attacks and their limitations.

Visibility reporting. One may implement a visibility reporting approach that attaches observers [96] to VR ad entities to check their visibility to users, thus mitigating BST attacks. This requires revising existing 3D JS libraries or frameworks (e.g., *Three.js* and *A-Frame*) to compute the intersections between ad entities and users' viewports. However, this type of defense does not block the AAD attack because ad entities are actually rendered in the auxiliary display. Also, the adversary may conduct a GCJ or CCJ attack that induces a victim to trigger clicks on ad entities when the victim watches or clicks non-promotional entities. That is, visibility reporting does not address GCJ, CCJ, or AAD attacks because these attacks stem from no access control when third-party scripts read or revise first-party resources.

HTC supports the eye-tracking API [145], which can be used for visibility reporting. However, this API is unavailable to WebVR, and the current specification [148] does not define interfaces for retrieving eye-tracking information. Furthermore, allowing access to user's eye movements would entail a privacy risk by third parties abusing the information, which necessitates sandboxing third-party scripts.

We also believe that WebXR specification changes cannot address GCJ, CCJ, or AAD attacks. Blocking these attacks requires restricting third-party script behaviors; however, the specification is designed to define interfaces for providing VR worlds and peripherals.

Native browser support. One possible defense is to integrate AdCube with a browser engine, thereby sandboxing third-party scripts. We believe that native browser supports for sandboxing general websites require a long-term development plan with significant engineering effort. Implementing browser-level sandboxing requires identifying the source of a given script to execute; this is because the browser should determine whether to sandbox a given script based on its source. However, the dynamic nature of JS makes it difficult to determine the true sources of dynamically generated JS scripts when the generation involves multiple origins.

Furthermore, it is important to maintain the creator's origin of each DOM element because a browser should determine the accessibility of such DOM elements. However, this requires significant changes to today's modern browsers. Chrome developers discussed implementing a similar functionality of tracking the creators of dynamically generated iframe DOMs and concluded that its implementation would introduce numerous corner cases, providing a false sense of security [22].

We propose a practical sandboxing tool that requires no change to browsers. AdCube addresses the aforementioned two challenges by not allowing dynamically generated scripts and leveraging security policies specified by publishers.

AR support. Recently, WebAR services [79, 105] have been introduced, and several vendors [11, 12, 71, 138] have provided JS libraries that enable AR services in websites. A website owner is able to pop up 3D augmented entities in a user's mobile browser when this user's camera looks at a marked predefined for user recognition. AdCube can be integrated with such a WebAR service; it is able to visually confine augmented entities from untrustworthy third parties and to sandbox their execution when they come with JS scripts.

4.10 Summary

Assuming a malicious adversary who abuses the lack of built-in browser support of sharing canvas DOMs, we have devised four new attack variants to conduct VR ad fraud. Our user study showed that the devised attacks are effective in conducting stealthy impression and click fraud. To defend against the presented threats, we proposed AdCube, which allows honest publishers to confine the locations of ad entities as well as to sandbox third-party ad scripts. We advocate ad service providers and publishers to alarm the presented risks in WebVR and adopt AdCube.

Chapter 5. Conclusion

The Web is constantly embracing new technologies as shown by the browser's frequent update cycles. However, when new technologies are introduced on the Web, some unwanted situations arise in which the traditional security-related principles of the Web are lost or not extended. In order to reduce the damage of Web attacks, it is necessary to investigate these situations in advance and come up with countermeasures. In this dissertation, we performed in-depth studies on two recent Web technologies, PWA and WebVR. PWA provides native app features such as push notification and offline mode through a background execution component called service worker. The persistence of service workers has opened a new execution paradigm for Web hosts to trigger execution at any time. With this new execution pattern, we showed that existing Web Attacks such as phishing, browsing history sniffing, and cryptojacking attacks can be performed more easily and powerfully in PWA. We then presented guidelines to defend against them. Secondly, WebVR is a technique that enables VR on the Web, making VR content accessible easily. However, the Web's sandboxing technique which securely executes multiple origins in 2D space is not simply applied to 3D space, making it difficult to provide isolated rendering area between multiple origins in the WebVR environment. To address this problem, we proposed AdCube that allows publishers to specify the behaviors of third-party ad library and enforce this specification. The Web is widely adopted and has the characteristics that rapidly evolving, it is necessary to provide fast and practical defense techniques against new security threats. At the same time, research on redesigning secure Web browsers will bring us closer to building a safe browsing environment.

Bibliography

[1] A-Frame. A WebVR Implementation Platform. https://aframe.io/docs/0.9.0/introduction/.

[2] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2014.

[3] S. V. Acker, P. D. Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: Least-Privilege Integration of Third-Party Components in Web Mashups. In *Proceedings of the Annual Computer Security Applications Conference*, 2011.

[4] D. Adams, A. Bah, C. Barwulor, N. Musaby, K. Pitkin, and E. M. Redmiles. Ethics Emerging: The Story of Privacy and Security Perceptions in Virtual Reality. In *Fourteenth Symposium on Usable Privacy and Security*, 2018.

[5] Admix. An Online Advertising Service. https://admix.in/.

[6] Admix. State Farm Case Study. https://admix.in/case-studies/state-farm/.

[7] Adverty. An Online Advertising Service. https://adverty.com/.

[8] P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: Complete Client-side Sandboxing of Third-party JavaScript without Browser Modifications. In *Proceedings of the Annual Computer Security Applications Conference*, 2012.

[9] Urban Airship. https://www.urbanairship.com/.

[10] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song. Clickjacking Revisited: A Perceptual View of UI Security. In *USENIX Workshop on Offensive Technologies*, 2014.

[11] argon.js. WebAR JS Library. https://www.argonjs.io.

[12] AR.js. WebAR JS Library. https://github.com/AR-js-org/AR.js.

[13] G. Barequet and S. Har-Peled. Efficiently Approximating the Minimum-Volume Bounding Box of a Point Set in Three Dimensions. *J. Algorithms*, 38(1):91–109, 2001.

[14] A. Barth, C. Jackson, and J. Mitchell. Securing Frame Communications in Browsers. In *Proceedings of the USENIX Security Symposium*, 2008.

[15] T. Blizard and N. Livic. Click-Fraud Monetizing Malware: A Survey and Case Study. In *International Conference on Malicious and Unwanted Software*, 2012.

[16] J. E. Bos and W. B. and E. L. Groen. A Theory on Visually Induced Motion Sickness. *Displays*, 29(2):47–57, 2008.

[17] Anca Bradley. Avoid These 7 Annoying Ad Placement Techniques on Your Site. https://www.entrepreneur.com/article/240098.

[18] Bugzilla. iFrame Onload Event Does Not Fire. https://bugzilla.mozilla.org/show_bug.cgi?id=444165.

[19] Facebook Ads Help Center. About Descriptions in News Feed Ads. https://www.facebook.com/business/help/1130862553791128.

[20] P. Chapman and D. Evans. Automated Black-box Detection of Side-channel Vulnerabilities in Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

[21] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[22] Chromium. Issue 1615523002: Transitively Keep Track of an Isolated World's Children Scripts and Worlds. https://codereview.chromium.org/1615523002/.

[23] Chromium. Javascript iFrame Onerror Event. https://bugs.chromium.org/p/chromium/issues/detail?id=365457.

[24] Coinhive. Coinhive – Monero JavaScript Mining. https://coinhive.com/.

[25] Federal Trade Commission. Advertising and Marketing on the Internet : Rules. https://www.ftc.gov/tips-advice/business-center/guidance/advertising-marketing-internet-rules-road.

[26] M. Cova, C. Kruegel, and G. Vigna. There is No Free Phish: An Analysis of Free and Live Phishing Kits. In *Proceedings of the Conference on USENIX Workshop on Offensive Technologies*, 2008.

[27] J. Crussell, R. Stevens, and H. Chen. MAdFraud: Investigating Ad Fraud in Android Applications. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, 2014.

[28] A. Das, G. Acar, N. Borisov, and A. Pradeep. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2018.

[29] Apple Developer. Apple Certificates Support. https://developer.apple.com/support/certificates/.

[30] Chrome Developer. Chrome Extentions - Content Settings. https://developer.chrome.com/extensions/contentSettings#type-ContentSetting.

[31] Google Developers. AliExpress. https://developers.google.com/web/showcase/2016/aliexpress.

[32] Google Developers. Flipkart Triples Time-on-site with Progressive Web App. https://developers.google.com/web/showcase/2016/flipkart.

[33] Google Developers. Introduction to Caja. https://developers.google.com/caja.

[34] Google Developers. Introduction to Progressive Web Apps. https://codelabs.developers.google.com/pwa-dev-summit.

[35] Google Developers. Introduction to Push Notifications. https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications.

[36] Google Developers. Introduction to Secure EcmaScript. https://github.com/google/caja/wiki/SES.

[37] Google Developers. Mythbusting HTTPS. http://www.codechannels.com/video/Chrome/chrome/mythbusting-https-progressive-web-app-summit-2016/.

[38] Google Developers. PWA Case Studies. https://developers.google.com/web/showcase.

[39] Google Developers. Web Push Protocol. https://developers.google.com/web/fundamentals/push-notifications/web-push-protocol.

[40] R. Dhamija, J. Tygar, and M. Hearst. Why Phishing Works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006.

[41] Three.js Docs. Objects - Group. https://threejs.org/docs/#api/en/objects/Group.

[42] Chromium Documents. Do Service Workers Live Forever? https://github.com/chromium/chromium/blob/master/docs/security/service-worker-security-faq.md#do-service-workers-live-forever.

[43] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang. Protecting Sensitive Web Content from Client-Side Vulnerabilities with CRYPTONS. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2013.

[44] X. Dong, M. Tran, Z. Liang, and X. Jiang. AdSentry: Comprehensive and Flexible Confinement of JavaScript-based Advertisements. In *Proceedings of the Annual Computer Security Applications Conference*, 2011.

[45] E. Felten and M. Schneider. Timing Attacks on Web Privacy. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2000.

[46] D. Florencio and C. Herley. Password Rescue: A New Approach to Phishing Prevention. In *1st USENIX Workshop on Hot Topics in Security*, 2006.

[47] Linux Foundation. Let's Encrypt. https://letsencrypt.org/.

[48] FoxPush. https://www.foxpush.com/.

[49] Jorge Fuentes. HalloVReen: A WebVR Experiment for Kids. https://www.jorgefuentes.net/projects/halloVReen/.

[50] C. George, M. Khamis, E. Zezschwitz, M. Burger, H. Schmidt, F. Alt, and H. Hussmann. Seamless and Secure VR: Adapting and Evaluating Established Authentication Systems for Virtual Reality. In *Usable Security Workshop on NDSS*, 2017.

[51] T.V. Goethem, M. Vanhoef, F. Piessens, and W. Joosen. Request and Conquer: Exposing Cross-Origin Resource Size. In *Proceedings of the USENIX Security Symposium*, 2016.

[52] Google. Google Safe Browsing. https://developers.google.com/safe-browsing/.

[53] W3C Groups. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/.

[54] W3C Groups. Notification API. https://notifications.spec.whatwg.org/.

[55] W3C Groups. Push API. https://w3c.github.io/push-api/.

[56] W3C Groups. Service Workers Nightly. https://w3c.github.io/ServiceWorker/.

[57] W3C Groups. Web Workers. https://w3c.github.io/workers/.

[58] G. Guninski. Frame spoofing using loading two frames. https://bugzilla.mozilla.org/show_bug.cgi?id=13871.

[59] X. Han, N. Kheir, and D. Balzarotti. PhishEye: Live Monitoring of Sandboxed Phishing Kits. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2016.

[60] Google Ads Help. How do I Stop Ads From Covering Text? https://support.google.com/google-ads/thread/1452412?hl=en.

[61] L. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium*, 2012.

[62] IAB. Digital Advertising. https://www.iab.com/wp-content/uploads/2016/04/HTML5forDigitalAdvertising2.0.pdf.

[63] L. Ingram and M. Walfish. Treehouse: Javascript Sandboxes to Help Web Developers Help Themselves. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[64] Izooto. https://www.izooto.com/.

[65] Dean Jackson and Jeff Gilbert. WebGL Specification. https://www.khronos.org/registry/webgl/specs/latest/1.0/.

[66] T. Jagatic, N. Johnson, M. Jakobsson, and F. Menczer. Social Phishing. *Commun. ACM*, 2007.

[67] John Koetsier. Mobile Ad Fraud: What 24 Billion Clicks on 700 Ad Networks Reveal. https://blog.branch.io/mobile-ad-fraud-what-24-billion-clicks-on-700-ad-networks-reveal/.

[68] K. Lebeck, K. Ruth, T. Kohno, and F. Roesner. Securing augmented reality output. In *2017 IEEE symposium on security and privacy (SP)*, pages 320–337. IEEE, 2017.

[69] S. Lee, H. Kim, and J. Kim. Identifying Cross-origin Resource Status using Application Cache. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[70] T. Lee. How Bitcoins Became Worth $10,000. https://arstechnica.com/tech-policy/2017/11/how-bitcoins-became-worth-10000/.

[71] Letsee. WebAR SDK. https://www.letsee.io/ko/.

[72] W. Li, H. Li, H. Chen, and Y. Xia. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, 2015.

[73] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[74] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the USENIX Security Symposium*, 2010.

[75] T. Majchrzak, A. Biørn-Hansen, and T. Grønli. Progressive Web Apps: the Definite Approach to Cross-Platform Development? In *Hawaii International Conference on System Sciences*, 2018.

[76] D. Malandrino and V. Scarano. Privacy Leakage on the Web: Diffusion and Countermeasures. *Computer Networks*, 57(14):2833–2855, 2013.

[77] I. Malavolta. Beyond Native Apps: Web Technologies to the Rescue. In *Proceedings of the 1st International Workshop on Mobile Development*, 2016.

[78] I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirovic. Assessing the Impact of Service Workers on the Energy Efficiency of Progressive Web Apps. In *International Conference on Mobile Software Engineering and Systems*, 2017.

[79] First Man. WebAR Serve. https://moon.firstman.com.

[80] M. Marciel, R. Cuevas, A. Banchs, R. Gonzalez, S. Traverso, M. Ahmed, and A. Azcorra. Understanding the Detection of View Fraud in Video Content Portals. In *Proceedings of the International Conference on World Wide Web*, 2016.

[81] R. McPherson, S. Jana, and V. Shmatikov. No Escape From Reality: Security and Privacy of Augmented Reality Browsers. In *International World Wide Web Conference*, 2015.

[82] L. A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[83] P. Milgram, H. Takemura, A. Utsumi, and F. Kishino. Augmented reality: A class of displays on the reality-virtuality continuum. In *Telemanipulator and telepresence technologies*, 1995.

[84] mitmproxy. https://mitmproxy.org/.

[85] Moloco. The "Axis of Evi" in Mobile Ad Fraud. https://medium.com/@moloco/bad-ad-networks-the-axis-of-evil-in-mobile-ad-fraud-89ca577de2b6.

[86] T. Moore and R. Clayton. Discovering Phishing Dropboxes using Email Metadata. In *eCrime Researchers Summit*, 2012.

[87] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2019.

[88] Mozilla Developer Network. AppCache is deprecated. https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache.

[89] Mozilla Developer Network. Cache - Web APIs. https://developer.mozilla.org/en-US/docs/Web/API/Cache.

[90] Mozilla Developer Network. CSP: frame-ancestors - HTTP. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors.

[91] Mozilla Developer Network. Document Object Model (DOM). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

[92] Mozilla Developer Network. HSTS - Strict Transport Security. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security.

[93] Mozilla Developer Network. HTML Canvas Element. https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement.

[94] Mozilla Developer Network. HTTP caching. https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching.

[95] Mozilla Developer Network. iframe: Inline Frame Element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.

[96] Mozilla Developer Network. Intersection Observer. https://developer.mozilla.org/ko/docs/Web/API/Intersection_Observer_API.

[97] Mozilla Developer Network. postMessage() API. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.

[98] Mozilla Developer Network. Same-Origin Policy (SOP). https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[99] Mozilla Developer Network. WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly.

[100] Web of Trust. Web of Trust - Website reputation and review service. https://www.mywot.com/.

[101] Omnivirt. An Online Advertising Service. https://www.omnivirt.com/.

[102] OneSignal. https://onesignal.com/.

[103] PlayCanvas. A WebVR Implementation Platform. https://playcanvas.com/industries/vr.

[104] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *Proceedings of the USENIX Security Symposium*, 2011.

[105] Purina. WebAR Serve. https://one28daychallengear.purina.com.

[106] pushcrew. https://pushcrew.com/.

[107] PushEngage. https://www.pushengage.com/.

[108] PushWoosh. https://www.pushwoosh.com/.

[109] Vhite Rabbit. WACKARMADIDDLE: A WebVR Wack-A-Mole game. https://constructarca.de/game/wackarmadiddle/.

[110] A. Ramachandran and N. Feamster. Understanding the Network-level Behavior of Spammers. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2006.

[111] D. Ross. HTTP Header Field X-Frame-Options. https://tools.ietf.org/html/rfc7034.

[112] N.V. Saberhagen. CryptoNote v 2.0. https://cryptonote.org/whitepaper.pdf.

[113] G. Saride, J. Aaron, and J. Bose. Secure Web Push System. In *International Conference on Communication Systems and Networks*, 2016.

[114] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-scale Legacy Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

[115] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *Proceedings of the USENIX Security Symposium*, 2017.

[116] SendPulse. https://sendpulse.com/.

[117] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the USENIX Security Symposium*, 2012.

[118] D. Silver, S. Jana, E. Chen, C. Jackson, and D. Boneh. Password Managers: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium*, 2014.

[119] Sketchfab. A WebVR Implementation Platform. https://www.sketchfab.com.

[120] P. Snyder, C. Taylor, and C. Kanich. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2017.

[121] Cecropia Solutions. The Hall: A WebVR demo that displays art. https://cecropia.github.io/thehallaframe/.

[122] S. Son, D. Kim, and V. Shmatikov. What Mobile Ads Know About Mobile Users. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[123] S. Son and V. Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[124] A. K. Sood and S. Zeadally. Drive-By Download Attacks: A Comparative Study. *IT Professional*, 18(5):18–25, 2016.

[125] K. Springborn and P. Barford. Impression Fraud in On-line Advertising via Pay-Per-View Networks. In *Proceedings of the USENIX Security Symposium*, 2013.

[126] Statista. Active Virtual Reality Users Forecast WorldWide 2014-2018. https://www.statista.com/statistics/426469/active-virtual-reality-users-worldwide/.

[127] Statista. Global Consumer Spending: AR/VR Content and Apps 2021. https://www.statista.com/statistics/828467/world-ar-vr-consumer-spending-content-apps/.

[128] Statista. VR and AR Ownership in the U.S. by Age 2017. https://www.statista.com/statistics/740760/vr-ar-ownership-usa-age/.

[129] T. Steiner. What is in a Web View? An Analysis of Progressive Web App Features When the Means of Web Access is not a Web Browser. In *International World Wide Web Conference*, 2018.

[130] B. Stock, M. Johns, M. Steffens, , and M. Backes. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In *Proceedings of the USENIX Security Symposium*, 2017.

[131] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-scale Spam Campaigns. In *Proceedings of the Conference on Large-scale Exploits and Emergent Threats*, 2011.

[132] B. Stone-Gross, R. Stevens, A. Zarras, R. Kemmerer, C. Kruegel, and G. Vigna. Understanding Fraudulent Activities in Online Ad Exchanges. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, 2011.

[133] Symantec. Elliptic Curve Cryptography Certificates Performance Analysis. https://www.websecurity.symantec.com/content/dam/websitesecurity/digitalassets/desktop/pdfs/whitepaper/Elliptic_Curve_Cryptography_ECC_WP_en_us.pdf.

[134] Monero.org Team. Introduction to Monero (XMR) Coins. https://monero.org/.

[135] Techjury. 43 Virtual Reality Statistics That Will Rock The Market In 2020. https://techjury.net/stats-about/virtual-reality/#gref.

[136] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. Abu Rajab. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[137] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, D. Margolis, V. Paxson, and E. Bursztein. Data Breaches, Phishing, or Malware? Understanding the Risks of Stolen Credentials. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2017.

[138] three.ar.js. WebAR JS Library. https://github.com/google-ar/three.ar.js.

[139] Three.js. A JavaScript 3D Library. https://threejs.org/.

[140] Y. Tian, Y. C. Liu, A. Bhosale, L. S. Huang, P. Tague, and C. Jackson. All Your Screens Are Belong to Us: Attacks Exploiting the HTML5 Screen Sharing API. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.

[141] T. Tran, R. Pelizzi, and R. Sekar. JaTE: Transparent and Efficient JavaScript Confinement. In *Proceedings of the Annual Computer Security Applications Conference*, 2015.

[142] P. Vadrevu, J. Liu, B. Li, B. Rahbarinia, K. Lee, and R. Perdisci. Enabling Reconstruction of Attacks on Users via Efficient Browsing Snapshots. In *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[143] VentureBeat. 1 Billion AR/VR Ad Impressions. https://venturebeat.com/2018/12/05/1-billion-ar-vr-ad-impressions-what-weve-learned/.

[144] J. Vilk, D. Molnar, B. Livshits, E. Ofek, C. J. Rossbach, A. Moshchuk, H. J. Wang, and R. Gal. SurroundWeb: Mitigating Privacy Concerns in a 3D Web Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[145] Vive. VIVE Eye Tracking SDK. https://developer.vive.com/resources/vive-sense/sdk/vive-eye-tracking-sdk-sranipal/.

[146] Mozilla VR. A-Blast: A WebVR Wave Shooter Game. https://aframe.io/a-blast/.

[147] W3C. WebVR 1.1. https://immersive-web.github.io/webvr/spec/1.1/.

[148] W3C. WebXR Device API. https://www.w3.org/TR/webxr/.

[149] Z. Weinberg, E.Y. Chen, P.R. Jayaraman, and C. Jackson. I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[150] T. Whalen and K. Inkpen. Gathering Evidence: Use of Visual Security Cues in Web Browsers. In *Proceedings of the Graphics Interface*, 2005.

[151] WHATWG. HTML Living Standard. https://html.spec.whatwg.org/multipage/iframe-embed-object.html#the-iframe-element.

[152] WHATWG. Offline Web Applications. https://html.spec.whatwg.org/multipage/offline.html.

[153] WHATWG. the WebSocket API. https://html.spec.whatwg.org/multipage/web-sockets.html.

[154] Business Wire. Global Virtual Reality Content Creation Market Expected to Grow with a CAGR of 77.10% Over the Forecast Period, 2019-2026. https://www.businesswire.com/news/home/20200224005672/en/Global-Virtual-Reality-Content-Creation-Market-Expected.

[155] Wonderleap. An Online Advertising Service. https://wonderleap.co.

[156] M. Wu, R.C. Miller, and S.L. Garfinkel. Do Security Toolbars Actually Prevent Phishing Attacks? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006.

[157] Z. Xu and S. Zhu. Abusing Notification Services on Smartphones for Phishing and Spamming. In *Proceedings of the Conference on USENIX Workshop on Offensive Technologies*, 2012.

[158] S. Zawoad, A. Dutta, A. Sprague, R. Hasan, J. Britt, and G. Warner. Phish-Net: Investigating Phish Clusters using Drop Email Addresses. In *APWG eCrime Researchers Summit*, 2013.

[159] M. Zhang, W. Meng, S. Lee, B. Lee, and X. Xing. All Your Clicks Belong to Me: Investigating Click Interception on the Web. In *Proceedings of the USENIX Security Symposium*, 2019.

[160] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies Lack Integrity: Real-World Implications. In *Proceedings of the USENIX Security Symposium*, 2015.