

INVESTIGATING THE SECURITY OF EMERGING WEB TECHNOLOGIES AND THEIR ROLE IN SOCIAL ENGINEERING ATTACKS

by

KARTHIKA SUBRAMANI

(Under the Direction of Roberto Perdisci and Kyu Hyung Lee)

ABSTRACT

Owing to the drastic increase in the number of Internet users, web applications are moving towards providing a native-app like user experience. This transition has in turn given rise to a number of new web technologies such as Progressive Web Apps, Web Push Notifications, Captchas, Browser Fingerprinting etc. Unfortunately, in the never-ending fight between functionality and security, security time and again takes a back seat. Therefore, even though these technologies are developed for legitimate purposes to help web users and website owners, attackers find ways to leverage them for a number of malicious purposes. One such simple but devastating form of attacks is Social Engineering Attacks(SEAs) that despite being in existence for many years has proven difficult to combat and be dealt with. Social Engineering Attacks have been constantly evolving thus rendering the existing defenses ineffective. Further, attackers are always looking to pursue new avenues, such as leveraging new emerging web technologies, in order to evade detection and successfully carry out the attacks. To stay ahead of the curve and prevent attacks such as Social Engineering Attacks, it is imperative to gain a better understanding of these new web technologies from a security perspective and analyze their possible abuse.

In this dissertation, we first systematically analyze one of the integral components of a trending web technology, Service Workers(SWs), and categorize a number of existing attacks and new attacks possible via exploiting Service Worker features including Social Engineering Attacks. Then, we discuss a number of open SW security problems that are currently unmitigated, and propose SW behavior monitoring approaches and new browser policies that we believe should be implemented by browsers to further improve SW security. Furthermore, we implement a proof-of-concept version of several policies in the Chromium code base, and also measure the behavior of SWs used by highly popular web applications with respect to these new policies. Second, we develop a system called PushAdMiner that is dedicated to (1) *automatically* registering for and collecting a large number of web-based push notifications(WPNs) from publisher websites using instrumented browser, (2) finding WPN-based ads among these notifications, and (3) discovering the abuse of WPN-based notifications for Malvertising. Furthermore, we found that 51% of all WPN ads we collected are malicious, and that traditional ad-blockers and URL filters were

mostly unable to block them, thus leaving a significant abuse vector unchecked. Finally, we build a smart crawler that automatically inspects the phishing pages, infers the requested information and interacts with the page. Then, we analyze the information collected by crawler to observe and measure new traits followed in modern phishing sites that uses new emerging techniques such as service workers, human verification systems such as captchas, multi-factor authentication, browser fingerprinting.

INDEX WORDS: Social Enginnerring Attack, Service Workers, PWA, Progressive Web Apps, Malvertising, Phishing, Web Security

INVESTIGATING THE SECURITY OF EMERGING WEB TECHNOLOGIES AND THEIR
ROLE IN SOCIAL ENGINEERING ATTACKS

by

KARTHIKA SUBRAMANI

B.E., Sri Ramakrishna Engineering College, Tamil Nadu, India, 2012

A Dissertation Submitted to the Graduate Faculty of the
University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2021

©2021
Karthika Subramani
All Rights Reserved

INVESTIGATING THE SECURITY OF EMERGING WEB TECHNOLOGIES AND THEIR
ROLE IN SOCIAL ENGINEERING ATTACKS

by

KARTHIKA SUBRAMANI

Co-Major Professor: Roberto Perdisci

Co-Major Professor: Kyu Hyung Lee

Committee: Le Guan
Lakshmish Ramaswamy

Electronic Version Approved:

Ron Walcott
Dean of the Graduate School
The University of Georgia
December 2021

DEDICATION

I dedicate this work to my father, Dr. Subramani Jambulingam, and my mother, Mrs. Bhuvaneswari Subramani, who taught me to dream higher and went far and beyond to help me pursue my dreams.

To my brother, Karthikeyan Subramani, for he constantly pushes me to be better and supports me in all my endeavours.

To my husband, Sanath Bhat, for being my ardent believer and making this bumpy ride of a life endurable and enjoyable.

To my thatha, Mr. Balasubramanian, and ammachi, Mrs. Jothi, for being the warmest and kindest souls I have met and for being there for me at all times.

ACKNOWLEDGMENTS

Throughout my PhD program I have received a great deal of support and assistance. First and foremost, I am extremely grateful to Dr.Roberto Perdisci for helping me pursue my research in Web Security and for his invaluable expertise in guiding me through my research. I am in awe of his sheer dedication and passion towards teaching and research. I am also immensely grateful to Dr.Kyu Hyung Lee for his continued support and encouragement to push through and reach my goals. I am thankful to him for all the opportunities I was given to further my research. I would also like to thank Dr.Le Guan and Dr.Lakshmish Ramaswamy for their valuable input and mentorship.

I would like to thank my family members for their patience and help in dealing with other aspects of my life. I would like to thank my friends who provided stimulating discussions as well as happy distractions to rest my mind outside my research. I would also like to thank my teachers and colleagues throughout the years who believed in me and inspired me to dream bigger.

CONTENTS

Acknowledgments	v
List of Figures	vii
List of Tables	ix
1 Introduction and Literature Review	1
1.1 Introduction	1
1.2 Research Overview	3
1.3 Literature Review	5
2 SoK: Workerounds - Categorizing Service Worker Attacks and Mitigations	13
2.1 Abstract	14
2.2 Introduction	14
2.3 Background	16
2.4 Service Worker Abuse	18
2.5 Existing Mitigations	25
2.6 Open Problems and New Mitigations	28
2.7 Measuring In-The-Wild SW Behavior	33
2.8 Implementing New SW Policies	39
2.9 Related Work	41
2.10 Conclusion	42
3 When Push Comes to Ads: Measuring the Rise of (Malicious) Push Advertising	43
3.1 Abstract	44
3.2 Introduction	44
3.3 Motivating Example and Background	46
3.4 System Overview	47
3.5 Data Collection Module	49
3.6 Data Analysis Module	52
3.7 Measuring WPN Ads in the Wild	56

3.8	Related Work	66
3.9	Discussion and Limitations	67
3.10	Conclusion	68
4	SomethingPhishy: Investigating New Traits in Modern Phishing Attacks	69
4.1	Introduction	69
4.2	System Design	72
4.3	Data Collection Module: Smart Crawler	74
4.4	Data Analysis Module: Trait Analyzer	77
4.5	Experiments and Results	81
4.6	Related Works	90
4.7	Conclusion	91
5	Conclusion	92
	Bibliography	94
	Appendices	110
A		110

LIST OF FIGURES

2.1	Approximate timeline of attacks publication and mitigations. The ∞ symbol denotes new attacks discovered in this paper, which are not yet mitigated. Notice that some mitigations were implemented before official attack publication, perhaps thanks to responsible disclosure (the “*” next to WebBot denotes that a mitigation was described in online documentation related to Firefox, but our own tests show the attack appears to still be possible on that browser).	26
2.2	SW behavior measurements. Each graph displays the distribution (CDF) of occurrences of an event within a specific time window: a) Push count per hour; b) Push count per day; c) Third-party fetch count per SW activation; d) SW execution time per activation (in minutes); e) SW execution time per day (in minutes).	35
2.3	Examples of spam/malicious notifications	36
2.4	Third-Party(TP) Imports (a) Count of TP imported domains per SWs (b) Top 10 TP domains imported by SWs	38
3.1	Example of malicious advertisement served through web push notifications	46
3.2	PushAdMiner System Overview	48
3.3	Steps involved in Serving Ads via WPNs	51
3.4	Examples of WPN clusters	61
3.5	Graphical Representation Examples of Meta Clusters	63
3.6	Distribution of WPNs w.r.t. Ad Networks	65
4.1	Example of Multi-Step Phishing Pages targeting Paypal (a) Page verifying Human Interaction (b) Form requesting login credentials (c) Form requesting security questions and answers (d) Form requesting contact information and card Information (e) Final confirmation form	71
4.2	Overview of PhishTraits System	72
4.3	Example of Real-World Phishing Site using Image with Form Background to evade detection. (a) Phishing Site with a Form requesting Personal Information. (b) Background Image of the Form with the labels of the Input Fields	73

4.4	Example of Multi-Step Phishing Pages targeting Netflix and requesting Personal Credentials at each step (a) Phishing Form requesting login credentials. (b) Phishing Form requesting Card and contact information (c) Phishing Form requesting SMS Code (d) Phishing Form requesting Card information again	78
4.5	Examples of Captchas found in real Phishing sites	80
4.6	Plots displaying the Count of Field Categories found at different stages of the Phishing sites. (a) Displays count for the site labeled as “Multi-Staging” (b) Displays count for the rest of the visited sites	83
4.7	Plot displaying the Count of Field Categories found at different stages of crawling legitimate sites	85
4.8	Few Examples of Target Domain based Phishing Clusters. Red nodes indicate the landing page of phishing sites. Yellow nodes indicate intermediate domains. Blue node represents final target domain that the user was navigated to.	85
4.9	Few Examples of Captcha Detection Results on Real-world Phishing Pages (a) Test cases where the captchas were correctly detected with a confidence score of 85% or more. (b) Test cases where the model failed to detect the captchas or detected the wrong element.	87
4.10	Few Examples of Phishing pages that employed two-factor authentication and requested for SMS code.	89
A.1	Example of double permission prompt in use on a popular website.	III

LIST OF TABLES

1.1	Survey of Phishing Detection and their Defenses	10
2.1	Overview of attacks and impacted browser versions. Legend: (●) first attack impact; (○) fix released; (◐) partial fix released; (⊠) no fix released yet; (☆) attack not possible. . . .	19
2.2	Ranking bands for Alexa's top 100K websites	34
2.3	Event counts at specific distribution percentiles – The <i>threshold value</i> is the percentile value from the corresponding CDF (see Figure 2.6), whereas B- <i>n</i> represents the Alexa ranking band <i>n</i>	37
3.1	URLs and Notification Permission Request counts	58
3.2	URL rankings on Alexa's top 1M sites	58
3.3	Measurement Results of Data analysis Module	59
3.4	Measurement Results at Stages of Clustering	62
3.5	Examples of Singleton Clusters	64
3.6	Results on Existing Ad Blockers	66
4.1	Field Classifier results reporting Precision, Recall and F1-score	82
4.2	Overall Count of Pages requesting each Field Categories	84
4.3	Captcha Detection Model results reporting Average Precision(AP) per class for testing set	86

CHAPTER I

INTRODUCTION AND LITERATURE REVIEW

I.1 Introduction

According to this report by BroadBandSearch [1], there has been rapid increase of 1,266% in the number of Internet users between 2000 and 2020. Following this drastic increase, web applications are moving towards providing a native-app like user experience. This transition has in turn given rise to a number of new web technologies such as Progressive Web Apps, Web Push Notifications, Captchas, Browser Fingerprinting etc. that are introduced to provide customized experience to web users and website owners. As always, vendors publishing new functionalities and attackers exploiting or abusing these new functionalities is a never-ending issue. This phenomenon has been observed since the dawn of Internet to the recent advent of IoT devices. Security has always known to take a back seat to introducing new and exciting functionalities leading to attack classified as “Abuse of Functionality” [2]. For example, attackers are still exploiting vulnerable features in Windows Operating System to perform destructive attacks such as WannaCry attack [3]. Similarly, as the IoT devices gained popularity, attackers turned these devices into zombies to perform large-scale DDoS attacks [4]. With respect to emerging web technologies, following the rapid growth of online advertising ecosystem, attackers have started using advertisements as a way to deliver malicious contents including phishing and survey scams to the end users [5]–[8]. Papadopoulos et al. [9] explain how features of Progressive Web Apps could be exploited by attackers for creating a Botnet to perform various attacks such as DDoS, Cryptojacking etc. Acharya and Vadrevu [10] demonstrate how fingerprinting techniques could be used to avoid being detected by automated crawlers. In [11], authors detect the use of browser fingerprinting for client-side cloaking by Phishing sites. These works attest the fact that newly introduced functionalities could still be abused at the current age and poses a challenge for security researchers to address the security issues in them before they are exploited.

In the recent few years, Social Engineering Attacks (SEAs) targeting businesses and home users have increased at a tremendous rate. According to FBI’s IC3 report in 2020 [12], Social Engineering Attacks were found to be the most common attacks and this trend is expected to continue in the upcoming years.

As per the report, *6.9 million* phishing and scam pages were created in 2020 mostly taking advantage of the COVID situation to scare users and take advantage of their vulnerable state. While Remote Code Execution attacks and DDoS attacks require higher level expertise to perform attacks at large scale, equally devastating web based attacks that doesn't require as much expertise is "Social Engineering Attacks". For example, the "2020 Twitter Bitcoin Scam" accruing over \$100K within minutes was a result of "Social Engineering Attacks". Social Engineering Attacks being in existence for many years has proven that they are difficult to combat and to be dealt with. One of the main reasons for such attacks to be successful is that they target Humans, the weakest link in cyber security and trick them into revealing critical information [13]. However, it should also be noted that Social Engineering Attacks have been adapting rapidly rendering the proposed defenses ineffective. Further, attackers are always looking to pursue new avenues, such as leveraging new emerging web technologies, in order to evade detection and successfully carry out the attacks. To stay ahead of the curve and prevent attacks such as Social Engineering Attacks, it is imperative to gain a better understanding of these new web technologies from a security perspective and analyze their possible abuse.

In order to defend against such modern Social Engineering Attacks that are constantly evolving, it is pertinent to analyze these new technologies from a security perspective. One of the most trending new technologies that has been adopted by most of the websites is "Progressive Web Apps". Progressive Web Apps(PWAs) help web applications provide a native app-like user experience to cross-platform applications. With these new developments, PWAs also gain access to more resources compared to their web counterparts. Therefore, it is imperative to analyze security issues and prevent any potential abuse. Service Workers [14] are a *powerful feature* [15] at the core of *Progressive Web Apps* [16], namely web applications that can continue to function when the user's device is offline and that have access to device sensors and capabilities previously accessible only by native applications. Web applications are focussing on bringing users a native-app like experience. As a result, browsers such as Chrome has a number of features such as geofencing, periodic background sync etc., that would be added in the near future to make this transition possible [14]. It should also be noted that other browsers such as Safari and Firefox have shown some restraint in implementing these new features in lieu of privacy issues [17]. Further, even some of the features that were primarily introduced for protecting users and websites such as Captchas and Multi-factor authentication have been abused by attackers to perform complex and targeted attacks [18]–[20].

Currently, there exists only a limited number of works that focus on the abuse of new emerging web technologies, especially with respect to Social Engineering Attacks. In this dissertation, we focus on addressing this gap and investigate some of the emerging web technologies including Service Workers, Captchas, Multi-Factor Authentication(MFA), Web Push Notifications etc., and measure their role in Social Engineering Attacks. Accordingly, we approach this problem in 3 parts. First, we perform an extensive analysis of Service Workers that are integral part of Progressive Web Apps. Despite there being certain security restrictions that already restrict the abuse of Service Workers, in our research we still find multiple ways how these security policies could be bypassed and abused for attacks including but not restricted to Social Engineering Attacks. Second, we analyze how ad networks use Web Push Notifications(WPNs) for

delivering advertisements and detect and measure if they are malicious in nature. As part of this analysis, we instrumented Chromium browser to log a number of events and properties related to service workers and web push notifications that helps us gain further insight into their working and ecosystem. Third, we focus on the latest trends that are used in Phishing that is most-common form of Social Engineering Attacks. Over the years there has been a plethora research works that focus on Phishing Attacks, their Evasion and detection [10], [11], [21]–[23]. As compared to those works, we focus on analyzing the new traits used in phishing attacks including multi-stage credential stealing, fingerprinting, use of Human verification system to gain user’s trust and evade detection.

The rest of the thesis is organized as follows. Section 1.2 gives a brief overview about the 3 research works included in this thesis. Section 1.3 discusses the state of the art and significant work concerning Social Engineering Attacks and the security analysis of new emerging technologies related to the research. Chapters 2, 3, 4 talk about our research works in much more detail. Chapter 5 concludes this dissertation followed by Appendices containing additional information to support the research.

1.2 Research Overview

Our research’s main focus is to investigate few of the new emerging web technologies and measure their involvement in a number of Social Engineering Attacks. Through this research, we aim to identify security vulnerabilities that the attackers might take advantage of and propose defenses that could be used against such attacks; systematically measure the use of new technologies in SEAs and shed light on the need for advanced defenses against SEAs. We provide a brief overview of our 3 research works as follows.

1.2.1 Security Analysis of PWAs

Progressive Web Apps(PWAs) help web applications provide a native app-like user experience to cross-platform applications. With these new developments, PWAs also gain access to more resources compared to their web counterparts. Therefore, it is imperative to analyze various security issues related to these new features. Specifically, we focus on ‘Service Workers’, one of the powerful components of PWAs. In practice, a Service Worker(SW) is a JavaScript Worker script with the following high-level properties: (i) it is installed by a web application rendered in a browser; (ii) after installation, the SW can act as a proxy for network requests issued by its web application, and can thus control how web content is retrieved(e.g., from a local cache or the network) and what content is eventually passed to the application; (iii) it is an event-driven process that runs in the background, even when its web application is not actively rendered on the browser, and that can be activated by the browser based on events such as receiving a web push message.

For this research, we reproduce and analyze known attack vectors related to SWs, and explore new abuse paths that have not previously been considered. We first systematize this information by grouping the attacks into different categories, based on the fundamental SW security weaknesses that make the attacks including Social Engineering Attacks possible. Afterwards, we analyze whether, how, and estimate

when these attacks have been published and mitigated by different browser vendors. Then, we discuss a number of open SW security problems that to the best of our knowledge are currently unmitigated. Accordingly, we propose SW behavior monitoring approaches and new browser policies that we believe should be implemented by browsers to further improve SW security. While preventing all types of SW abuse may not be possible, we aim to propose policies that can limit the damage that potential SW attacks can make while minimizing the impact the proposed browser changes may have on existing legitimate SW code.

1.2.2 Malicious Advertisements via Web Push Notifications

In this section, we focus on malicious advertisements that are delivered to the users via Web Push Notifications(WPNs). Web Push notifications refers to push notifications that a website can send to it's user via the user's browser. This new feature of HTML5 that allows browsers to push notifications have created a great opportunity for the website owners to reach and retain their customer base. On the other hand, it also serves as an effective platform for the websites to monetize by serving advertisements through these notifications. Almost all ad networks have started providing ad services via push notifications and websites prefer them to traditional ads due to the following benefits.

- First, Push Notifications can be sent to user at any point of time as opposed to traditional way which requires users to visit the websites.
- Second, traditional Ad blockers have almost minimal to zero control over these advertisements and hence these ads would be delivered to the users at a larger rate.

Service workers play a vital role in the delivery of these notifications to the users and as such there exists only couple of works that focuses on the security implication of service workers[9], [24]. In this research, we perform a detailed study on the working of WPNs and it's extent in the real world. To measure and analyze the prevalence of ads served via WPNs, we extend the modified Chromium browser of JSGraph [25] to collect logs at every step of the push notification service and automate multiple user interactions that are required to receive notifications from websites. Then, we perform an extensive analysis on the significant information collected in the logs measure the spread and exploitation of push notifications for malicious nature and use machine learning techniques to identify malicious or suspicious advertisements. The outcome of this research will enable to block such malicious advertisements possibly by integrating with existing defenses.

Few works exist that focuses on studying these adwares such as suspicious ads shown on live-streaming sites[26], ads leading to tech support scams[27], large-scale study on malicious advertisements [8], tracking social engineering attack campaigns using ads[5], their detection using sources such as virus total, Google Safe Browsing, URL Void etc., and mitigation using ad blockers and blacklists [6], [7]. However, these works are focused on advertisements that are served via web pages. Our research focuses on a new avenue to distribute ads that is unmitigated by the current defenses.

1.2.3 Investigation of Modern Phishing Sites

Phishing attacks thrive despite a number of existing counter-measures and continue to be one of the most common and dangerous form of Social Engineering Attacks(SEAs). According to Internet Crime report by FBI [12], a record-breaking number of **6.9 million** phishing pages were created in the year 2020. Their prevalence could be attributed to the following major reasons, 1) availability of ready-made attacks and services 2) deployable by even novice attackers 3) cheap scaling infrastructure and, 4) availability of a number of platforms including email, social media etc., to initiate the attack. Nowadays, phishing sites have become more sophisticated by incorporating modern technologies such as browser fingerprinting, multifactor authentication, captcha, notifications etc. to gain users' trust and to evade automated detection. Similar to the age when people falsely believed HTTPS sites to be safe, one cannot rely on the presence of captchas and multifactor authentication to blindly trust the websites. Attackers design phishing sites with an intent to closely resemble legitimate sites in terms of both appearance and functionality while using a number of cloaking techniques to evade or delay detection. In the current world, data remains one of the most valuable assets and attackers are focussed on harvesting the most information possible from the victims. As a result, there are also a number of phishing campaigns that follow a multi-step phishing process to request more than login credentials from the users including personal contact information and payment information. Further, in order to avoid suspicion, attackers request different information using successive pages to successfully harvest more information.

Although phishing is a well-investigated topic [10], [11], [21]–[23], [28], number of research works that focus on such new traits of phishing methods is limited. Therefore, in this research, we focus on investigating the modern phishing pages that employ new techniques and multi-layered campaigns to steal credentials from users. Specifically, we use a crawler that automatically interacts with the forms in any encountered page. This crawler analyzes the forms and inputs fake generated data that is required by the form and submits the data. The submitted form may lead to the page requesting more data in which case the crawler proceeds with processing the form. Our crawler also collect information related to resources requested by the form including tracker and captcha scripts, service workers installed by the page and permissions requested by the page. Then, we analyze the information collected by crawler and measure the different traits that are used by the phishing sites. We believe that our measurements on the modern phishing sites would provide insights into their detection and can be used alongside existing detection systems.

1.3 Literature Review

In this section, we discuss state-of the art works related to two major parts of our research. The first part discusses works that address the abuse of web based technologies including the emerging web technologies that are included in our research and their security analysis. The second part deals with the works related to different form of Social Engineering Attacks that are focussed on this thesis. While the first part informs us of the gap in security research concerning new technologies, the second part includes work that

complement our work and therefore can be used alongside our work to provide a more efficient defense against the Social Engineering Attacks.

1.3.1 Abuse of Functionality

In this section, we discuss some of the works that analyze the abuse of functionality related to multiple web features such as Email Header, Password Reset, Web Cache, Password Managers etc. Müller et al. in the paper [29] how legitimate features such as OpenPGP and S/MIME, major standards to encrypt, and digitally sign email could be used to spoof signatures and forge emails. In this work, the authors raise concerns about the security of these protocols that are supposed to guarantee authenticity and integrity of the messages. Few other works [30], [31] raise concerns about the standards followed by websites in case of password-recovery mechanism which was introduced to help users recover their account. Further, they also identify various opportunities of attacks that could potentially lead to account takeover by attackers.

Password manager is yet another technology meant to help users keep track of their credentials and critical information that lacks sufficient security resulting in its abuse. In [32], the authors analyze the different functionalities of password managers and identify potential attacks that could lead to attackers obtaining sensitive information stored by the password managers. Another extensively used web technology is browser extensions. Although they provide a number of useful and benign features, it has been shown in a number of works [33], [34] that they are abused and used for malicious purposes. In [34], the authors identify extensions that might appear legitimate at first but is modified using updates to include malicious content.

Hu et al. in [35] analyzes the security of different browsers against IDN Homograph attacks. Internationalized Domain Names (IDN) allow people around the world to use their native languages for domain names. Since characters from different languages can look similar, IDNs have been used to impersonate popular domains for phishing. The authors find that the defenses are not sufficient and that attackers could still perform Homograph attacks. Recent work by Mirheidari et al. [36] deals with a recent abuse of web cache functionality referred to as Web Cache Deception(WCD). In this paper, they propose an attack that could gain attackers access to a user's account by crafting request URLs in such a manner that it could trick the web server into storing critical information in plain format in a cache. Based on the discussed works, it is clear that as new functionalities are introduced more effort needs to be taken to ensure its security and demote potential abuse vectors.

Emerging Web Technologies and Their Abuse

In this section, we briefly discuss works related to emerging web technologies that are dealt in this thesis such as Service Workers, Human verification systems, Multi-Factor Authentication and Browser Fingerprinting. Although these technologies are widely adopted by most of the websites, studies regarding their potential issues that could be(or are) abused by attackers are limited. We discuss each of these works and their contribution as follows.

Steiner in [37] discuss the various features of Progressive web apps and methods to detect these features in mobile’s web view. Other works related to Progressive Web Apps [38]–[40] focus on the performance of PWAs in multiple environment and their energy usage. Papadopoulos et al. [9] propose multiple attack techniques to register malicious SWs that can control a victim’s web browser(e.g., steal computation power for performing nefarious activities). There exist other studies [41], [42] that demonstrate similar attacks on Web Workers. Lee et al. [24] study the risks posed by PWAs. For instance, they collect SW scripts from top-ranked website homepages and analyze the risks for phishing due to the use of brand logos in push notifications. They also found a number of push-based phishing attacks that exploit WhatsApp and YouTube icons. Furthermore, they identify security flaws in the design of browsers and third-party push notification libraries. In a recent paper by Karami et al. [43], the authors propose two different history-sniffing attacks. Both approaches involve the user visiting the attacker’s website, which includes an *iframe* that loads content from a third-party target site. Also, the attacks assume that the target website was previously visited by the user, and that it registered a Service Worker. In [44], Chinprutthiwong et al. present an XSS attack that can be used to hijack a legitimate site’s SW. They found that the URL path of a Service Worker script can in some cases be manipulated to inject an attacker’s script into the SW code.

New, recently published studies [45], [46] discuss attacks that leverage SW features. In [45], Squarcina et al. describe a powerful DOM-based XSS attack that could be used to inject code into a script cached by a SW, through which the injected code remains persistent in the client’s machine for a longer duration compared to traditional XSS attacks. This attack is similar to cache poisoning attacks for the *Browser Cache* [47], [48], *Client Storage* [49] and *Web Cache* [50], [51] mechanisms. Unlike attacks discussed, the role of SWs in this attack is limited to serving already poisoned cache resources. However, it still highlights the need for stricter access restrictions similar to our proposed mitigation of isolating SW cache access. In [46], the authors discuss a number of security issues related to rehosted websites and describe a persistent man-in-the-middle attack implemented by leveraging SWs. Since all rehosted websites may be placed under the same origin as the attacker site that registers a SW, the SW could gain control of all resources under the rehosted origin, rendering the Same-Origin-Policy for SWs ineffective. Mitigating this attack likely requires defense measures at the rehosting level, as the attack violates a number of security policies. Soon after SWs were introduced by browsers, a few studies [52], [53] discussed the possibility of side-channel attacks that leverage SW features.

More works [54]–[58] are dedicated to measuring privacy leakage due to third-party code, such as extensions, external libraries and ad injections. In [59], the authors discuss issues related to blind trust between cross origins and explain the extent of damage it could lead to. Other works that focus on client-side web security are [60]–[62], which include policy-based access to restrict Javascript APIs(such as Performance API) to mitigate timing based side-channel attacks. Similarly, [63] presents a cost-benefit analysis to enable restrictions per site without affecting its legitimate use. In [64], Jackson and Barth argue that the concept of “fine-grained origins”(FGO) is a flawed solution to curb origin contamination. In this paper, we discussed the use of scopes as a way to isolate third-party SWs and proposed additional measures to restrict the capabilities of third-party SWs to mitigate possible origin contamination issues for SW scripts.

Other systematization of knowledge papers that consider both attacks and mitigation measures have focused on web security [65]–[67], mobile security [68], and IoT security [69]. Our work is different because we focus on attacks on Service Workers specifically, discuss existing mitigations, present a timeline of when attacks and mitigations were introduced, present open security problems, and proposed new policies that browsers could adopt to limit the damage that SW abuse could cause.

Maroofi et al. [70] demonstrate that existing anti-phishing entities are inefficient in dealing with phishing sites that include Human Verification Systems such as Captchas, Alert-Boxes and session-based evasion. In this work, they craft phishing websites that employ one of the human verification systems and submit it to different anti-phishing entities to test if their sites are detected as phishing sites. Similar experiments and results were found by authors in the paper [71]. While these papers measure the effectiveness of black-lists against tailored phishing pages with captchas, we aim to measure their use in the wild phishing sites. There exists a number of works [72]–[76] that deals with captcha solving and analyzing the efficiency of captchas to prevent crawling by bots. While these works focus on solving the captcha images, we do not aim to solve the captchas in our work. Rather, we measure the presence of known and custom captchas in Phishing pages being used as an evasion technique. In the paper [77] by Siadti et al., the authors conduct a research survey to SMS based social engineering attacks to obtain the two-factor authentication code and show that 50% of the participants had fallen victim to such scam. They also emphasize that such success rate in Social Engineering attacks is huge and attribute to the fact that people do not expect to be scammed by SMS and therefore blindly trust the messages. Few other works [10], [11], [78], [79] focus on browser fingerprinting and their roles in tracking users and blocking automated crawlers. Acharya and Vadrevu [10] demonstrate how fingerprinting techniques could be used to avoid being detected by automated crawlers. In [11], authors detect the use of browser fingerprinting for client-side cloaking by Phishing sites.

1.3.2 Social Engineering Attacks

In this section, we discuss a number of works that are related to two forms of Social Engineering attacks, Malvertising and Phishing. The works discussed include works that deal with detection, evasion and measurement of these attacks.

Malvertising

Malvertising refers to adversaries tricking users to click on advertisements that leads to suspicious/malicious sites. Advertisements are also being used as a way to lead users to phishing sites, survey scams and malware downloads. Online advertisements is a rapidly growing industry with websites and mobile users gaining more users. Currently, different work exists that study the multiple platforms that are used to deliver these malvertisements [5]–[8], [26], [80]. Here are detailed survey on these works.

A line of studies [81]–[83] is dedicated to investigate the online advertisement ecosystem. They trace the information passed between ad exchanges and analyze the revenue collected using ads. Another body of studies [84]–[92] focus on identifying *traditional* malicious web advertisements and ad campaigns

in both desktop and mobile environments. [7] develops a system called MadTracer that collects source and destination information about Ads from Alexa’s top 1 million sites and checks all URLs from a list of publisher domains and ad domains from EasyList on Google Safe Browsing and Microsoft Forefront blacklisting services to identify malicious ads. This system relies completely on the blacklists and thus unable to detect most malicious ads.

[5] develops analyzes websites that are known to use certain ad networks and identifies social engineering campaigns that are not blacklisted by known blacklisting platforms. To detect the malicious pages, it uses visual similarity to between the multiple ads that are shown that leads to multiple domains as a factor of maliciousness. On submitting their suspicions to the blacklisted services, they later found that the number of malicious sites from their dataset increased from 1% to 15%.

[26] analyzes the websites that provide free streaming services and found that most of these services are free because of their malicious use. These streaming videos pop up a lot of ads that leads the users to different malicious websites including lottery scams, drive-by-downloads, phishing etc. Moreover, they also found that these ads were aggressive in nature and can’t be easily ignored by the users.

Our paper [80] studies the delivery of malvertisements via a more persistent platform of Web Push Notifications(WPNs). WPNS are relatively new and is being misused by the ad networks to deliver malicious advertisements, although they were initially only developed for the websites to keep their users more engaged. This delivery platform is harmful compared to other known platforms because these ads can be sent to user at any point of time even when the user is not actively visiting the website. We also collect fine-grained information about these ads that can e used later for detecting them at real-time.

Preventing Malvertisements from spreading requires effort from ad networks, browser communities and ad blocking services. Although ad blockers are good at blocking ads from known ad networks, there are multiple ways to evade the said ad networks and based on my research, ads delivered through WPNs cannot be monitored by ad block extensions currently.

Phishing

Phishing is one of the attacks that has been prevalent for decades and still rising in nature despite various defense measures. This observation can be attributed mainly to the following reasons: Availability and easy access to resources to perform the attack without any advanced knowledge and Failure of real-time automated detection of the attacks. It is also one of the well-investigated domain and the attacks are categorized systematically based on their detection approach. According to [93], phishing detection approaches are based on Blacklists[94], [95], Heuristics[96], [97], Visual Similarity[22], [23], [98]–[100] and Machine Learning techniques [101]–[103]. More recently, multiple works[104]–[109] have focused on using Deep Learning to classify phishing pages with higher efficiency. A detailed survey in terms of existing works on Phishing, their approach and effectiveness is shown in Table 1.1.

The most widely used form of detecting approach in the wild is Blacklists. Therefore, few recent studies [71], [110] measure the effectiveness of these blacklists and find traditional ways of reporting to be less effective. In the paper [111], the authors measure the phishing attacks from end to end starting from email delivery to account takeover and found that few well crafted phishing sites are responsible for almost

90% of successful attacks that fooled the users. Few works [21], [112]–[114] focus on analyzing phishing kits to understand and devise better ways to defend them. Kondracki et al. in [21] devise an ML-based approach to detect Man-In-The-Middle Phishing kits by using network traffic based features. Another collection of works [28], [115] focuses on the post success of phishing attacks and analyze how the stolen information is used by the attackers. In [115], the attackers feed a set of fake credentials to phishing sites and monitor the logs to determine if they are accessed by third-party collectors besides the attacker itself.

Table 1.1: Survey of Phishing Detection and their Defenses

Detection System	Description	Detection Approach	Detection Features
[94]	A study on the effectiveness of Heuristics over Blacklisting. Results prove that Heuristics are better than blacklists in detecting phishing attempts and a large delay exists in updating the blacklists.	Blacklisting and Heuristics	HTML Page content and URL features
[97]	Introduces a rule-based filter to identify phishing emails. Performed better than GSB blacklist and SpamAssassin tool.	Heuristics	Email header and higher level analysis of URLs
[96]	Identifies Phishing pages based on the website’s ranks on Google and other information about the domain of the URL.	Heuristics	URL features and Google’s Ranking
[98]	Identifies Phishing pages that are visually similar by comparing the screenshot of the images and legitimate sites. Detection accuracy of 80% and FP rate of 18%. However, the dataset isn’t large and diverse enough.	Visual Similarity	Screen shots of the websites

[99]	Uses local and global features created from only using images to calculate signatures from phishing sites to detect Phishing pages. Local features being the target logo's image and global feature being the screenshot of the visible part of the web page.	Visual Similarity	Image of Logo and the website screenshot
[23]	Proposes creating signature based profiles of websites that could be matched with legitimate website profiles to find similarity. Visual similarity in terms of content and icon images used.	Visual Similarity	URL features, Domain Certificate and images from web page
[101]	Uses DBSCAN clustering to identify the website targeted by Phishing websites. Uses multiple features extracted from URL and the page content.	Machine Learning	URL features, page rank, content layout and text similarity
[102]	Uses Bayesian approach classify websites based on content and image similarity. Reports less than 2s as the time required to detect real time phishing cases. However, the detection involves comparing the website to limited set of 10 legitimate sites.	Machine Learning	textual and image features
[103]	Uses Gradient boosting to classify Phishing URLs. Independent of the language of page content. Higher accuracy and F1 score	Machine Learning	Contextual features on page keywords and URL features
[104]	Uses CNN-LSTM deep learning model to detect Phishing pages. Uses a larger dataset and achieves 98% accuracy and less than 1 FP rate.	Deep Learning	URL features, HTML text Content Features

[105]	Uses Deep Belief Network and analyzes traffic flows from ISP apart from URL features and website connectivity features to detect phishing pages. Based on a smaller dataset from real ISP data. 80% TPR but doesn't mention accuracy.	Deep Learning	IP Traffic flows from ISP, URL features and web page connectivity features
[107]	Uses Deep Neural Network to detect phishing sites on a Raspberry Pi device.	Deep Learning	URL features

CHAPTER 2

SoK: WORKEROUNDS - CATEGORIZING SERVICE WORKER ATTACKS AND MITIGATIONS

¹Karthika Subramani, Jordan Jueckstock, Alexandros Kapravelos, Roberto Perdisci.
Submitted to Euro Security and Privacy (S&P), 2022.

2.1 Abstract

Service Workers (SWs) are a powerful feature at the core of *Progressive Web Apps*, namely web applications that can continue to function when the user’s device is offline and that have access to device sensors and capabilities previously accessible only by native applications. During the past few years, researchers have found a number of ways in which SWs may be abused to achieve different malicious purposes. For instance, SWs may be abused to build a web-based botnet, launch DDoS attacks, or perform cryptomining; they may be hijacked to create persistent cross-site scripting (XSS) attacks; they may be leveraged in the context of side-channel attacks to compromise users’ privacy; or they may be abused for phishing or social engineering attacks using web push notifications-based malvertising.

In this paper, we reproduce and analyze known attack vectors related to SWs and explore new abuse paths that have not previously been considered. We systematize the attacks into different categories, and then analyze whether, how, and estimate when these attacks have been published and mitigated by different browser vendors. Then, we discuss a number of open SW security problems that are currently unmitigated, and propose SW behavior monitoring approaches and new browser policies that we believe should be implemented by browsers to further improve SW security. Furthermore, we implement a proof-of-concept version of several policies in the Chromium code base, and also measure the behavior of SWs used by highly popular web applications with respect to these new policies. Our measurements show that it should be feasible to implement and enforce stricter SW security policies without a significant impact on most legitimate production SWs.

2.2 Introduction

Service Workers [14] are a *powerful feature* [15] at the core of *Progressive Web Apps* [16], namely web applications that can continue to function when the user’s device is offline and that have access to device sensors and capabilities previously accessible only by native applications. In practice, a Service Worker (SW) is a JavaScript Worker [14] script with the following high-level properties: (i) it is installed by a web application rendered in a browser; (ii) after installation, the SW can act as a proxy for network requests issued by its web application, and can thus control how web content is retrieved (e.g., from a local cache or the network) and what content is eventually passed to the application; (iii) it is an event-driven process that runs in the background, even when its web application is not actively rendered on the browser, and that can be activated by the browser based on events such as receiving a web push message [116] or a request to fetch a web page on behalf of its web application, among others.

Because SWs are a powerful feature, browser developers are mindful of potential security risks that come with them. Therefore, over time browsers have implemented a number of security policies around SWs, to limit potential abuse (see Section 2.3). As an example, SW files can only be requested from a secure first-party origin (essentially, via HTTPS and from the same domain as the installing web application’s origin). However, during the past few years, researchers have found a number of ways in which SWs may still be abused to achieve different malicious purposes. For instance, SWs may be abused to build a web-

based botnet [9], launch DDoS attacks, or perform cryptomining [117]; they may be *hijacked* to create persistent cross-site scripting (XSS) attacks [44]; they may be leveraged in the context of side-channel attacks to compromise users’ privacy [43]; or they may be abused for phishing [117] or social engineering attacks using web push notifications-based malvertising [80].

In this paper, we reproduce and analyze known attack vectors related to SWs, and explore new abuse paths that have not previously been considered (Section 2.4). We first systematize this information by grouping the attacks into different categories, based on the fundamental SW security weaknesses that make the attacks possible. Afterwards, we analyze whether, how, and estimate when these attacks have been published and mitigated by different browser vendors, and organize this information into an *attacks and mitigations* timeline (see Section 2.5, Table 2.1 and Figure 2.1). Then, we discuss a number of open SW security problems that to the best of our knowledge are currently unmitigated. Accordingly, we propose SW behavior monitoring approaches and new browser policies that we believe should be implemented by browsers to further improve SW security (Section 2.6). While preventing all types of SW abuse may not be possible, we aim to propose policies that can limit the damage that potential SW attacks can make, while minimizing the impact the proposed browser changes may have on existing legitimate SW code. To demonstrate the feasibility of the proposed browser policy changes, we implement a proof-of-concept version of several policies in the Chromium code base, and also measure the behavior of SWs used by highly popular web applications with respect to these new policies (Sections 2.7 and 2.8).

In summary, we make the following main contributions:

- We reproduce previously known attacks that abuse SWs, discuss new paths for abuse that were previously not considered, and systematize these SW attacks into categories based on the fundamental features that make them possible.
- We study whether, how, and estimate when the SW attacks have been mitigated by different browser vendors, and organize this information in an *attacks and mitigations* timeline.
- We discuss open security problems related to SWs and propose new browser policies that aim to reduce the potential for future SW abuse.
- Finally, we implement a proof-of-concept version of a number of such policies in Chromium. Also, we measure how policy parameters could be tuned to limit SW abuse without significantly impacting legitimate SWs used by popular websites. To this end, we build a *SW forensic engine*, namely an instrumented Chromium browser that allows us to obtain fine-grained information on the behavior of SW code for real-world web applications.
- In addition, we disclosed the new attacks we found to browser vendors and obtained confirmation of their effectiveness. We will release our repository of reproduced and new SW attacks, SW forensic engine, proof-of-concept browser policy implementations, and measurement results to the community (the entire repository will be released after publication; the code and demos of our collection of attacks are available at [118]).

2.3 Background

In this section, we provide a brief background on Service Workers (SWs), focusing primarily on properties that are used as part of the attacks and mitigations described in later sections of this paper.

2.3.1 Service Workers

A Service Worker (SW) is a JavaScript Worker [14], namely an event-driven script that runs in the background and that does not have direct access to the DOM. To run in the background, a SW first needs to be registered by a web page. The SW code has to be contained in a JavaScript file hosted under the same origin as the origin of the web page that invokes its registration. Once installed, the SW can be programmed to cache web pages that may be later served to the user even if the browser is offline. This allows a web application running in the browser to behave more like a native application, even when the user's device connectivity to the Internet is unreliable. In addition, SWs can receive push messages and send web push notifications to the user even when the related web application is not open on the browser, in a way similar to native application's notifications.

2.3.2 SW Lifecycle

Once a website registers a SW, the SW code goes through an installation and activation phase, after which it can *control* web page requests under the website's origin. Before installation completes, the SW can import additional scripts into the worker's context by using the `importScripts` API. As such, additional code may be imported from any third-party origin. The SW is ready to use only after it is activated. Installed SWs can be updated at any point of time to a new version. Automatic checks for these updates are scheduled by the browser at an interval of 24 hours or whenever a user visits a web page that the SW controls. An update could also be triggered at any point of time by using the *Update* API. Furthermore, a SW can be explicitly de-registered by its web application.

Once installed, the SW is activated immediately, if there is no pre-existing SW installed from the same origin. Otherwise, it needs to wait for a previously installed SW to finish its execution. If required, this wait period can be skipped by using the *skipWaiting* API. Once activated, the status of the SW remains set to *running* until it is terminated by the browser. Each time an event is sent to the SW, the browser activates the SW code and signals the SW about the event.

2.3.3 SW Scope

Each SW has a *scope* that can be specified during the registration process [119]. The scope represents the URL path under which web pages are controlled by the SW¹. If no scope is specified, then by default the SW acquires the scope of the URL path under which the SW file is hosted.

¹For instance, a SW registered under origin `https://example.com` with scope `/test` has control over all web content requests under `https://example.com/test`.

Currently, a website can have only one SW registered with a given scope. However, multiple SWs can be registered under the same origin if they have different scopes. If a SW, SW_R , is registered with a scope at the root level (i.e., `scope='/'`), it will gain control over all pages of the website. However, if a second SW, SW_A , is registered with a more specific scope (e.g., `scope='/test'`), this SW is given priority over pages under its specific scope. Therefore, any requests made for web pages under this specific scope (e.g., `/test/page.html`) will be handled by SW_A and not SW_R . However, SW_A will not have access to requests made by web pages outside its scope.

Notably, a user doesn't have to visit a web page within the scope of the SW for that particular SW to be registered. For example, when the user visits a web page at the website's root level (e.g., `/index.html`), that page can register multiple SWs with different scopes.

2.3.4 Handling Network Requests

Once a SW is activated, it can listen to *fetch* events from web pages under its scope and thus intercept requests for web content. The SW can then make network requests for the requested content and cache them (using the *Cache* API). Later, when a cached resource is requested again, it can be served from the cache, which can help to reduce content load latency and enables a web application to continue working even if the device is offline. As a result, SWs gain a powerful ability that allows them to monitor users' requests and also modify the response sent back to the web page. As we will discuss in later sections, this ability could lead to SW abuse and potential leaks of sensitive information to third party sites (see Sections 2.4 and 2.6).

2.3.5 Push Notifications

A significant component of SWs is the ability to send web push notifications to users who grant permission. To use push notifications, a SW has to subscribe to a push service by using the *PushManager.subscribe* API[120]. This includes adding an *applicationServeKey* to the options. Once subscription is successful, the browser creates an endpoint URL and an *auth* secret key[121] that shouldn't be shared outside the application. These details are later used to steer push messages to the correct SW. Whenever, a push message is sent to the browser, the browser activates the corresponding SW and signals a *push* event that the SW can handle. More details about subscribing to push notifications can be found in [121].

While push messages are received in the background, SWs can also ask the browser to display a visual notification (typically in response to a push message) to the user. To this end, SWs can call *showNotification* to display a message on the user interface. Notice that while push messages and notifications are typically used together, SWs may choose not to call *showNotification* in response to a push message being received (in some browsers, this will trigger a default notification message issued by the browser itself). Similarly, a SW can call *showNotification* independently from receiving a push message.

To send notifications to the user, a SW needs to request a one-time explicit user consent (usually during the SW registration phase), by invoking *Notification.requestPermission()*. However, in addition to the user granting permission to the SW via the browser UI, the browser itself can only display visual

notifications to the user if OS-level permission is granted. Different OSes have their own policies regarding how applications (including the browser) can obtain such permission. As an example, in case of MacOS the permission is disabled by default, unless the user specifically grants the permission (for instance, at the end of the browser software installation process). On the contrary, Windows grant such permission by default.

2.3.6 Periodic Background Sync

The *Periodic Background Sync*[122] API allows web applications to configure their SWs so to make updates in the background at a periodic time interval. It can be used to trigger `periodicsync` events from the SW without any event being received from a remote server. Effectively, this feature allows a web application to keep its SW and cached content up to date. This API is currently supported by Chrome and other Chromium-based browsers, such as Edge and Opera. However, given its ability to operate in the background, the *Periodic Background Sync* poses a potential security threat that has refrained other browsers, such as Firefox and Safari, from implementing it [17]. To curb its possible abuse, browsers need to enforce a number of restrictions on the API use [123]–[125].

2.3.7 Security Policies

In general, browsers enforce a number of default security policies to limit potential SW abuse. For instance [126], [127]:

1. Only secure origins (HTTPS sites) can register SWs.
2. The JavaScript file containing SW code must be hosted under the same origin as the website that registers the SW.
3. A SW should be terminated if the SW code has been idle for more than 30 seconds or if an event takes more than 5 minutes to process.
4. Push notifications should trigger a user-visible notification if the SW does not explicitly issue one.
5. The use of some APIs (e.g., *Periodic Background Sync*) should be restricted by permissions that must be granted by the browser (not necessarily via a direct UI request to the user[123]).

Unfortunately, not all browsers implement all policies and, when implemented, differences exist among browser vendors. In the rest of the paper, we discuss both previously known and new ways in which an attacker could still abuse SWs to achieve malicious goals despite the SW constraints listed above.

2.4 Service Worker Abuse

In this section, we describe and categorize a number of attacks that can be launched by abusing Service Workers (SWs) in different ways. We group the attacks into categories based on the root SW features

Table 2.1: Overview of attacks and impacted browser versions. Legend: (●) first attack impact; (○) fix released; (◐) partial fix released; (⊠) no fix released yet; (☆) attack not possible.

SW Abuse Categories	Attacks	Abuse Vectors								Browsers				
		Push API	Notification API	Sync API	Performance API	Update API	ImportScripts	iframe inclusion	3rd-party code	Chrome	Firefox	Edge	Safari	Opera
Continuous Execution	WebBot [9]			✓		✓				● v69.0 ○ v70.0	● v57.0 ◐ v60.0	- ○ v80.0	☆	● v56.0 ○ v57.0
	PushExe [117]	✓								☆	● v59.0 ⊠	☆	☆	☆
	StealthierPushExe [New]	✓	✓							● v85.0 ⊠	☆	● v.85.0 ⊠	☆	● v71.0 ⊠
Side-Channels	OfflineOnload [117]							✓		☆	● v59.0 ⊠	☆	● VII.1 ⊠	☆
	PerformanceTiming1 [43]				✓			✓		● v79.0 ○ v83.0	● v72.0 ◐ v73.0	● v79.0 ○ v83.0	● v12.1 ◐ v14.0	● v66.0 ○ v69.0
	PerformanceTiming2 [43]				✓			✓		● v79.0 ⊠	● v72.0 ⊠	● v79.0 ⊠	● v12.1 ⊠	● v66.0 ⊠
Hijacking	XSS [44]						✓		✓	⊠	⊠	⊠	⊠	⊠
	ExtensionHijack [New]								✓	☆	● v82.0 ⊠	☆	☆	☆
	LibraryHijack [New]						✓		✓	⊠	⊠	⊠	⊠	⊠

that make them possible. For most of the attacks we discuss, we (re-)produce our own proof-of-concept implementations, which we tested on a large number of browser versions from five major browser vendors and have shared them publicly [118] (except for two new attacks that we disclosed to vendors but are not yet mitigated; see Section 2.4.5). A summary of the attacks we consider is provided in Table 2.1, which includes a reference to relevant previous publications or online resources in which an attack was described. To the best of our knowledge, some of the attack variants we discuss were not previously considered and are thus marked as “New.” In addition, Table 2.1 provides detailed information on different browser features or APIs that are exploited for each attack and information about what browser versions were first affected and what version provided a fix, if any. In this section we focus on categorizing the attacks, whereas browser mitigations are discussed in Section 2.5.

2.4.1 Continuous Execution

Because SWs can run in the background, can issue network requests, and *can be activated at any time* (even when their related website origin is closed), they can be abused to *stealthily* run unwanted or malicious code. For instance, SWs could be abused by a malicious website to run *cryptomining* code [117] or to build a *web-based botnet* [9]. Such types of attacks are generally enabled by artificially prolonging the amount of execution time granted by the browser to SW code running in the background, thus (approximately) achieving *continuous execution*.

WebBot: Papadopoulos et al. [9] describe how to build a SW-based botnet. If a victim visits a malicious website M , this website can register a SW, S_M , which can run in the background. When executed, S_M could implement code that (i) reaches out to a command-and-control (C&C) website to receive commands and (ii) execute the received command to perform actions such as participating in DDoS attacks, *distributed password cracking*, function as a *relay proxy*, etc.

For the botnet to function properly, S_M needs to be periodically (frequently) activated. Papadopoulos et al. [9] mention that this would be possible by using the *BackgroundSync* API [128]. Based on this information alone, we were initially unable to fully reproduce the attack. However, we found an online discussion about the attack by Chromium developers [125], which stated that the attack was possible primarily due to a bug that allowed the *Update* API (see Section 2.3) to be invoked from a SW's *activate* listener. By combining the information provided in [9] and [125], we were able to reproduce the attack as explained below. For the attack to work, the following components are required:

- SW script whose content keeps changing at server side to appear fresher than the SW already registered.
- Leveraging the *Update* API, which checks if there are any changes made to the SW file and fetches the updated version of the SW script from the server.
- Leveraging the *BackgroundSync* API to activate the SW every time the browser is re-opened.

```
function wait(ms) {
  const tmp = setInterval(() => { /* do bad */ }, 100);
  return new Promise(res => setTimeout(res, ms));
}
self.addEventListener('activate', event => {
  self.registration.sync.register('foo');
  // Wait < 30s
  event.waitUntil(wait(25000).then(() => {
    self.registration.update(); }));
});
```

Listing 2.1: Example of SW self-update on *activate*

As shown in Listing 2.1, as the SW is activated it registers a *BackgroundSync* and then calls the *Update* API after a predefined timeout. In general, the SW execution is supposed to terminate after a fixed period of time (a few minutes). Since the SW file in the server keeps changing, calling the *update* method will fetch the newer version of the SW script. This action is followed by the browser raising the *activate* event, which causes the corresponding listener function in the SW code to be executed, where malicious code can be invoked. This cycle repeats, until the browser is closed (or the SW is explicitly unregistered). When the browser is re-opened, the *BackgroundSync* triggers a sync event and the SW will be activated again restarting the execution cycle.

PushExe: Lee et al. [117] demonstrated that if an attacker was successful in registering a SW and obtaining push notification permission from the user, she could then leverage the *Push* API to activate the SW code at any moment. In some browsers, the attack could be rendered stealthy if the SW code does not explicitly

invoke the *showNotification* API when a push notification is received. Using this approach, the authors were able to keep the SW running continuously in the background for long periods of time, for instance to perform *cryptomining*.

This attack was found to work in Firefox, Edge, and the UC Browser [117], though the attack is not stealthy in Chrome because the browser displays a default notification message for every push event, which may alert the user about the presence of a malicious SW running in the background.

By independently reproducing and testing this attack, we verified that in Firefox and Edge the browser revokes the push subscription of a SW (i.e., the SW cannot receive new push events), if the SW fails to show a notification after receiving a push message for 15 and 3 times respectively, thus blocking the attack, as also mentioned in [117]. However, the attack could still be made continuously stealthy in Firefox (whereas Edge does not appear to be affected) by simply renewing the SW registration in the background, after a few push messages are received (i.e., before exceeding the browser’s limit for “silent” push events), as shown in Listing A.1 (in Appendix).

```
async function DisplayHideNotifications()
{self.registration.showNotification().then(
  async ()=>{
    const notifications = await self.registration.getNotifications()
    ;
    for(let i = 0; i < notifications.length; i++) {
      notifications[i].close();
    }
  })}
self.addEventListener('push', async function(event){
  event.waitUntil( DisplayHideNotifications());
});
```

Listing 2.2: Immediately hiding notifications on push events

[New] StealthierPushExe: While working to attain SW Continuous Execution, we discovered a variant of **PushExe** that can overcome the limitation of default notifications being displayed to the user, which would then prevent potentially alarming the user of suspicious activity. For instance, Chrome’s implementation requires that a call to *showNotification* be explicitly invoked by a SW upon receiving a push message. If *showNotification* is not explicitly called, a default notification stating ‘*This site was updated in the background.*’ will be displayed. To avoid this, we found it was possible to first invoke *showNotification* and then immediately invoke a function that parses through all notifications displayed by the browser for the SW’s origin (using *getNotifications*) and closes them using *Notification.close*, as shown in Listing 2.2 (notice that the code can be modified to selectively close only the attack-related notifications, if desired). As a result, at every push event the SW could display a notification and then immediately close it, making it unnoticeable by the user. This would allow for frequently activating the SW in a stealthier way (i.e., with no visible UI signal), making it possible to achieve stealthy continuous execution.

We verified that this new attack work on both desktop (Windows 10) and mobile (Android 11) devices, and have disclosed it to the Chrome developers (see Section 2.4.5).

2.4.2 Side-Channels

This category of abuse includes attacks that allow unauthorized parties to leverage SWs to gain sensitive information by bypassing browser isolation.

OfflineOnload: In [117], Lee et al. propose a history-sniffing attack that works as follows. A user first visits the attacker’s website, which registers a SW. At a later time, if the user again opens the attacker’s website in *offline mode*, the SW will intercept the request and return a page that includes a number of `iframe`s whose URL points to third-party target sites. The attacker’s goal is to determine if the user previously visited those sites. Lee et al. found that in some browser versions, such as Firefox 59.0.2 and Safari 11.1, if the browser is in offline mode, the top page (i.e., the attacker’s page) is sent an *onload* event related to an embedded `iframe` only if the target site had already been visited by the user and a corresponding SW (with offline support) had been registered. Therefore, the attackers can register an *onload* event handler to sense if a third-party site embedded in an `iframe` was previously visited by the user.

PerformanceTiming: In a recent paper by Karami et al. [43], the authors propose two different history-sniffing attacks. Both approaches involve the user visiting the attacker’s website, which includes an *iframe* that loads content from a third-party target site. Also, the attacks assume that the target website was previously visited by the user, and that it registered a SW. Furthermore, the *iframe*’s source URL must fall within the scope of the targeted website’s SW.

The first attack (**PerformanceTiming1**) identifies the presence of a previously registered third-party SW by monitoring two attributes of *PerformanceResourceTiming* API, namely *workerStart* and *nextHopProtocol*. The values of these attributes change depending on whether the resource is being loaded when the target page request is served via a SW, compared to when no SW is yet registered, and can thus be used to infer whether the page was previously visited by the user. While working to reproduce this attack, we additionally found that in Firefox there exists another property of *PerformanceResourceTiming*, called *initiator*, that can also be used to identify the presence of a SW in a similar way.

The second attack (**PerformanceTiming2**) is a timing-based side-channel attack that measures the loading time for the requested `iframe` resource on the user’s machine, which can be compared to a pre-calculated loading time of the resource without the presence of a SW. Because SWs often cache resources to optimize performance and enable offline browsing, the difference in the loading times can help determine the presence of a SW [43].

2.4.3 SW Hijacking

We now discuss attacks that involve *hijacking* SW functionalities, by either injecting malicious code into a legitimate SW or by injecting a malicious SW into a benign origin.

XSS: In [44], Chinprutthiwong et al. present an XSS attack that can be used to hijack a legitimate site’s SW. They found that the URL path of a SW script can in some cases be manipulated to inject an attacker’s script into the SW code. This is possible because some websites use dynamic URL query parameters in the SW’s URL path that depend on the *window.location* API. The authors demonstrate that the attacker could modify the URL parameters by tricking the users to visit a carefully crafted target URL. Although

the user ends up visiting the legitimate target website, failure to validate the URL parameters could result in the injection of attacker-controlled code into the SW context during the registration of a legitimate SW. Such an attack is stealthy in that it would go r by the user or the targeted website.

[New] ExtensionHijack: We discovered another possible approach to hijack a legitimate website’s SW. Specifically, we found that browser extensions can be used to inject malicious SW code in the scope of any benign origin.

Specifically, Firefox is vulnerable to SW hijacking by extensions because they are allowed to use the *FilterResponse* API, which enables them to modify the request made to fetch a SW script file during its registration phase. This API is unique to Firefox and we leverage it to demonstrate this new attack.

To this end, we developed a basic Firefox extension that has the capability to intercept requests using *WebRequest* and *WebRequestBlocking* permissions, which are commonly used by popular extensions, including ad blockers. Next, we need to filter requests made for obtaining the SW script. To achieve this, our extension uses the *OnBeforeSendHeaders* API to intercept requests and obtain their HTTP headers. We identify SW script requests, as well as scripts that are imported by the SW, by looking for the header parameter *name* with value *Service-Worker*. Once a request for SW code is identified, we read the SW file’s content using the API *FilterResponse*. Before sending the file’s data, we can insert at the beginning of the file a malicious code snippet (although in our proof-of-concept extension we inject harmless code) as shown in Listing A.2 (in Appendix).

The advantage of this attack is that the extension itself does not explicitly execute malicious code. Rather, the extension uses allowed APIs to inject additional code to be executed in the context of a SW. This may make it more difficult for extension stores to classify the extension itself as malicious in the first place. Additionally, because malicious extensions often go unnoticed for long periods of time [55], the impact of this attack may be significant. Even if the extension is detected as malicious and removed from the browser after installation, it may be too late, as the extension may have already injected malicious SW code under many highly popular website origins, which will continue to execute on a potentially large number of browsers even after the extension is removed from the store and the browser, until the SW code is updated.

[New] LibraryHijack: Website owners can leverage third-party libraries from “push providers” (e.g., OneSignal.com, SendPulse.com, iZooto.com, etc.) to conveniently enable and manage push notification campaigns. Typically, this entails including third-party code to run within the SW of a website, *W*. As a result, the provider of the third-party code gains complete access to *W*’s SW, whose capabilities go much beyond providing push notifications. For example, the SW script could be modified to intercept all fetch requests and inject new page content that may harvest sensitive user information and relay it back to an unauthorized server. Currently, there are no restrictions posed on functionalities of third-party SW libraries, and in Section 2.7 we discuss our findings on a third-party library that indeed seems to misuse imported push service code to track all web pages visited by the user on *W*.

2.4.4 Other Attacks

SWs can also be abused to launch social engineering attacks by presenting misleading or malicious web push notification (WPN) messages to users. In both the attacks described below, a website must first register a SW and then obtain permission from the user to send notifications. Unfortunately, social engineering attacks are difficult to mitigate directly through browser policies and restrictions on SWs. Rather, such attacks typically require content analysis, such as an analysis of the information displayed by WPN, what sites they lead to when clicked on, etc. [80]. Therefore, we discuss these attacks only briefly below and do not include them in Table 2.1. Other web attacks that are partly related to using SWs are also discussed in Section 2.9.

Phishing: Lee et al. [117] discussed the possibility of launching phishing attacks via WPNs. For instance, a malicious SW could issue a notification that displays the Chrome icon and a message such as “Google Chrome Premium,” and a “DOWNLOAD” button, which when clicked on could lead the user to installing malicious code. Furthermore, the authors discuss how in some cases an attacker could extract the *PushSubscription* object from network traffic [117], and then use it to spoof push messages as arriving from a legitimate domain.

Malvertising: Although web push notifications (WPNs) were initially meant to be used to send first-party messages to users to keep them engaged with a website’s own content, WPNs have since become an alluring platform for advertisers to reach users even when a given publisher website is not being visited. For instance, ad networks such as VWO Engage (formerly PushCrew), Roost, PushAd, etc., provide software that allows web developers to easily include WPN-based ads to their websites. To this end, web developers typically include third-party SW code provided by these companies to their websites. Besides potentially exposing a website’s SW to the LibraryHijack attack described earlier, the website may also be responsible for exposing users to malicious ads via their WPNs, as reported in [80].

2.4.5 Ethical Considerations and Disclosure

All attacks were tested using our own test websites and lab client machines. No real user or production website was affected by our tests.

We disclosed our findings to affected browser vendors. First, on April 23rd, 2021, we reported the StealthierPushExe attack (see Section 2.4.1) to the Chrome, Opera and Edge developers. After about one month, we received confirmation that the attack affects Chromium-based browsers (i.e., including Chrome, Opera, Edge, and likely several other less popular browsers). The Chromium developers are currently discussing (in a private online forum) possible fixes. Some of the steps being discussed for patching the issue follow an approach similar to some of the recommendations we propose in this paper for restricting SW execution (notice that we developed our proposed mitigation ahead of disclosing the attack to the Chromium team). To the best of our knowledge (at the time of writing), the StealthierPushExe attack has not yet been mitigated. Therefore, our public release of the collection of SW attacks [118] discussed in this paper excludes the StealthierPushExe attack. However, reviewers can access the entire collection of proof-of-concept attacks at [129].

Furthermore, we also disclosed the Extension Hijack attack (see Section 2.4.3) to the Firefox developers in June 2nd, 2021, who confirmed that Firefox is indeed still vulnerable to this attack. The developers are still discussing removing extensions’ access to network requests related to SW code and its import scripts. As for the Library Hijack attack, we have not disclosed it to browser vendors because this attack relates to how service workers are often misunderstood and misused by web developers, rather than being a browser vulnerability per se. However, we plan to inform browser vendors of this attack and the mitigation proposed in our paper once it is accepted for publication.

2.5 Existing Mitigations

In this section, we discuss existing mitigations to some of the attacks discussed in Section 2.4. As mentioned earlier, we have (re-)produced a proof-of-concept version of the attacks. We then tested the attacks using multiple versions of five different browsers from different popular vendors, namely Chrome, Firefox, Edge, Safari and Opera. To make testing with multiple combinations of browser version and operating systems easier, we made use of `BrowserStack.com`. Our purpose was to estimate *when* (at what browser version) an attack was fixed, whether it was *fully or partially mitigated*, or if the attack is *still feasible* in some or all browsers.

Table 2.1 provides an overview of browser versions that are vulnerable to the attacks, and what browser version (if any) fixed or mitigated the vulnerability. Furthermore, Figure 2.1 visualizes an approximate timeline of when the attack was made public and when a mitigation for the attack was released (if any). Overall, we found that some of the attacks have been mitigated by some browsers, but also that most of the attacks are still possible on at least some of the latest browser versions. Furthermore, some of the attacks introduced in Section 2.4 have not yet been considered for mitigation. Notice also from Figure 2.1 that some attacks appear to have been mitigated by some browsers before the attack was officially published, perhaps as a result of responsible disclosure processes. Below we discuss what mitigations have already been implemented or planned so far by browsers, whereas in Section 2.6 we discuss open problems and propose new mitigations.

2.5.1 Mitigating Continuous Execution

The continuous execution attacks described in Section 2.4.1 all require the victim to visit a malicious website that will register an attacker-controlled SW on the victim’s browser. By design, to avoid overwhelming the user with UI alerts for SW registration requests, the action of registering a SW is silently allowed by default and the user is not informed by the browser that the site she visited registered a SW. The mitigations discussed below still assume that SWs can be silently registered, but have the effect of limiting the number of users granting notification permissions, limiting the frequency with which a service worker is activated, or making SW execution due to web push events less stealthy.

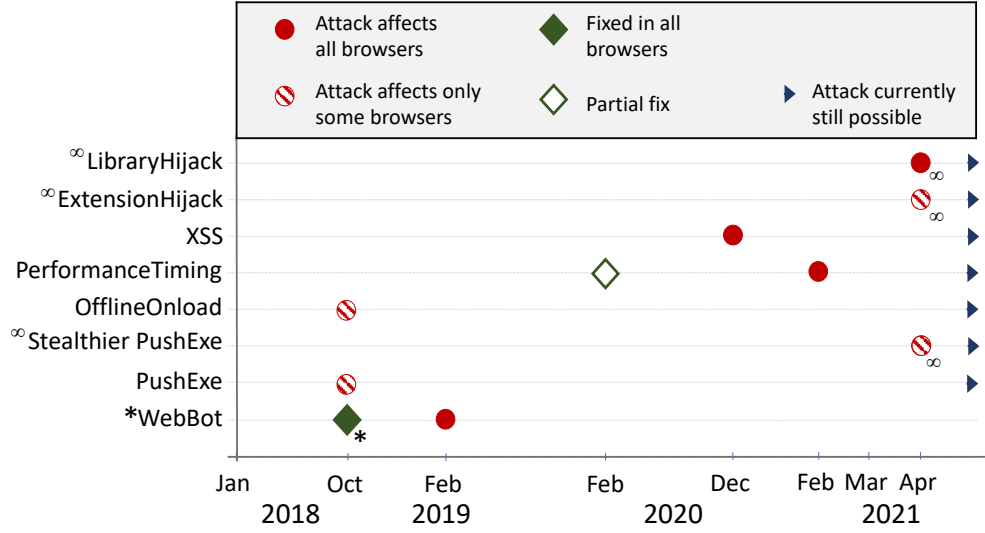


Figure 2.1: Approximate timeline of attacks publication and mitigations. The ∞ symbol denotes new attacks discovered in this paper, which are not yet mitigated. Notice that some mitigations were implemented before official attack publication, perhaps thanks to responsible disclosure (the “*” next to WebBot denotes that a mitigation was described in online documentation related to Firefox, but our own tests show the attack appears to still be possible on that browser).

Termination Delay Limits. This existing mitigation has the effect of fully preventing the **WebBot** attack.

As explained in Section 2.4.1, the *WebBot* attack exploited a self-update behavior by continuously invoking the *Update* API, causing the SW to self-update and continue executing malicious code. To defend against this vulnerability, major browsers such as Chrome, Edge and Opera have implemented a limit of up to three minutes on the SW termination timeout (e.g., `kMaxSelfUpdateDelay` is set to three minutes in the Chromium source code), when *update* is invoked while handling an *activate* or *install* event [125]. In case of Firefox, according to an online bug report [130] a fix was implemented in v70.0. However, while testing our reproduced attack code we were able to keep the SW running for hours. On the other hand, Safari terminates the SW as soon as the related website is closed and is therefore not affected by this attack. According to [125], this fix was implemented before the **WebBot** attack was officially published in [9], as reflected in Figure 2.1.

Notification UI Changes. Bilogrevic et al. [131] recently showed that although 74% of all browser permission prompts that users receive are about notification permissions, only 10% of these requests are granted by users on desktop devices (21% on mobile devices). Because notification permission requests cause unwanted interruptions during normal user browsing, a new and more quiet notification prompt has been introduced in Chrome (starting from version 80) and in Firefox [131]–[133]. In Chrome, the quiet notification permission prompt is shown within the URL bar when either of two conditions are

met: (i) the website requesting notification permission has a high average deny rate across its visitors, or (ii) the user recently denied notifications multiple times (e.g., 3 consecutive time) on different websites within a given timeframe (e.g., 28 days).

While we did not find indications that the quiet notification implemented by Chrome was intended to mitigate specific types of SW abuse, it may have some mitigating effect on the **PushExe** and **StealthierPushExe** attacks described in Section 2.4.1. Because these attacks require victims to grant notification permission to the attacker’s website, the Chrome UI change may cause the attacker’s website to be selected for the quiet notification prompt, potentially decreasing the number of users that will actually grant the permission, thus mitigating the attacks. However, as it is not a direct mitigation to those two attacks, we do not consider the quiet notifications UI as a fix for the purpose of Table 2.1. Furthermore, in Section 2.6 we also discuss how this mitigation may be easily circumvented by the attacker.

Default Notifications. We have confirmed that, at the time of writing, the *PushExe* attack described in Section 2.4.1 still works in the latest versions of Firefox (v91.0). In Chrome, the *PushExe* is not stealthy, because a default notification message² is shown after Chrome detects that no notification is explicitly shown by the SW. However, we verified that the *StealthierPushExe* attack that we introduced in Section 2.4.1 remains unmitigated in the latest versions of Chrome (v93.0), Edge (v93.0) and Opera (v78.0).

2.5.2 Mitigating Side-Channels

Event Signaling. To mitigate the **OfflineOnload** attack mentioned in Section 2.4.2, Chrome (at least since v50) ensures that the `iframe onload` event is triggered regardless of the presence of a SW. However, we were able to verify that this attack is still possible even for the latest version of Firefox (v91.0) and Safari (v14.0).

Site Isolation. The **PerformanceTiming1** attack summarized in Section 2.4.2 can be mitigated by making sure that meta-data related to a given origin is not revealed to third-party `iframes`. This has been fixed in Chrome since version 83.0. However, as mentioned in Section 2.4, we found that a variant of this attack appears to be still possible in the latest Firefox browser (v91.0) by monitoring the *initiator* property from a third-party `iframe`. Also, our tests with reproduced attack code for the **PerformanceTiming2** attack confirmed that it still remains unmitigated in the latest versions of all major browsers (see Table 2.1).

2.5.3 Mitigating Other Attacks

We are not aware of specific mitigations that have already been implemented by affected browsers to counter hijacking attacks (Section 2.4.3) or social engineering attacks (Section 2.4.4). We will discuss open problems and potential new mitigations to some of these attacks in Section 2.6.

²Message: “The site has been updated in the background.”

2.6 Open Problems and New Mitigations

In this section, we revisit some of the attacks presented in Section 2.4 and highlight open problems that, to the best of our knowledge, have not yet been addressed by browsers. Furthermore, we also propose new mitigations that we believe should be implemented in future browser versions to address the problems we identified.

2.6.1 Limiting SW Execution

Open Problem: In Section 2.4.1, we discussed different ways (both previously known as well as new ones) to (silently) extend the execution time of SWs, to approximately achieve continuous execution. Although some mitigations specific to the attacks in Section 2.4.1 have been employed by some browsers, it may still be possible to create similar attack conditions that exploit existing or future SW features. For instance, to circumvent existing mitigations related to always showing notification messages to users every time a push event occurs (see Section 2.5.1), the SW code could be activated only at a time when the user may not be paying attention to the screen (e.g., many users leave the browser always open, even at night), as also discussed in [117]. Furthermore, even if the SW is activated a large number of times in a row using many consecutive push messages, the SW can prevent the browser from showing multiple notifications from the same website, which may make the user suspicious. To make sure that the user will only see one single notification, the SW can keep reusing the same tag parameter value, as shown in Listing A.3 in Appendix. Fundamentally, we found that browsers do not currently put any constraints on the number of push messages a SW can receive or on the amount of execution time granted to any given SW. This leaves open possible abuse paths, as exemplified above.

Proposed Mitigation: To defend against present and future *continuous execution* attacks, we need a more generalized defense that can dynamically monitor the SW execution time and throttle it when abuse is suspected. This can be accomplished with additional browser policies. Specifically:

1. Monitor and limit the overall background execution time for which a SW runs every time it is activated. This will prevent known and unknown ways of artificially elongating the time a SW remains active (e.g., this would mitigate abuse vectors similar to the self-update exploit used in the **WebBot** attack).
2. Limit the number of push events received by a SW within a predefined time window. This would have the effect of limiting the frequency with which a SW can be remotely activated, thus throttling continuous execution attacks.
3. Ensure that a SW notification displayed to the user remains visible until the user interacts with it (e.g., by clicking on it or closing it explicitly). This would help to mitigate stealthy activations via push events.

4. Limit the volume of third-party network requests issued in the background by a SW. While this is not strictly a limitation on execution time, it can be useful to mitigate possible “bursty” bandwidth-exhaustion DDoS attacks (e.g., by issuing many background network requests in a short execution time) against third-party websites.

In Section 2.8, we discuss how we implemented a proof-of-concept version of some of these policies in Chromium. In Section 2.7, we measure how SWs are currently used by popular websites and propose concrete thresholds to limit SW execution with limited or no impact on legitimate SW functionalities.

2.6.2 Limiting Malicious SW Permissions

Because by design SWs can be silently registered by any website, preventing the registration of an arbitrary SW may not be possible³. However, notice that without being granted the notification permission the SW cannot receive push messages and the attacker is unable to launch effective *continuous execution* attacks (Section 2.4.1) or *social engineering* attacks (Section 2.4.4), thus limiting the damage that a malicious SW may cause. The new quiet notification permission requests described in [132] (see also Section 2.5.1) could therefore be seen as a way of greatly restricting the damage a malicious SW can do. The reason is that, presumably, only few users would grant notification permission to an untrusted website (notice also that the permission grant rate is already low in general for most websites [131]). Thus, it is likely that a malicious site that asks its visitors for notification permission would rapidly meet the criteria to qualify for the quiet notification UI. In turn, this may have the effect of further reducing the number of users who grant permission and whose browser can be meaningfully abused by the malicious SW.

Open Problem: Unfortunately, the quiet notifications UI is not in itself an effective mitigation for limiting the number of victims that may grant notification permission to a malicious SW. One reason for this is that malicious SWs can leverage the same *double permission* prompt pattern [135], [136] that is recommended as a good practice to legitimate web developers. The *double permission* prompt consists in asking the user twice whether they would like to receive notifications from a website. The first time, the website uses JavaScript code [135], [136] to create a notification permission dialog box within the page’s context (see Figure A.1 in Appendix). Only if the user confirms, typically by clicking on a custom “Yes” or “Sign up” button, the SW will go ahead and request the actual notification permission through the browser UI. The reason why legitimate websites often use this pattern is because they want to avoid being blocked from asking the user for notification permission again in the near future. Since the website controls the JavaScript dialog box, the browser will not be aware that the user may want to block notifications from this site, and therefore the website gets to ask again every time the user visits it. The net effect of this legitimate (and recommended) web development pattern is that the browser may grossly overestimate the notifications allow-rate for a given website. Intuitively, it is highly likely that users who do not want to receive notifications from a website will click on the “Not now” or “Dismiss” (or equivalent) button

³Obviously, blocking a known malicious website, for instance by using URL blocklists, would also prevent any related SW to be registered. Unfortunately, threat feeds and blacklists often have gaps and may not block a malicious site for a prolonged time [134], during which many victims could visit the SW and have a SW installed.

on the JavaScript dialog box and they will not be presented with the real SW's request for notification through the browser. Ultimately, given that one of the criteria to enable quiet notifications UI is a higher denial rate, malicious sites can evade this by simply adopting the double permission pattern. In general, the very recommendation to legitimate web developers on adopting the *double permission* prompt may make the newly introduced quiet notification UI much less effective than anticipated.

Another issue is due to the fact that the attacker's site could also launch social engineering attacks similar to the ones mentioned in [5] to encourage the user to explicitly click on the quiet UI's icon and explicitly grant permission.

Proposed Mitigation: Unfortunately, preventing users from granting notification permission to a malicious SW may be difficult, as discussed above. In addition, once a SW is registered and has been granted permission, the SW will persist until the user explicitly removes them following a cumbersome process that involves going through the browser preferences and settings. If little or no constraints are imposed, this may lead to significant abuse, as described in Section 2.4.

As a mitigation, we argue that the browser should monitor each SW's behavior for signs of abuse. The browser could then explicitly offer the user to de-register a SW (with a specific UI dialog box) when anomalous behavior is detected, or it could automatically de-register the SW.

For instance, consider the following scenario:

- The user visits a malicious website once, at which point a SW is registered and notification permission is requested.
- Assume that the user grants notification permission at first visit (e.g., due to a social engineering attack), and that the user never visits the site again.
- Afterwards, the SW runs frequently in the background (e.g., due to frequent push events) to achieve continuous execution using one of the approaches described earlier.

In this example scenario, the browser could detect that the SW is violating one or more of the policies we proposed in Section 2.6.1, which will automatically limit SW execution. At the same time, the browser could detect that the user has not visited the site again since the first time the SW was registered, or that the site has a very low engagement score as defined by Chromium[137]. In this case, the browser could inform the user and ask whether she would like to de-register the SW. To make the decision easier, the browser could let the user know that the SW has been running anomalously and frequently, and that the user has visited the website only once (or very rarely). As an alternative, if the SW is not explicitly de-registered by the user and the browser continues to observe that the SW abuses execution limit policies, it may simply de-register the SW outright (notice that the SW could always be re-registered next time the user visits the same site, if the user so desires).

2.6.3 Restricting Third-Party Code Inclusions

Open Problem: It is well known that third-party JavaScript code inclusions come with security risks [138]. As discussed in Section 2.4.3, the common practice of including third-party code into SWs could lead

to *hijacking* attacks. Content Security Policies [126] (CSPs) can be used to defend against SW *hijacking attacks* such as **XSS** or **LibraryHijack**, for instance by using the `script-src` to restrict imported code into a SW to be loaded only from authorized domains.

However, implementing this defense is up to web developers, and in Section 2.7 we show that only a small fraction of websites express SW-specific CSP restrictions (also, low CSP adoption is a known issue in general [139]). Unfortunately, when CSP policies are missing, the browser poses no restrictions to importing third-party code into a SW.

Proposed Mitigation: We argue that, given the potential for abuse related to SWs, the browser should follow the *fail-safe defaults* principle and deny the ability to import third-party code by default. Namely, the browser should always assume a default `script-src`: ‘self’ policy for SWs. The web developer could then express exceptions to this default policy by explicitly listing authorized third-party origins in the `script-src` directive (this CSP directive would need to be sent to the browser with every SW file response, which can be easily configured in modern web servers). In Section 2.7, we will also show that the number of different origins that would need to be authorized in current production SWs is very small (only one or two, if any).

Unfortunately, `script-src`: ‘self’ does not prevent `eval()` to be used in SW code [140], leaving a door open to potential code hijacking attacks such as variants of the **XSS** attack proposed in [44]. Instead, the use of `eval()` should be disabled by default and enabled explicitly by adding the directive `script-src`: ‘unsafe-eval’, as for page JavaScript code. Notice also that the `worker-src` CSP directive [141] can be used to restrict what URLs may be used to load a SW file, but does not apply to the *importScripts* API. Furthermore, `worker-src` does not have any effect on blocking the use of `eval()` either.

2.6.4 Restricting the Scope of Third-Party Libraries

Open Problem: In some cases, web developers may want to explicitly allow third-party services, such as *push services*, to include code into their SWs. For instance, assume that website W wants to make use of push service P (e.g., OneSignal.com, iZooto.com, etc.). In this case, W would want to import P ’s third-party code into its SW, S_W (notice that P ’s origin can be easily specified in S_W ’s `script-src` CSP directives, as discussed earlier). Unfortunately, once P ’s code is imported in the context of S_W , there is no way to restrict what APIs P ’s code can use, thus potentially enabling a **LibraryHijack** attack (see Section 2.4.3).

To attempt to isolate their third-party code from the first-party website’s S_W , web developers could register a separate SW, S_P , with a different *scope*, instead of running the SW under the root path of W . This would allow S_P to coexist with other SWs registered under W . In addition, S_P would not be able to intercept network requests related to content outside of its scope, thus effectively isolating the third-party SW code. However, while this would be an improvement, it does not prevent S_P from being able to directly accessing W ’s cookie store [142]. Furthermore, S_P would also be able to access the cache [143] and thus any content previously stored by S_W , since there is currently no cache isolation for SWs registered

with different scopes under the same origin. Consequently, the third-party code could still potentially access highly sensitive information related to W .

```
if ('serviceWorker' in navigator) {  
  // proposed register() options to only enable use of push  
  notifications  
  // while prevent the use of other sensitive APIs  
  // such as cookie store, cache, fetch events, etc.  
  navigator.serviceWorker.register('/pushservice_sw.js',  
    {scope: './pushservice_sw_scope/',  
     capabilities: 'push', 'notifications'})  
}
```

Listing 2.3: Proposed change to register SW with limited capabilities.

Proposed Mitigation: To mitigate the risk of **LibraryHijack** attacks, a possible approach is to explicitly limit the *capabilities* that a given SW script can have. This list of capabilities could be expressed at registration time. At the moment, when registering a SW under a given origin (via `ServiceWorkerContainer.register()` [119]), only the *scope* of the service worker can be limited. However, we argue that the *options* parameter should be extended to allow expressing additional constraints. For instance, we could express what set of functionalities or APIs the SW is allowed to access, or what set of events it is allowed to listen to. This way, we could restrict the capabilities of a third-party SW (i.e., a SW that imports third-party code) to using the *push* and *notifications* APIs while denying the use of the cookie store, the cache, or the fetch API, as in the example code in Listing 2.3. On the other, first-party SW scripts could be registered without capability restrictions so that they can use any functionality made available to SWs by the browser.

More fine-grained changes would also be useful, such as expressing whether the SW is allowed to access the cache but at the same time indicate whether the cache for this SW should be isolated by scope (notice that this proposed cache isolation mechanism could be implemented in a way similar to the cache isolation approach used for the now-deprecated *AppCache* API [144]). These browser modifications would still allow push services to provide a convenient way of managing push notification campaigns on behalf of a website W while limiting exposure of potentially sensitive information belonging to W 's users. Notice also that the proposed fine-grained SW policies would be somewhat analogous to *Feature Policy* for iframes [145].

2.6.5 Restricting Extensions Permissions

Open Problem: Even if the mitigations described in earlier sections are implemented, a website's SW could still be hijacked by extensions, using the **ExtensionHijack** attack described in Section 2.4.3. Furthermore, extensions can tamper with CSP directives in network responses [55], which can further facilitate SW hijacking attacks.

Proposed Mitigation: Other major browsers, such as Firefox, should follow Chromium's approach and prevent extensions from interfering with responses to SW file requests (e.g., by preventing access to the *FilterResponse* API).

2.7 Measuring In-The-Wild SW Behavior

In this section, our objective is to measure the behavior of in-the-wild SW code used by popular websites. The main goal is to learn how production SW code may be impacted by some of the mitigations we discussed in Section 2.6. We focus mostly on mitigations against continuous execution attacks (Sections 2.6.1 and 2.6.2) and potentially malicious third-party code inclusions (Sections 2.6.3 and 2.6.4), and aim to learn how policy enforcement thresholds may be tuned to curtail possible attacks while minimizing their impact on legitimate SW behaviors.

2.7.1 Browser Instrumentation for SW Forensics

To obtain fine-grained information on the behavior of SW code for real-world web applications, we first developed an instrumented version of the Chromium browser (v84.0.4147.121) with an embedded *Service Worker Forensics* engine.

Our SW forensics engine logs fine-grained information regarding the following:

- The occurrence of SW life-cycle events such as *install*, *activate*, *update*, *uninstall*, and *termination*.
- Any permissions that were requested by the SW code, such as push notifications and geo-location. In addition, we automatically grant these permissions to monitor how their related APIs are utilized.
- Detailed information about network requests issued by SWs, including the URL of resources being fetched.
- API calls made by SW code with respect to caching, push, and notification APIs.
- CPU and memory consumption, and network usage (e.g., number of third-party network requests and related URLs) for each SW instance.
- We also simulate user interactions with the browser that are required to trigger events to be handled by a SW.

The logs generated by our forensics engine are then analyzed offline to measure useful properties about the behavior of SWs in the wild. Since Chromium serves as a basis for many popular browsers, such as Chrome, Opera, Edge, etc., the measurement results we obtained can be considered as representative of SW code running on a variety of browsers.

2.7.2 Experimental Setup

Because the main objective of our measurements is to understand how real-world SW code behaves with respect to the mitigations we proposed in Section 2.6, we focus on analyzing SWs registered by the most popular websites according to Alexa.com [146]. We organize our measurement results by dividing the

top Alexa websites into different *bands*, based on their ranking (e.g., 0-1K, 1K-5K, 5-10K, etc.), as shown in Table 3.2. Different rank bands provide insights into the behavior of websites at different levels of popularity. In total, we identified 5,918 websites that registered at least one SW, with 5,309 sites among the top 100k ranking and an additional 609 sites with rank >100k. However, notice that only a subset of these SWs request notifications permission.

Since we are especially interested in mitigations against continuous execution attacks, we focused our investigation on 1,750 (out of 5,918 websites that register a SW) whose SW code requested notifications permission, and created a small farm of automated instrumented browsers (see Section 2.7.1) to interact with these web pages. For each of these 1,750 web applications, we continued interacting with them and monitored their SW behavior for 3 days. To drive our instrumented browser to automatically visit and interact with these web pages we made use of custom Puppeteer **puppeteer** scripts.

Notice that our data collection and analysis does not include websites such as social media and messaging apps that may require login to send notifications to users. The reason is that for these websites the behavior of their SWs, such as the number and frequency of web push notifications, is highly dependent on user activities and social network. We exclude these sites from our measurements, as it would be highly challenging to simulate a realistic network of users that send messages to each other in a way that is representative of a large and diverse user population. However, it should be noted that SW security policies that aim to impose limits on push notifications may also include a customizable allow-list for popular social media and messaging websites.

Table 2.2: Ranking bands for Alexa’s top 100K websites

Ranking Bands	Band_1	Band_2	Band_3	Band_4	Band_5	Band_6	Total
Ranking Range	0-1K	1K-5K	5K-10K	10K-50K	50K-100K	100K-1M	
#Registered SWs	160	426	437	1917	2369	609	5918
#Analyzed SWs	56	151	145	585	204	609	1750

2.7.3 SW Behavior Results

In Section 2.6.1 we discussed a number of restrictions that could be imposed on SWs to limit their execution time and reduce potential damage that a malicious SW may cause due to *continuous execution* attacks (see also Section 2.4.1). Specifically, among other mitigations (see Section 2.6.1 for details) we proposed to (a) limit overall background execution time, (b) limit the number of push events within a given time window, (c) ensure notifications are visible to users, and (d) limit the volume of third-party network requests.

In the following, we measure how current production SWs behave, compared to the limitations listed above. This will help us in two ways: (i) determine how different limits may impact existing SW behaviors, and (ii) inform the choice of thresholds that could be used in the implementation of new SW security policies. Detailed measurement results are reported in Figure 2.2 and Table 2.3, which we discuss below.

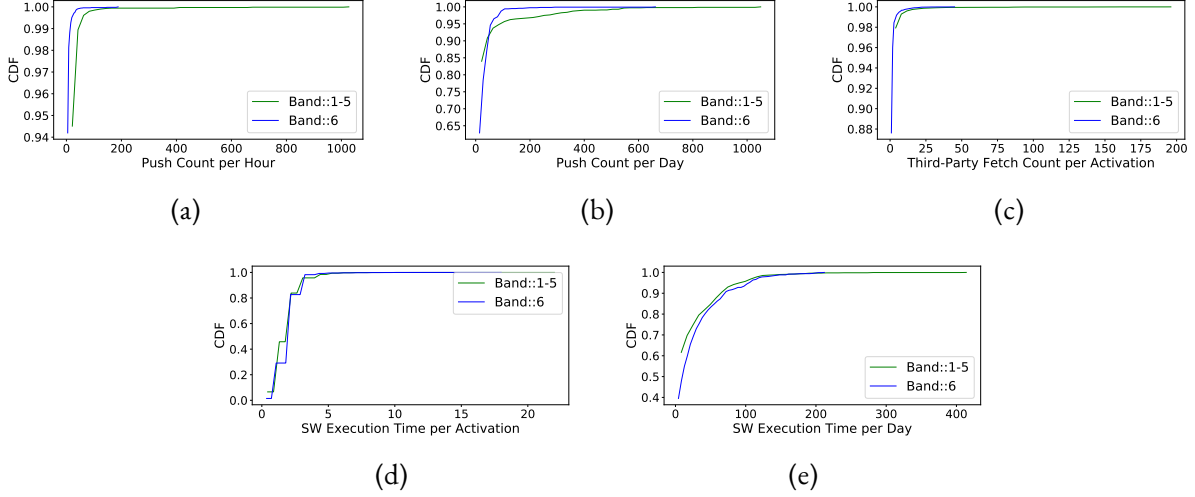


Figure 2.2: SW behavior measurements. Each graph displays the distribution (CDF) of occurrences of an event within a specific time window: a) Push count per hour; b) Push count per day; c) Third-party fetch count per SW activation; d) SW execution time per activation (in minutes); e) SW execution time per day (in minutes).

Frequency of Push Events

Among the 1,750 websites we monitored, 518 of them have a SW that received at least one push event during our analysis period (i.e., 3 days). To estimate the frequency with which push events are received by our instrumented browser, we divide the timeline into slots of one hour, and count the number of push events per hour for each SW. Figure 2.2a shows the distribution (a CDF) of the number of push events per hour across all monitored SWs. Also, from Table 2.3 we can see that at the 90th percentile, SWs receive 14 push events or less per hour.

From Table 2.3, we can see that if we implemented a SW security policy that limits the number of push events per hour to ≤ 14 , this policy would affect (i.e., throttle) the SWs of 49 different websites (49 is the sum of the number of SWs under each ranking band), with almost half of the impacted websites having a ranking above 100k (Band-6). While at a first glance this result may look like a potentially significant impact on production SWs, a detailed manual analysis of the push notifications that would be curtailed by the new policy reveals something different. In fact, we found that all 49 websites that would be potentially impacted sent notifications that could be considered abusive. Specifically, the notifications we recorded from those sites are related to (potentially malicious) WPN ads, which were previously also identified by other researchers in [80] as being often abusive (some example notifications reconstructed from our logs are reported in Figure 2.3). Thus, it appears that the proposed limit on the frequency of push messages would at most throttle the number of push-based ads received by users, without significantly affecting most legitimate SWs. At the same time, limiting the number of push events that can activate a SW can



Figure 2.3: Examples of spam/malicious notifications

help to decrease the potential for continuous execution attacks that may be used for instance to perform cryptomining, DDoS attacks, or other malicious tasks, as further discussed later.

Execution Time

For each SW, we also measured the maximum execution time per activation. Namely, we measure the time for which a SW ran without releasing control or being forced to stop by the browser (as before, these measurements were performed throughout our 3 days of monitoring per each web application’s SW code).

As it can be seen in Figure 2.2d and Table 2.3, at 99% of the instances, SWs were alive for a maximum duration of 5 minutes per activation. At the same time, we also found that 20 websites had a SW that at some point remained active beyond the maximum limit (5 minutes) allowed by the browsers. The maximum continuous execution time per activation that we observed was 22 minutes. These cases of long continuous execution were possible because the SW termination was delayed by the browser as the SW received multiple events (e.g., multiple consecutive push events) in close succession, with the next event arriving and being handled before the SW finished handling the previous event. This again demonstrates that the possibility of abusing SWs to perform malicious tasks such as cryptomining and DDoS attacks remains open.

As in Section 2.6.1, we argue that the browser should impose stricter limits to continuous SW execution. For instance, it could be limited to 5 minutes, since 99% of all SWs activations we measured never exceeded this threshold. Longer continuous execution times should be considered as anomalous and potentially dangerous.

Besides the execution time per activation, we also calculate the overall SW execution time per day, as the sum of the execution time spent during all activations of a given SW for a day of observation. As shown in Figure 2.2e, 95% of SWs were active for less than 90 minutes per day. However, we found 17 websites whose SWs were active from 146 up to 400 minutes (over 6 hours). By analyzing the logs, we found that these websites “spammed” the browser with a large number of potentially malicious notifications (similar to Figure 2.3). As an example of a website whose SW exhibited long execution times, we found

that waploaded.com (ranked 51,299) registered a SW that sent over 50 push events per hour in multiple time windows, and as a result activated its SW and kept it running for long periods (e.g., 22 minutes in one single activation and over 4 hours in a single day).

Table 2.3: Event counts at specific distribution percentiles – The *threshold value* is the percentile value from the corresponding CDF (see Figure 2.6), whereas B-*n* represents the Alexa ranking band *n*.

Event Count	No. of SW Origins	Number of SW Origins above threshold value																				
		90%							95%						99%							
		Threshold Value	B-1	B-2	B-3	B-4	B-5	B-6	Threshold Value	B-1	B-2	B-3	B-4	B-5	B-6	Threshold Value	B-1	B-2	B-3	B-4	B-5	B-6
Push Count per Hour	518	14	0	1	5	11	10	22	22	0	1	3	7	5	8	43	0	0	2	6	2	5
Push Count per Day	518	38	1	2	5	13	9	43	88	1	1	3	8	4	11	392	0	0	0	3	0	1
Third Party Fetch Count per Activation	416	1	12	10	13	35	30	44	2	8	8	10	25	26	36	6	5	4	8	12	19	17
SW Execution Time Per Activation	761	3	2	1	6	16	16	38	3	2	1	6	16	16	38	5	1	1	2	5	5	6
SW Execution Time Per Day	761	64	1	2	7	9	10	33	90	1	0	3	6	4	26	146	1	0	2	3	2	9

Third-party Background Network Requests

To reduce the risk and impact of SWs participating in DDoS attacks, in Section 2.6.1 we proposed to limit the volume of third-party network requests that the SW could issue while in the background (i.e., when the related web application is not rendering on a browser tab).

To understand what may be a good volume threshold, we measured the number of fetch requests that were made to third-party origins by each SW while running *in the background*. Specifically, to identify these requests we perform two different checks: (a) first, we make sure that the request was issued by a SW by checking whether the JavaScript execution context that issued the fetch request belongs to a SW script⁴, then (b) we make sure the request was executed in the background by checking if it was made from inside a *fetch* event listener, which would indicate that it was invoked when a page request is handled by a SW and thus it was not issued in the background. To verify (b), we should notice that whenever a *fetch* listener is started, it invokes the `StartFetchEvent` method and at its completion it invokes `DidHandleFetchEvent` under `ServiceWorkerGlobalScope`. We log these calls and filter out any fetch requests made between these two events, since they are not background requests. At the end of this filtering process, we are left with the *background network requests* made by SWs.

To account for network requests to explicitly authorized third-parties, such as push services that are intentionally imported into a SW's code, we first determine the domain name associated with all URLs in *importScripts* calls and exclude them from our background network requests statistics (i.e., network requests from a SW to its push service domains are effectively counted as first-party requests). After the filtering explained above, we found that 99% of all SWs issued no more than 5 background network requests to third-party origins per each activation.

Although we did not find any evidence of in-the-wild SWs that performed malicious attacks such as DDoS attacks, we were able to reproduce attack code that can indeed send a large number (e.g., 50 per second) of third-party background network requests with no browser limitations. Therefore, we believe that limiting the number of such background requests (e.g., to ≤ 5) per activation, in combination with

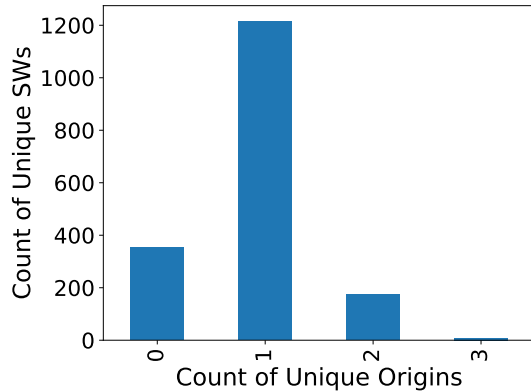
⁴As determined by calling `IsServiceWorkerGlobalScope()` of `ExecutionContext`.

limiting the frequency of activations due to push events, as discussed in Section 2.7.3, is necessary to significantly reduce the risk for SW-based DDoS attacks while having minimal or no impact on the vast majority of legitimate SWs.

Third-Party Code Inclusions

To measure whether it is possible to limit the potential for *hijacking* attacks (see Section 2.6.3), we analyze the number of third-party scripts imported by SWs. To this end, Figure 2.4a shows the count of third-party scripts imported per SW, whereas Table 2.4b shows the top 10 origins related to imported scripts.

The vast majority of origins recorded in our logs belong to third-party push services (just among the top 10 origins, 7 are related to web push services). Also, as we can see from Figure 2.4a, the vast majority of SWs that import third-party code load it from at most one or two origins. Therefore, we believe that the *fail-safe defaults* approach we proposed in Section 2.6.3, whereby the browser should set a `script-src: 'self'` default CSP for every service worker, could be implemented with no significant impact on existing SWs. This is because the developers of existing SWs would only need to add one or two authorized origins from which additional SW code can be imported (e.g., in the Apache web server this could be done with a minimal `.htaccess` file associated with the SW file hosted on the website’s first-party origin [147]).



(a)

Third-Party Domains	# of unique SW Domains
api.pushnami.com	196
www.gstatic.com	140
cdn.onesignal.com	95
app.najva.com	63
s3.amazonaws.com	50
cdn.subscribers.com	40
cdn.aimtell.com	35
storage.googleapis.com	33
cdn.izooto.com	31
cdn.p-n.io	28

(b)

Figure 2.4: Third-Party(TP) Imports (a) Count of TP imported domains per SWs (b) Top 10 TP domains imported by SWs

We also measured whether CSPs are currently used in relation to SW code. To this end, we analyzed the HTTP response for every SW file fetched from the 100K Alexa websites. To identify the request (and related response) for a SW file, we look for the *Service-Worker* request header, which is explicitly found when a page fetches a SW file. We found that 4.8% of all SW files have a CSP headers in the response. However, only 0.8% include the `script-src` directive. Applying `script-src: 'self'` as a default policy would reduce the risk of potential hijacking attacks, such as the **XSS** attacks presented in [44].

Third-Party Code Behavior

Although the default CSP restrictions discussed above may help to mitigate some hijacking attacks, such as **XSS** attacks [44], we need to be mindful that third-party libraries explicitly allowed by web developers may still behave maliciously. For example, as we studied the results of our measurements on SW code imports, we found a few cases of potential unauthorized tracking implemented in third-party libraries. For instance, we found that code imported from *coinPush* (a third-party push service) listens to *fetch* events, and can track all URLs visited on the importing website, even though this may not be necessary to enable the advertised push notification services. Listing 2.4 shows the (simplified) code that appears to track all visited URLs and send them to a remote site.

Such examples demonstrate that it is possible for third-party services to potentially abuse their privileged access to SW code, and should therefore be subjected to stricter default policies by browsers and much more attention by developers. Therefore, our recommendation is that the browser should implement more fine-grained SW policies (similar to *Feature Policy* for iframes) and web developers should carefully isolate SWs that import third-party code by correctly setting their *scope*, as we proposed in Section 2.6.4.

```
//Unauthorized Fetch handler to track request URLs
self.addEventListener('fetch', function (e) {
  if (e.request.url.indexOf(location.origin) === 0 && isDocument(e.request))
  {
    trackingUrl(e.request.url); } });
//simplified version of trackingUrl method
function trackingUrl(url){
  var body = {registration_id: obj.token,
              sender_id: obj.senderId,
              logs: {url: url, timestamp: timestamp}}
  return fetch(TRACKING_SERVER + '/tracking_url',
              { method: 'POST',
                mode: 'no-cors',
                body: body } )
}
```

Listing 2.4: SW Code of 3rd Party Push service

2.8 Implementing New SW Policies

To demonstrate that implementing the new policies proposed in Section 2.6 is possible with reasonable engineering effort, in the following we discuss our own proof-of-concept implementation in the Chromium (v84.0.4147.121) browser of some of those policies. We plan to release our source code after publication.

```
{  "name": <policy-name>,
    "severity": <value>,
    "threshold": <threshold>,
    "duration_in_minutes":<duration>  }
```

Listing 2.5: Template for count-based policies

To implement the new policies, we developed a new class called `SWPolicies` within the *Blink* rendering engine. In case of policies concerned with limiting the frequency of events that activate a SW, we follow a template similar to the one shown in Listing 2.5. Each time an event such as *push* or *fetch* occurs, we invoke a corresponding method to update the related counter (e.g., *push_count_per_hour*) and check if the count falls within a predefined *threshold*, which could be selected based on the trade-offs we discussed in our SW behavior measurements results (see Section 2.7). When a policy violation occurs (i.e., the *threshold* is exceeded), we log the violation and increase a *severity* indicator. Then, if the severity level reaches a predefined maximum value, the browser immediately terminates the SW (and could also deregister it, if the user engagement score for the SW’s origin is very low).

```
void SWPolicies::OnSWActivated(ExecutionContext* ex)
{ //start timer
  swpolicy_info->sw_timer->Start(
    FROM_HERE, kServiceWorkerRunningDelay,
    base::BindOnce(&SWPolicies::OnSWTimeout,
    base::Unretained(this), To<SWGlobalScope>(ex)));
}
void SWPolicies::OnSWTimeout(SWGlobalScope* gs)
{ // immediately terminate SW
  gs->SetIdleDelay(base::TimeDelta::FromSeconds(0));
}
```

Listing 2.6: Example code for limiting SW execution time

As a simplified example of how to terminate a SW, to stop a SW that exceeds a given execution time we can start a timer whenever the service worker is activated, as shown in Listing 2.6, and attach a callback method that will be called once the timer expires. At this point, the callback can check the state of the SW and terminate it (if it is still running) by calling Blink’s `SetIdleDelay` method with delay set to 0 seconds.

We use an approach similar to that described above to implement and enforce the policies proposed in Section 2.6.1. Furthermore, we tested these policies against a number of SW attacks (using the approaches we described in Section 2.4) that attempt to perform DDoS attacks, cryptomining, notification spam, etc., and verified that we are indeed able to greatly throttle such attacks, rendering them ineffective.

2.8.1 Discussion

As we consider the implementation of new browser policies that would restrict SW behaviors, as discussed earlier, we should carefully consider how they could impact legitimate SWs. In our measurements (Section 2.7), we showed that it would be possible to find enforcement thresholds whose effect is to greatly limit abuse while interfering with the behavior of only few actual legitimate SWs. In addition, we should consider that the implementation of the proposed policies could include a customizable allow-list that can be pre-populated by the browser vendor and extended with help from the user, if preferred. For instance, consider the limits on the frequency of push notifications proposed in Section 2.6.1. If a popular website

(e.g., a social media platform) legitimately needs to send a large number of notifications (e.g., many tens or hundreds of notifications per day), such an application could be added to this allow-list.

The effect of the proposed policies against continuous execution attacks (Section 2.6) is that they may also limit the execution time for a small fraction of legitimate SWs. However, we should notice that browsers already implement a mechanism to terminate a SW's execution after a certain amount of execution time. Therefore, SW developers already need to take into account that their code could be forced into an idle state. Unfortunately, as we demonstrated in Section 2.4, attackers can trick the browser into executing SWs for much longer than it would be otherwise allowed, which motivates the new policies we proposed Section 2.6. Ultimately, we believe that limiting the execution time of SWs would have a small to negligible impact on legitimate SW code, while drastically reducing the risk for SW abuse. Additionally, before enforcing the new policies, browsers could grant a grace period during which an alert is issued every time a SW policy is violated without strictly enforcing the policy itself. During this transition period, developers will then have the time to adjust their SW code to make sure the new policies are not violated moving forward.

In general, because the vast majority of legitimate web applications do not require a completely unfettered access to push events, web notifications, background third-party requests, etc., as shown in Section 2.7, and considering the potential damage that SW abuse could cause given its powerful features (see Section 2.4), we believe browsers should follow an approach akin to the *least privileges* principle as much as possible and limit those and other SW privileges.

2.9 Related Work

Throughout the paper we have discussed a number of previous works that focus primarily on attacks against SWs or in which SWs play a fundamental role. In this section, we briefly discuss other works related to multiple aspects of SW security, including some additional attacks and mitigation measures.

New, recently published studies [45], [46] discuss other attacks that leverage SW features. In [45], Squarcina et al. describe a powerful DOM-based XSS attack that could be used to inject code into a script cached by a SW, through which the injected code remains persistent in the client's machine for a longer duration compared to traditional XSS attacks. This attack is similar to cache poisoning attacks for the *Browser Cache* [47], [48], *Client Storage* [49] and *Web Cache* [50], [51] mechanisms. Unlike attacks discussed in Section 2.4, the role of SWs in this attack is limited to serving already poisoned cache resources. However, it still highlights the need for stricter access restrictions similar to our proposed mitigation of isolating SW cache access (Section 2.5).

In [46], the authors discuss a number of security issues related to rehosted websites and describe a persistent man-in-the-middle attack implemented by leveraging SWs. Since all rehosted websites may be placed under the same origin as the attacker site that registers a SW, the SW could gain control of all resources under the rehosted origin, rendering the Same-Origin-Policy for SWs ineffective. Mitigating this attack likely requires defense measures at the rehosting level, as the attack violates a number of security policies.

Soon after SWs were introduced by browsers, a few studies [52], [53] discussed the possibility of side-channel attacks that leverage SW features. More works [54]–[58] are dedicated to measuring privacy leakage due to third-party code, such as extensions, external libraries and ad injections. In [59], the authors discuss issues related to blind trust between cross origins and explain the extent of damage it could lead to. Other works that focus on client-side web security are [60]–[62], which include policy-based access to restrict Javascript APIs (such as Performance API) to mitigate timing based side-channel attacks. Similarly, [63] presents a cost-benefit analysis to enable restrictions per site without affecting its legitimate use. In [64], Jackson and Barth argue that the concept of “fine-grained origins” (FGO) is a flawed solution to curb origin contamination. In this paper (Section 2.6.4), we discussed the use of scopes as a way to isolate third-party SWs and proposed additional measures to restrict the capabilities of third-party SWs to mitigate possible origin contamination issues for SW scripts.

Other systematization of knowledge papers that consider both attacks and mitigation measures have focused on web security [65]–[67], mobile security [68], and IoT security [69]. Our work is different because we focus on attacks on Service Workers specifically, discuss existing mitigations, present a timeline of when attacks and mitigations were introduced, present open security problems, and proposed new policies that browsers could adopt to limit the damage that SW abuse could cause.

2.10 Conclusion

In this paper, we reproduced and analyzed known attack vectors related to Service Workers and explored new abuse paths that have not previously been considered. We systematized the attacks into different categories, and analyzed whether, how, and when these attacks have been published and mitigated by different browser vendors. Then, we discussed a number of open SW security problems that are currently unmitigated, and propose SW behavior monitoring approaches and new browser policies that we believe should be implemented by browsers to further improve SW security. Furthermore, we implement a proof-of-concept version of several policies in the Chromium code base, and also measure the behavior of SWs used by highly popular web applications with respect to these new policies. Our measurements show that it is feasible to implement and enforce stricter SW security policy without a significant impact on most legitimate production SWs.

CHAPTER 3

WHEN PUSH COMES TO ADS: MEASURING THE RISE OF (MALICIOUS) PUSH ADVERTISING

¹Karthika Subramani, Xingzi Yuan, Omid Setayeshfar, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. 2020. In Proceedings of the ACM Internet Measurement Conference (IMC '20). Association for Computing Machinery, New York, NY, USA, 724–737.

Reprinted here with permission of publisher.

3.1 Abstract

The rapid growth of online advertising has fueled the growth of ad-blocking software, such as new ad-blocking and privacy-oriented browsers or browser extensions. In response, both ad publishers and ad networks are constantly trying to pursue new strategies to keep up their revenues. To this end, ad networks have started to leverage the *Web Push* technology enabled by modern web browsers. As web push notifications (WPNs) are relatively new, their role in ad delivery has not yet been studied in depth. Furthermore, it is unclear to what extent WPN ads are being abused for *malvertising* (i.e., to deliver malicious ads). In this paper, we aim to fill this gap. Specifically, we propose a system called PushAdMiner that is dedicated to (1) *automatically* registering for and collecting a large number of web-based push notifications from publisher websites, (2) finding WPN-based ads among these notifications, and (3) discovering malicious WPN-based ad campaigns.

Using PushAdMiner, we collected and analyzed 21,541 WPN messages by visiting thousands of different websites. Among these, our system identified 572 WPN ad campaigns, for a total of 5,143 WPN-based ads that were pushed by a variety of ad networks. Furthermore, we found that 51% of all WPN ads we collected are malicious, and that traditional ad-blockers and URL filters were mostly unable to block them, thus leaving a significant abuse vector unchecked.

3.2 Introduction

In the past few years, the rapid growth of online advertising has fueled the growth of ad-blocking software, such as new ad-blocking and privacy-oriented browsers (e.g., Brave [148]) or browser extensions (e.g., AdblockPlus [149]). In response, both ad publishers and ad networks are constantly trying to pursue new strategies to keep up their revenues. To this end, ad networks have started to leverage the *Web Push* technology enabled by modern web browsers [150]. Until relatively recently, push notifications were mostly limited to native apps on mobile platforms, and web-based applications were unable to connect to their users out of active browsing sessions. However, now Web Push allows for web applications to send out Web Push Notifications (WPN) at any time to re-engage their users, even when the browser tab in which the web application was running is closed (the browser itself needs to be running, but does not need to be in the foreground for a WPN to be delivered to the user). Furthermore, unlike push notifications from native mobile apps, WPNs allow for notifications to be displayed on both desktop and mobile devices. Thus, they serve as a single tool with support to reach users on multiple platforms.

Although WPNs were initially designed for websites to deliver simple messages (e.g., news, weather alerts, etc.), they have become an effective way to also serve online ads, and can therefore be abused to also deliver malicious ads. In particular, the use of WPNs for ad delivery has some unique advantages. First, unlike traditional online ads (banner ads, pop-up ads or pop-under ads), advertisers do not have to wait for users to reach the web page that publishes the ad. Instead, advertisers can send out notifications that can allure users to their targeted content. Secondly, thanks to years of experience with native mobile app notifications, users have been trained to compulsively interact with push notification messages (at least

on mobile devices). WPN-based ads may also be less prone to *ad blindness* [151], compared to traditional web ad delivery mechanisms such as page banners. Furthermore, ad-blocking software are not currently effective at blocking WPN-based ads (see Section 3.7.4), in part because browser extensions are not allowed to interfere with the *Service Workers* code through which WPNs are delivered [9]. For these reasons, some ad networks are focusing their business specifically around WPN ads (e.g., RichPush [152]).

As WPNs are relatively new, their role in ad delivery has not yet been studied in depth. Furthermore, it is unclear to what extent WPN ads are being abused for *malvertising* (i.e., to deliver malicious ads). In this paper, we aim to fill this gap. Specifically, we propose a system called PushAdMiner that is dedicated to (1) *automatically* registering for and collecting a large number of web-based push notifications from publisher websites, (2) finding WPN-based ads among these notifications, and (3) discovering malicious WPN-based ad campaigns. To build PushAdMiner, we significantly extend the Chromium browser instrumentations developed by [153] and [5], which have been open-sourced by the respective authors. Specifically, neither [153] nor [5] are able to track the activities of Service Workers in detail. Therefore, we implement our own set of browser instrumentations that allows us to track WPNs in all their aspects, from registration to notification delivery, on both desktop and mobile devices. We then build a custom WPN crawler around our instrumented browser to automatically receive, track, and interact with generic WPNs, including collecting malicious WPN ads and their respective malicious landing pages. Finally, we develop a data mining pipeline to analyze the collected WPNs and discover malicious WPN-based campaigns.

To the best of our knowledge, ours is the first systematic study that focuses on automatically collecting and analyzing WPN-based ads and on discovering malicious ad campaigns delivered via WPNs. In contrast, previous work focused on other security-related aspects of Service Workers and Push Notifications, such as building stealthy botnets [9], or social engineering attacks that attempt to force users into subscribing to push notifications [5], but without studying the resulting push messages. Lee et al. [154] study Progressive Web Apps. They collect Service Worker scripts from top-ranked website homepages and analyze their push notifications. Their work studies potential security vulnerabilities related to Service Workers, App Cache, and discusses how push notifications may be abused to launch phishing attacks, without measuring how prevalent these attacks are in the wild. Our work is different, in that we aim to *automatically* collect and analyze WPN-based ads, to discover WPN ad campaigns, and to measure the prevalence of malicious WPN-based ad campaigns in the wild.

In summary, we make the following contributions:

- We present PushAdMiner, a system that enables the automated collection and analysis of online ads delivered via web push notifications (WPNs) on both desktop and mobile devices.
- To track WPNs, we extend a Chromium-based instrumented browser developed in [5], [153] to allow for a detailed analysis of Service Workers, which are at the basis of WPN deliveries. Furthermore, we build a custom WPN crawler around our instrumented browser to collect and automatically interact with WPNs.

- Using PushAdMiner, we collected and analyzed 21,541 WPN messages by visiting thousands of different websites. Among these, our system identified 572 WPN ad campaigns, for a total of 5,143 WPN-based ads that were pushed by a variety of ad networks. Furthermore, we found that 51% of all WPN ads we collected are malicious, and that traditional ad-blockers and URL filters were mostly unable to block them, thus leaving a significant abuse vector unchecked.

3.3 Motivating Example and Background

In this section, we provide an example of WPN-based malicious ad, and then briefly explain the concepts and technologies behind web notification services.

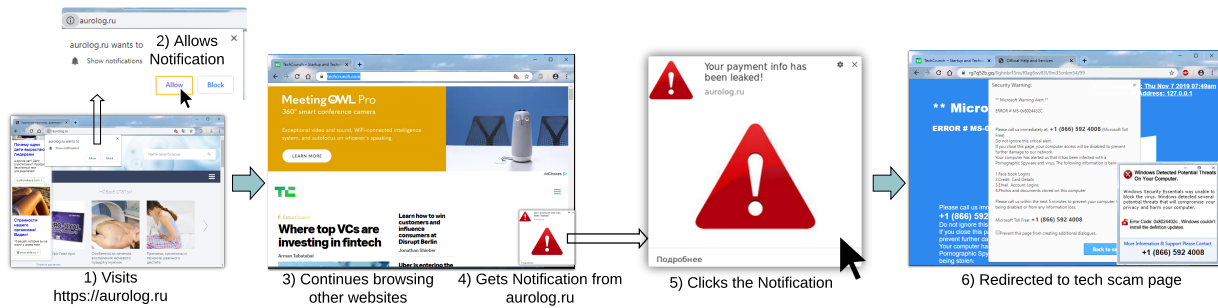


Figure 3.1: Example of malicious advertisement served through web push notifications

3.3.1 Motivating Example

Figure 3.1 provides an example of malicious WPN-based ad. During the preliminary stages of our research, we stumbled upon a website on aurolog.ru. When visiting the main page, the site requested permission to send us notifications. We granted permission by pressing the *Allow* button on the browser dialog box, and subsequently received a WPN ad with the following alert message: “Your payment info has been leaked” (see Figure 3.1). After clicking on the notification, we were redirected to a *tech support scam* [155]. To our surprise, the landing URL was neither blocklisted by Google Safe Browsing[95] nor detected as malicious by any of the web page scanners on Virus Total[156]. This example confirmed our suspicion that WPNs may be abused for malvertising, and sparked our investigation to determine whether such cases of malicious WPN-based ads could be automatically collected and analyzed.

3.3.2 Technical Background

Recent changes in HTML5 have introduced new web features, such as *Service Workers*[157], *Push Notifications*[158] and *AppCache*[159]. Websites that adopt these technologies are called Progressive Web Apps (PWAs). Throughout this paper, we refer to push notifications sent by PWAs using a browser as Web

Push Notifications (WPN), to distinguish them from push notifications sent by native apps on mobile devices, and refer to Service Workers as SWs for brevity.

Service Workers and Push Notifications: A *Service Worker* (SW) is an event-driven script executed by the browser in the background, separately from the main browser thread and independently of the web application from which it was initially registered and that it controls. In practice, a SW comes in the form of a JavaScript file that is registered against the origin and path of the web page to which it is associated (only HTTPS origins are allowed to register a SW). In effect, SW can be viewed as “a programmable network proxy that lets you control how network requests from your page are handled” [160].

Service Worker can use the *Push API*[116] to receive messages from a server, even while the associated web application is not running. It is worth noting that a single web app is allowed to register multiple SWs. Service Workers can also use the *Notifications API*[158] to display system notifications to the user. A prerequisite is that the web application must first request permission to display notifications to the user (only allowed for HTTPS origins). If the user accepts (i.e., clicks on “Allow” instead of “Block” on the notification request popup) to receive notifications from the web application’s origin, this permission persists across browser restarts, and until the user explicitly revokes the permission via browser settings/preferences (notice that non-expert users may find it difficult to understand, find, and disable notifications in the browser’s settings).

Web notification messages have a number of customizable parameters, such as title, body, target URL, icon image, display image and action buttons. The user can interact with a notification by either clicking on it, closing it or performing any custom actions displayed in the notification message. SW can listen to such user events and take action according to the input. This includes loading target URL on a separate tab, following a user’s click on the notification box.

Firebase Cloud Messaging (FCM): FCM is a cross-platform messaging solution for Push Notifications. It can serve as a central authority that mediates the communication between the ad server and the Service Worker. Upon initial registration, FCM creates a unique registration ID per user and per Service Worker, which is sent along with an endpoint URL [161] to the ad server. For further details, refer to FCM’s online documentation [162].

3.4 System Overview

In this section, we provide an overview of how PushAdMinerworks, leaving a detailed description of the main system’s components to Sections 3.5 and 3.6. A high-level representation of the system is provided in Figure 3.2.

PushAdMinerconsists of three main components: (i) an instrumented browser to collect fine-grained information about SWs and WPNs; (ii) a custom crawler that automatically visits sites and interacts with the browser, including granting notification permissions and interacting with WPNs (Section 3.5); and (iii) a data analysis component aimed at identifying WPN-based ad campaigns and labeling likely malicious ones (Section 3.6).

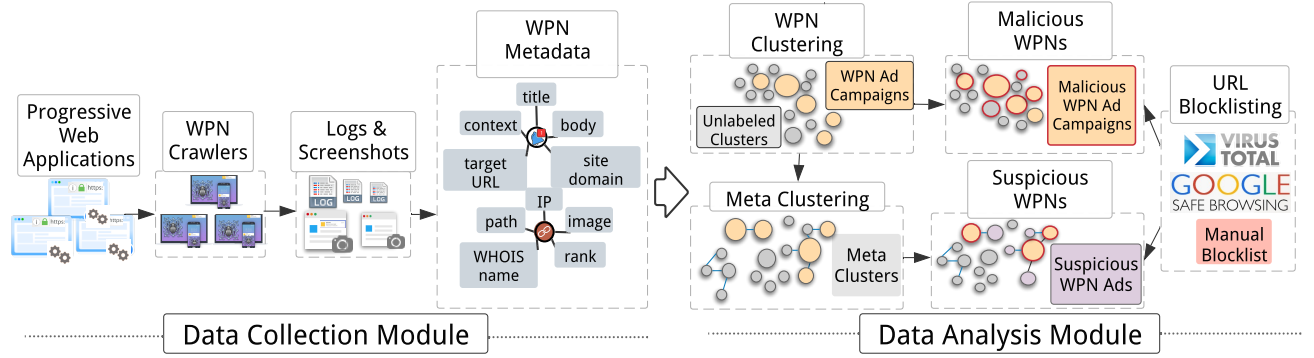


Figure 3.2: PushAdMiner System Overview

While a number of browser automation and crawling systems have been proposed, including Selenium [163], Puppeteer [164], and others [5], [153], [165], currently they do not fully support the automatic user interactions with WPNs and collection of all details about SWs needed for our study. We therefore built an instrumented browser based on Google Chromium, by significantly extending existing open-source Chromium instrumentations [5], [153]. In addition, we leveraged Puppeteer [164] for browser automation and event logging, and wrote custom scripts to record SW registrations and network requests.

Figure 3.2 presents an overview of how PushAdMiner collects information about WPNs, and how the browser logs are analyzed to identify ad campaigns in general and discover malicious ones among them. First, we discover a set of URLs that may send push notifications with the help of an ad network and filter those that actually request for notification permission (see Section 3.7.1 for details). For the web pages that ask for notification permissions, we log details about the responsible SW code, automatically grant permission (via browser code instrumentation), and then collect the notifications that are later pushed to our instrumented browser. When a notification is displayed by the browser, we record fine grained details about the notification message itself (including message text and icons), automatically simulate a user click on the notification box (via browser code instrumentation), and track all events resulting from the click. If the click results in a new page being open, we record detailed information about the related network requests, including all browser redirections, as well as detailed logs and a screenshot of each new page the browser visits, including the landing page (i.e., the final web page reached due to the click).

Finally, we extract relevant information from the detailed logs of our instrumented browser, and apply a clustering strategy to find notifications that are similar to each other, which allows us to identify WPN-based ad campaigns. We then leverage URL blocklists to find WPN ad campaigns that are likely malicious (e.g., because one or more landing pages are known to be malicious).

Note that in this paper we do not focus on building a malicious WPN ad campaign detector, such as using statistical features or machine learning classifiers. Rather, our focus is on discovering, collecting, and analyzing WPN ad campaigns in general, and on measuring the prevalence of both benign and malicious

campaigns. As we will show in Section 3.7, URL blocklists tend to miss a significant number of malicious URLs that we determine to be related to malicious ad campaigns. The analysis we present in this paper could therefore be used as a starting point for developing an automated malicious WPN ad campaign detector. We leave this latter task to future work. Our code to collect and analyze WPNs is publicly available in a Github repo¹ and a Docker container with the instrumented Chromium is in Docker Hub².

Ethical Considerations: To track WPN-based ads and label malicious ones, it is necessary to collect information about the landing page that an ad eventually redirects to. For instance, for most malicious ads the attack is effectively realized only once the user reaches the landing page, especially in case of social engineering and phishing attacks. As we do not know in advance what landing page will be reached by clicking on a WPN message, and whether a WPN ad is malicious or not, our system will likely click on both legitimate and malicious ads. In turn, this may cause legitimate advertisers to incur a small cost for our clicks, as they will likely have to pay a third-party publishing web page and ad network for their services (notice that we obviously receive no monetary gain whatsoever during this process). This is common to other similar studies, such as [5], [83], and we therefore address the ethical considerations for our study following previous work.

To make sure we do not have a significant negative impact on legitimate third-parties, we estimated the cost incurred by these advertisers due to ad clicks performed by our system, and found that our system has negligible impact on advertisers. Specifically, among the WPN ads we identified, we consider legitimate ads to be those whose landing pages are not labeled as malicious by Virus Total’s URL classification services. Then, we estimate the *cost per landing domain* based on the number of ads we clicked on that lead to a specific domain, using the *Cost Per Mille (CPM)* [166] for push notification ads according to iZooto [167]. The maximum cost per landing domain throughout our entire study was USD 1.12 (due to landing on the same domain 444 times), which we calculated using the standard CPM of USD 2.54. On average, we visited each landing domain 18 times, which corresponds to an average cost of USD 0.04 per landing domain (i.e., per advertiser). Considering these low values, we believe the impact of our system on advertisers is not significant, and is on par with previous work [5], [83].

3.5 Data Collection Module

In this section, we describe in detail how PushAdMiner’s data collection module is implemented. The steps referred to in the following sections follow the numbering given in Figure 3.3.

¹<https://github.com/karthikaS03/PushAdMiner>

²https://hub.docker.com/repository/docker/dockerammu/docker_puppeteer_chromium_xvfb

3.5.1 Desktop Environment

To discover WPN ad campaigns, we first need to collect WPN messages. To this end, we build a crawler consisting of an instrumented browser and browser automation scripts. As our crawler encounters a website that asks for permission to send push notifications, our goal is to automatically allow the permission request, so that we can collect notifications from that origin. To this end, we instrumented our browser as follows: we introduce changes to the `RequestPermission` and `PermissionDecided` methods under the `PermissionContextBase` class in Chromium's C++ code base, to log all details about the permission request (e.g., the origin requesting it) and to automatically grant permissions.

Service Worker Registration (step 2): To record when a SW is registered by a web page, we use a custom script for Puppeteer [164]. Specifically, we listen to `serviceworkercreated` events and log information such as details about the page that registered the SW, and the URL from which the SW code was retrieved.

Network Requests (step 3): When the browser receives a push message from the FCM service, it will alert the SW to which the message is destined. The correct SW is identified via a unique ID included in the push message. As part of handling the push message and related notification, the SW may issue one or more network requests directly to an ad network server or other third-party servers. For instance, the SW may contact an ad network server to determine the landing URL associated to a given push message. Also, after a pushed notification receives a user click, the SW may send a network request to an ad server to notify it of the user click, or to facilitate other tracking related activities. We capture and log such network requests issued by SWs by leveraging a custom Puppeteer [164] script that listens to SW request and response events. Specifically, for every SW's network request we record information such as the requested URL, the data sent/received, possible network redirections, and the related response content.

Notifications (steps 5-6): Notifications are displayed by invoking `showNotification` under `ServiceWorkerRegistrationNotifications` (in Chromium's C++ code). We therefore add an instrumentation hook to record calls to that method. Specifically, we log the URL of the SW that called for showing the notification, the notification title, body, icon URL, and target URL (if present).

Notification Clicks and Navigations (steps 7-8): Depending on the type of notification, clicking on it can navigate the browser to a new web page (e.g., on a separate tab). For WPN-based ads, clicking on the notification box typically takes the user to the page advertised in the WPN ad (i.e., the ad's landing page). Collecting such pages is especially important for studying malicious WPN ads, because the landing page often implements a critical component of the attack.

For instance, Figure 3.1 shows that the user reaches the actual *tech support scam* page after clicking on the notification. The landing page advertises the scam phone number, without which the attack could not be monetized.

To automate the process of collecting the landing page associated to push notification messages, we need to simulate a user click. Unfortunately, browser automation frameworks such as Puppeteer [164] and Selenium [163] do not allow for interacting with WPNs. Therefore, we again had to build our own custom browser instrumentation. Specifically, we found that Chromium uses the `Add` method

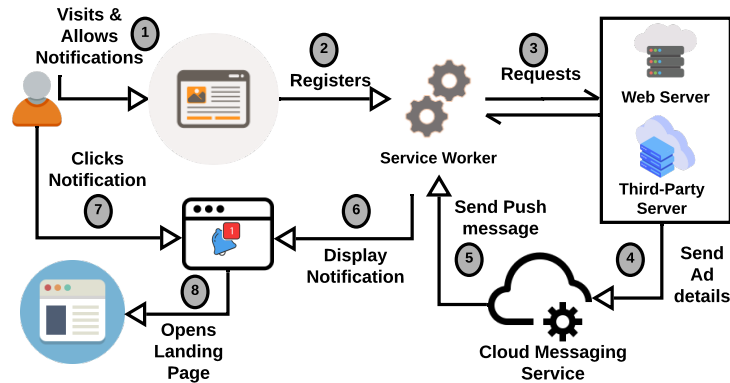


Figure 3.3: Steps involved in Serving Ads via WPNs

in `MessageCenterNotificationManager` (in the C++ code base) to trigger the display of a notification. Also, we found that `WebNotificationDelegate` has a method called `Click` that is responsible for propagating user clicks to the notification. Therefore, to simulate a user click we instrument the `Add` method mentioned above so that, after the notification is displayed, it waits for a short delay (e.g., a few seconds) and then calls `WebNotificationDelegate::Click`. If the `Click` call results in a web page navigation, our browser records all network requests involved in the process (including all redirections) and fine-grained details about the rendering of the landing page, which includes detailed information about any JavaScript code executed in the context of the landing page.

3.5.2 Mobile Environment

We also developed a version of `PushAdMiner` for Android. Due to some technical differences between how WPNs are displayed on a mobile OS, compared to desktop environments, we had to adapt some of the system components to run specifically on Android. First, at the time when we started building our system, `Puppeteer` [164] did not appear to support Android Chromium automation. Only recently there have been online posts in which `Puppeteer` users describe how they have been able to “hack” their configurations to remotely control an Android browser. We therefore built our own browser automation framework that works via the Android Debug Bridge (ADB). The capabilities of our ADB-based automation framework are limited, but sufficient for enabling data collection for `PushAdMiner`. We plan to explore the use of `Puppeteer` for `PushAdMiner` on Android as future work.

Logging Internal Browser Events: We compile our instrumented Chromium browser for Android, so that we can collect intimate details about internal browser events related to WPNs, including recording information about the related Service Workers and the rendering of the landing page resulting from clicking on a WPN. Browser logs are sent via the `logcat` ADB command to a remote logging machine.

Interacting with Notifications: Unlike on desktop devices, in which WPN messages are displayed by the browser, on Android device it is the Android OS that displays a WPN as a system notification. Also, unlike on desktop environments, the browser does not need to be activated for a WPN message to be received, though the browser may be activated after tapping on a notification (e.g., to navigate to the URL pointed to by the notification). We therefore had to implement a different mechanism to simulate user interactions with WPNs on Android. Specifically, we developed an Android application that leverages Android’s *Accessibility Service*. The Accessibility Service is aimed to help people with disabilities in using the device and apps. It is a long-running privileged system service that helps users process information from the screen and lets them interact with the content meaningfully in an easy way. Android developers can leverage the Accessibility Service API and develop apps that are made aware of certain events, such as `TYPE_VIEW_FOCUSED` and `TYPE_NOTIFICATION_STATE_CHANGED`. Furthermore, the accessibility service API can also be used to initiate user actions such as click, touch and swipe.

We install our app with Accessibility Service permission on an Android physical device, and use it to interact with every notification event fired. Whenever a new notification pops up, our application will automatically swipe down the notification bar and click on the notification to complete the action, while our instrumented Android browser produces detailed logs about the consequences of such interactions (e.g., loading a new web page).

As an alternative, Android Debug Bridge (ADB) could be leveraged to implement the same browser automation that we implemented using Accessibility Service. However, in practice, we found that it could create large traffic overhead through the USB cable connected with the device. As we already use ADB to retrieve fine-grained browsing logs, we decided to avoid further USB overhead. Therefore, we found the use of the accessibility features to be better in practice for our specific application.

3.6 Data Analysis Module

In this section, we first describe in detail how we mine the collected WPN messages to identify WPN-based ad campaigns, and later explain how we label malicious campaigns.

3.6.1 WPN-based Ad Campaigns

To identify WPN-based ad campaigns, we mine the dataset of WPN messages collected by PushAdMiner from a large and diverse set of websites. To distinguish between generic WPN messages and WPN-based ads, we start by considering the following intuitions. Roughly speaking, advertisers tend to promote their products or services on multiple websites, and clicking on a WPN notification typically leads to a third-party landing page, on a different origin than the website from which the WPN message was received. On the other hand, non-ad WPN messages are typically related to alerts (e.g., breaking news, weather alerts, etc.) that are specific to the notifying website itself, and clicking on the notification often leads to the same origin to which the notification was related.

Following the above intuitions, we broadly define a *WPN ad campaign* as a set of WPN messages from multiple sources that deliver similar content promoting the same (or similar) products or services. In practice, this translates into a (potentially large) group of similar WPN messages pushed by multiple different websites that lead to the same landing page, or different landing pages that show similar content.

To find such WPN ad campaigns among a large collection of generic WPN messages, we leverage a document clustering approach. This clustering process aims to group together WPN messages that are similar to each other in terms of message content and landing page. As a result, WPN ads that belong to a large WPN ad campaign will tend to form larger clusters. On the other hand, non-ad WPNs would tend to be isolated into singleton clusters or clusters that contain messages related to only one source website and a landing domain that points back to the source website itself.

Because clustering is an unsupervised learning process, it is usually challenging to tune the hyperparameters to obtain perfect clustering results. Therefore, to minimize the chances of grouping together ads and non-ads, we tune our clustering system to be conservative and yield “tight” clusters. Namely, WPN messages assigned to the same cluster will exhibit high similarity. Although this may result in some WPN ads being left out of the campaign (i.e., the cluster) they may also belong to, we will see later that we can reconnect them to their respective campaigns using a meta-clustering step (Section 3.6.3).

WPN Clustering Features and Approach

To cluster similar WPNs, we first need to define the features to be compared and a similarity function that calculates the closeness of two WPN messages. Thanks to our instrumented browser, each WPN we collect is accompanied by metadata such as the notification title, message body, the images and icons found in the WPN, the URL of the landing page, a screenshot of the landing page reached after clicking on the WPN (in the case of our desktop browser), the events that occur immediately after a notification click, etc.

Among the above features, to discover WPN ad campaigns we use the following ones for clustering: message title, message body, and the URL path (i.e., we exclude the domain name) of the message’s landing page URL. We then use these features to calculate the pairwise distance between WPN messages. Specifically, given two messages, we compute the distance between their title and body text, and separately the distance between their landing URL paths. Then, we compute the total distance between two WPNs as the average of these two distances. We explain the distance measures in more detail below. The remaining information contained in the WPN metadata collected by our browser (e.g., domain names, screenshots, etc.) that are not used as features will instead be used later to help us validate the clustering results.

WPN Messages Distance: The text contained in the title and body of a WPN message is typically short and includes specific keywords that reflect the *theme* of the message. To measure the similarity between these short pieces of text (the concatenation of title and body), we require a measure that gives importance to significant keywords found in the content of WPNs. To this end, we use the *soft cosine similarity* [168] measure, which considers the semantic relationship between words. To compute the similarity between words, we first use Word2Vec [169] on all WPN messages to obtain a term-similarity matrix. We then convert each WPN message into a bag-of-words representation, and input both the term-similarity matrix

and the bag-of-word vectors for each pair of WPN messages into the cosine similarity function (we use `softcosim()` implemented in `gensim`[170]) to obtain a pairwise similarity matrix for WPN messages. Since the similarity value $s \in [0, 1]$, we calculate the distance as $d = (1 - s)$.

URL Paths Distance: Given the landing page URL of a WPN (which we collect along with other metadata, as explained earlier), we extract the URL path by excluding the domain name and the query string values, while retaining the relative path to the page and query string parameter names. To calculate the distance between two URL paths, we use the Jaccard distance between tokens extracted from the URL path, such as the components of the directory path, the page name, and the name of the query string parameters.

WPN Clustering: Once the pairwise distances are computed, we use agglomerative hierarchical clustering over the distance matrix to form *WPN clusters*. To determine where to cut the resulting dendrogram, we compute the average silhouette score [171] for clustering results obtained at different cuts, and choose the cut with the highest score.

WPN Ad Campaigns: As per our definition of WPN ad campaigns, to determine if a WPN cluster is formed by ads, we take into consideration the source of the WPNs. Namely, we count the number of effective second-level domain names associated with the websites that sent the push notifications per each cluster. This number tells us if the WPNs have been published on multiple sources. If a cluster contains more than one distinct second-level source website domain, we label the cluster as a *WPN ad campaign*.

3.6.2 Identifying Malicious WPN Clusters

To determine the maliciousness of a WPN cluster, we leverage two well-known URL blocklisting services: Google Safe Browsing (GSB)[95] and Virus Total[156] (VT). We submit the full URLs of all the landing pages reached from all WPN messages in the cluster to these services. Then, we label a particular WPN message as *known malicious* if the landing page URL is blocklisted as malicious by any of the blocklisting services. Next, we use a simple label propagation policy to flag as malicious the WPN clusters that contain at least one known malicious WPN.

It is worth noting that we submit full URLs to GSB and VT. If a full URL, u , is blocklisted, it does not necessarily mean that all URLs under the same domain name as u will also be blocklisted (in fact, we found some cases that confirm this observation for VT). In addition, similar malicious WPN messages often lead to different domain names, mainly as an attempt to evade blocking by URL blocklists. At the same time, because WPN messages in the same cluster are very similar, thanks to our conservative clustering approach, they share very similar title, body, and structure of the landing page URL path. Intuitively, if one WPN message is known to lead to a malicious landing page, it is highly likely that all other WPN messages in the same cluster also lead to similar malicious pages, as we will also discuss in our measurement results (see Section 3.7). That is why we rely on the simple “guilty by association” policy mentioned above for label propagation.

3.6.3 WPN Meta Clustering

Because URL blocklists have limited coverage, it is possible that some malicious WPN clusters will not be immediately identified using the labeling approach discussed above. Furthermore, because our clustering approach is conservative, it is possible that separate clusters of WPN messages may in fact be related to each other. To compensate for this, we perform a meta-clustering step that aims to group together WPN clusters that may belong to the same WPN “operation” (e.g., the same advertiser), as explained below.

To this end, we generate a bipartite graph $\mathcal{G} = (W, D, E)$. W is a set of nodes in which each node represents a WPN cluster obtained as described in Section 3.6.1. D is the set of all domains pointed to by the WPN messages we collected (i.e., all domains related to any of the landing page URLs found in the WPN messages), and E is a set of edges in which each edge connects a node $w \in W$ to a node $d \in D$. Specifically, we connect each WPN cluster to the domains related to the landing page URLs linked by the WPN messages in the cluster. Then, we find all isolated components, $G_1, G_2, \dots, G_m \in \mathcal{G}$, and consider each isolated component as a *meta cluster* of WPN messages. Notice that this leads us to groups of WPN clusters that are related to each other because they collectively share common landing page domains. Figure 3.5 visually shows two examples of such meta clusters.

3.6.4 Identifying Suspicious WPN Clusters

In this section, we provide a systematic way to identify “potentially malicious” (also referred to as suspicious) WPN ads in addition to the malicious WPN ads that were found in the previous section. Further, this helps to drastically reduce the manual effort required to analyze all WPNs.

First, we consider those WPNs as *suspicious* whose landing domain names are associated with full URLs that have been previously labeled as malicious by GSB or VT. Let $G_i \in \mathcal{G}$ be a meta cluster that includes one or more of such known malicious URLs. Then, we label WPN clusters in G_i (and thus all WPN messages in those clusters) as suspicious, unless they were previously labeled as malicious according to the process described in Section 3.6.2.

In addition, given a meta cluster G_i , if it contains at least one WPN cluster $w_j \in W$ that we previously identified as a *WPN ad campaign* (see Section 3.6.1), we consider all WPN messages contained in the WPN clusters within G_i as WPN ads. This is because those WPN messages point to domain names related to WPN-based advertising, since they are linked to one or more WPN ad clusters, and thus are highly likely WPN-based ad messages themselves.

Apart from using their association with malicious URLs, to identify additional suspicious WPN ad campaigns that were not previously labeled based on the process described above and in Section 3.6.2, we proceed as follows. We notice that ad networks such as Google Ads and Bing Ads recommend advertisers not to promote the same product, brand, or similar customer experience by redirecting users to multiple destination websites. Violating this policy is referred to as *Abuse of Ad Network* by Google Ads[172] and *Duplicate Ads* by Bing Ads[173].

Besides the fact that these practices do not conform to advertising policies, malicious advertisers often prefer hosting similar malicious content on multiple domains to evade detection and to continue the

attack even if one of their domains is blocklisted. We then identify such practices and label the related meta-clusters that lead to multiple different landing domains as *suspicious*, to trigger further (manual) analysis to confirm if they are malicious. We provide detailed measurements on the identified suspicious WPN ads in Section 3.7.3.

Manual Verification: To validate the malicious URLs detected by URL blocklists, and to measure the number of suspicious WPN clusters that are in fact malicious, we manually analyze all malicious and suspicious WPN clusters discovered by our data analysis module. During manual analysis, we consider multiple factors to determine if the WPNs are indeed malicious. Once we manually confirm that a WPN cluster is malicious, we add it to a *manual blocklist*, which we then use to inform the measurement results presented in Section 3.7.

Some of the factors considered during manual blocklisting are as follows. We recognize a WPN message as malicious if it meets a combination of the following conditions: (1) leads to a landing page visually similar to a known malicious page (as determined by GSB and VT); (2) contains the same WPN message as a known malicious WPN message, but leads to a different product/site (i.e., a different landing page); (3) includes message content that is highly likely malicious or leads to a page with likely malicious content such as rewards that clearly seem too good to be true, as is typical of survey scams [174], false financial alerts, etc.; or (4) leads to a landing page that shares several domain-related properties with known malicious sites, such as IP address, registrant, similar domain name, etc. We further discuss our manual analysis process by presenting some examples in Section 3.7.3.

3.7 Measuring WPN Ads in the Wild

In this section, we report measurements on the usage of WPNs as an ad delivery platform, and provide insights into the malicious use of WPN ads.

3.7.1 Data Collection Setup

We first describe PushAdMiner’s setup for harvesting in-the-wild WPN messages for both desktop and mobile environments. Because our internal browser instrumentation are implemented by extending the browser code provided by [5], our data collection process leverages Chromium’s code base version 64.0.3282.204, which we built for both Linux and Android environments.

Seeding the WPN Crawler

Our main goal is to setup the WPN data collection system to maximize our chances of collecting WPN-based ads, so that we can measure their properties and discover abuse. To this end, we rely on the `publicwww.com` code search engine. Compared to simply crawling popular websites, this allowed us to more efficiently discover a diverse set of URLs (both popular and unpopular) that use push notifications. We first identify a *seed* set of popular advertisement networks that support push notification advertisements. Specifically, by manually performing online searches and reading articles and forums dedicated

to advertising online, we manually discovered 15 popular ad networks that provide push advertisement services. We registered an account with 9 of these ad networks to obtain the JavaScript code that needs to be embedded in an ad-publishing websites to include the ad networks' SWs that will control their push notifications requests. We then used the `publicwww.com` code search engine to find websites (i.e., URLs) that likely embed the identified ad network's code. The code search engine (`publicwww.com`) itself provides a list of *ad networks to search keywords* mappings [175], which we leveraged to obtain URLs of websites that likely embedded ads from the remaining 6 seed ad networks of interest, without the need to register an account with those networks as well. The list of 15 seed ad networks we discovered this way is shown in Table 3.1.

In addition to the search keywords related to the 15 ad networks mentioned above, we further used *generic keywords* that potentially indicate that a web page employs push notifications, such as `Notification`, `RequestPermission`, `pushmanagersubscribe`, a 'push' parameter value for `addEventListener`, etc. Overall we obtained a list of 19 code search keywords to be used for searches on `publicwww.com` that are likely to lead to URLs that make use of push notifications. These keywords allow us to explore a large set of web pages that goes beyond websites where ads from the seed ad networks are most likely hosted. Table 3.1 provides a count of the HTTPS URLs found via `publicwww.com` for each ad network and generic WPN-related keyword.

As a result of the above search, we were able to gather a total of 87,622 HTTPS URLs that could potentially register service workers and publish WPN ads (notice that SWs can only be registered by HTTPS origins). These URLs were hosted on 82,566 distinct second-level domain names. We use this set of URLs as seed for PushAdMiner's WPN message crawlers. Notice that there is no guarantee that the sites we crawl will in fact lead us to registering for and receiving WPN-based ads. Rather, only a subset of these web pages will actually request notification permissions. Therefore, we visited each URL and retained only those that actually make a request for a notification permission. Then, we used PushAdMiner to automatically grant notification permission requests on those URLs.

As shown in the last column of Table 3.1, overall we identified 5,849 URLs hosted on 5,697 distinct second-level domains that issued a notification permission request. Of these 5,697 domains, 2,040 (36%) ranked within the `Alexa.com` top one million most popular websites, and 1,383 of them actually sent us notifications during our experiment period. Table 3.2 shows a detailed breakdown of how the 2,040 domains ranked among the `Alexa.com` most popular websites. These results show that our process for collecting WPNs is not limited to unpopular, "low-tier" websites. In addition, they show that even highly ranked websites can push WPN ads, some of which resulted to be malicious.

Collecting WPNs in a Desktop Environment

To automatically harvest WPN advertisements at a large scale, we leverage Docker containers [176] to launch several parallel instances of our instrumented browser-based crawlers. During our experiments, we used four different Ubuntu 16.04 Linux machines having between 8 to 32 CPU cores and 64 to 128 GB of memory each, running a total of 20 to 50 Docker sessions in parallel at a time. For each seed URL that issues a notification permission request, we start the monitoring phase described below.

Every time we visit a URL, we wait 5 minutes to make sure the website has sufficient time to present our browser with a notification request. Once a permission request is received and automatically granted by a browser instance, a SW is registered. If a SW was registered, we keep the related Docker container alive for an additional 15 minutes, to allow the browser to receive the first (or more than one) WPN message from the visited URL. To select this 15 minutes threshold, we first performed pilot experiments with much longer waiting times (up to 96 hours) for a large subset of URLs (1,425 URLs, to be precise), and observed that 98% of them sent their first notification within 15 minutes of when the permission was first granted.

Given a container, and therefore a browser instance that granted notification permissions to a specific URL, after the first 15 minutes of its life we suspend the container to free up resources for instantiating a new container that will visit a new URL. However, we periodically resume suspended containers to see if they will receive additional notifications, which are queued in the FCM and sent to the browser as it comes back online.

Table 3.1: URLs and Notification Permission Request counts

Ad Network	URLs	NPRs
Ad-Maven	49,769	1,168
PushCrew	15,177	427
OneSignal	11,317	2,933
PopAds	1,582	73
PushEngage	796	215
iZooto	676	278
PubMatic	647	7
PropellerAds	335	9
Criteo	154	5
AdsTerra	115	2
AirPush	52	0
HillTopAds	21	3
RichPush	12	0
AdCash	10	0
PushMonetization	9	5
Generic Keywords	URLs	NPRs
NotificationrequestPermission	3,965	538
pushmanagersubscribe	2,667	158
addEventListener('Push')	263	9
adsblockpushcom	55	19
Total	87,622	5,849

Table 3.2: URL rankings on Alexa's top 1M sites

Alexa Ranking	Request Notifications	Sent Notifications	WPN Ads (# of Sites)	Malicious WPN Ads (# of Sites)
< 5k	37	21	6	3
< 10k	93	46	8	5
< 50k	317	151	21	11
< 100k	515	230	33	14
< 500k	1494	610	90	33
< 1M	2040	1383	135	60

Collecting WPNs in Mobile Environment

During our study, we found that WPN messages sent to mobile devices tended to be somewhat different than the ones collected by desktop browsers, in that they were more tailored to mobile users. In particular, malicious mobile WPN messages included fake *missed call* notifications, fake *amber alerts*, “spoofed” *Gmail* or *WhatsApp* notifications, fake *FedEx* notifications, etc. In addition, we found that these malicious messages were much more likely to appear on real Android devices, rather than emulated environments (likely due to some form of emulator detection). Therefore, to automatically collect mobile WPN messages we instrumented a real mobile device. Specifically, we used a Google Nexus 5 device with 2 GB of RAM and a 1080×1920 pixels display. The Android version we used was aosp_shamu-userdebug 7.1.1 N6F26Y.

As we attempted to scale our PushAdMiner’s mobile WPN crawlers on a real device, we identified two challenges. First, Docker or other container techniques do not support Android, and therefore we cannot easily visit multiple URLs in parallel with isolated browsing sessions. Second, we considered to use app cloning techniques [177] to open multiple browser instances separately in isolated execution environments. However, the limited computing power of mobile device restricted us to scale up and visit a large number of URLs simultaneously. Therefore, we decided to open multiple URLs in one chromium app but in separate tabs.

Table 3.3: Measurement Results of Data analysis Module

	WPNs with Landing Pages	WPN Ad Campaigns	WPN Ads	Malicious WPN Ad Campaigns	Malicious WPN Ads
Desktop	9,570	572	5143	318	2615
Mobile	2,692				
Total	12,262				

3.7.2 WPN Messages Dataset

We start with the 5,849 initial URLs that we collected as explained in Section 3.7.1, over 5,697 distinct second-level domain names. By clicking on WPN messages issued by these initial URLs, we collect an additional 10,898 URLs across 2,269 distinct second-level domains. When visited, many of these additional URLs presented our browser with a notification request, which our crawler automatically granted. This brought us to a total of 7,951 URLs that registered a SW with Push permission and were therefore able to push notifications to our instrumented browser instances over time.

During the course of about two months (September and October 2019), we were able to collect a total of 21,541 push notification messages, including 12,441 notifications for the desktop environment and 9,100 for the mobile environment. PushAdMiner interacted with each of these WPN messages. However, not

all automated clicks on notification boxes led to a separate landing page. In addition, some landing pages appeared to cause a crash in the browser's tab (but not the browser) in which they rendered, preventing us from collecting detailed information on those pages (this was likely due to the fact that our instrumented Chromium browser is not based on the most recent stable code base). We filtered out these notifications, leaving us with 12,262 WPN messages (9,570 on desktop and 2,692 on mobile) that when clicked on lead to a valid landing page. We then used this final set of WPN messages for clustering process (refer Section 3.6).

3.7.3 Data Analysis Results

Summary of findings: Table 3.3 summarizes the overall results of our analysis process. From the 12,262 WPN messages mentioned above, PushAdMiner identified 572 WPN ad campaigns and a total of 5,143 WPN ads related to these campaigns. Moreover, PushAdMiner identified 51% of all WPN ads as malicious. Specifically, PushAdMiner found 318 (out of 572) campaigns to be malicious; in aggregate, these malicious campaigns included 2,615 WPN ads.

This is quite a staggering result, in that it appears that ad networks that provide WPN ad services are heavily abused to distribute malicious content. Later, in Section 3.7.4, we also show that ad blockers are ineffective at blocking such ads, which is an additional cause of concern. In the following sections, we discuss the clustering and labeling results in more detail.

WPN Clusters and Ad Campaigns

As discussed in section 3.6.1, we cluster the collected WPN messages based on their message content and landing page information. After clustering 12,262 WPN messages that led to a valid landing page, we obtained 8,780 WPN clusters, of which 7,731 were singleton clusters containing only one element (i.e., only one WPN message). Of the remaining non-singleton clusters, 572 were labeled as WPN ad campaigns, according to the process described in Section 3.6. In aggregate, these WPN ad campaigns pushed 3,213 WPN ad messages to our browsers, during a period of about two months.

Figure 3.4 provides some concrete examples of WPN clusters. In both WPN-C1 and WPN-C2, the respective WPNs were pushed from multiple sources (i.e., multiple second-level domain names), as also shown in Figure 3.4. WPN-C3 included 4 identical WPN messages pushed by a single source website, a bank, alerting users on their loan offers. These messages appear to be legitimate, and led back to the site that pushed them. WPN-C4 is an example of WPN message isolated into a singleton cluster. According to the definition provided in Section 3.6, we label WPN-C1 and WPN-C2 as WPN ad campaigns, because the WPNs in each of the clusters deliver very similar (or the same) message promoting very similar products from multiple sources. However, WPN-C3 and WPN-C4 do not meet the definition and are thus not labeled as WPN ad campaign.

Malicious WPN Ad Campaigns

As described in Section 3.6.2, we submit landing page URLs related to all WPN messages to GSB [95] and VT [156]. On our initial scan, less than 1% of the URLs were detected as malicious by GSB or VT, in



Figure 3.4: Examples of WPN clusters

aggregate. For instance, initially VT flagged 108 landing page URLs as malicious, of which 88 were related to WPN messages labeled by our system as belonging to ad campaigns. Notice that for VT we consider a URL as malicious if at least one of the URL detection engine reports it as malicious, and later manually review all results to filter out possible false positives. After one month, we submitted the same set of URLs once again, and we found that 1,388 URLs (11.31%) were detected by VT, though GSB still only flagged 1% of them.

PushAdMiner relies on label propagation to label WPN messages and clusters as malicious, based on results from VT and GSB, as explained in Section 3.6.2. To limit the chances of amplifying possible false positives from VT and GSB, we manually verified all 1,388 URLs to check whether they actually led to malicious content. We were able to confirm that 96.8% of them indeed appeared to be malicious. Of the remaining 44 URLs that we could not confirm as malicious, 13 were found to belong to popular benign domains such as `bing.com`, `kbb.com`, `tophatter.com`, etc.; 24 URLs were related to unpopular blog/news sites; 3 led to adult websites; and 4 led to websites hosting non-English content that we could

Table 3.4: Measurement Results at Stages of Clustering

	# clusters	# ad-related clusters	# WPN ads	# known malicious ads	# additional malicious ads
After WPN Clustering	8780	572	3213	758	367
After Meta Clustering	2046	224	1930	210	1280
		Total:	5143	968	1647

not verify. Given that these sites may be benign, since we do not have all the information VT and GSB had to label them as malicious we take a conservative stance and remove the malicious label from them. Accordingly, we label 1,344 WPNs as *known malicious*. Among them, 758 WPNs were part of 572 WPN clusters that we previously classified as ad campaigns (see Section 3.7.3). The remaining 586 WPN messages that led to malicious landing pages were not immediately found to belong to WPN ad clusters, as they formed separate small clusters. We will determine whether they are related to WPN campaigns later, in Section 3.7.3) after the meta-clustering step (explained in Section 3.6.3).

By using a “guilty by association” label propagation policy, as explained in Section 3.6.2, we label WPN ad campaigns as malicious if they include at least one *known malicious* WPN (remember that this policy is justified by the close similarity in content and landing page URL path between messages in the same cluster). This yielded 152 (out of 572) *malicious* WPN ad campaigns, which overall included 376 WPN (or more precisely their landing pages) that GSB or VT missed to detect as malicious. After manually inspecting these 376 WPN ads, we were able to confirm that 367 of them are indeed malicious ads that lead to survey scams, phishing pages, scareware, fake alerts, social media scams, etc. We were not able to confirm the maliciousness of the remaining 9 ads (i.e., 2.4%) that led to different pages that welcome/thank the user for subscribing to the notification all hosted on the same IP address. The take away from the above discussion is that, using our WPN clustering approach, we were able to increase the number of confirmed malicious WPN ads from 758 to 1,125 (i.e., 758 plus 367), which represents an increase of about 50% as summarized in Table 3.4, first row.

Referring to the examples provided in Figure 3.4, in cluster WPN-C1, 35 out of the 40 WPNs were labeled as *known malicious* WPNs, according to VT. However, PushAdMiner labeled this entire cluster as *malicious*. After manually inspecting all 40 messages, we confirmed that the remaining 5 messages in the cluster were indeed related to the 35 malicious sweepstakes/survey scam ads.

Finding Suspicious Ads

So far, we have leveraged the labels provided by VT and GSB to identify malicious WPN ads, and label WPN ad campaigns. Unfortunately, both URL blocklists suffer from significant false negatives, when it comes to detecting malicious landing pages reached from WPN ads. As an example, consider cluster

WPN-C2, which PushAdMiner identifies as an ad campaign. This cluster contains 12 WPNs; none of which were labeled as *known malicious* according to VT. However, PushAdMiner flags this cluster as suspicious since it contains *duplicate ads* (see definition in Section 3.6.4) and via manual inspection we found that the WPN messages in this cluster display fake PayPal alerts that lead users to survey scam pages; therefore, we manually label the entire WPN-C2 cluster as malicious. This example demonstrates the gaps left by current URL blocklisting services, and how ineffective they could be if they were used to detect and block malicious ad notifications. Below we discuss how we use the meta-clustering approach explained in Section 3.6.4 to automatically identify and label more of such cases.

As described in Section 3.6.3, we apply a meta-clustering method to group WPN clusters that may relate to each other, as they share common landing page domains. To this end, we create a bipartite graph $\mathcal{G} = (W, D, E)$, here W is the set of all 8,780 WPN clusters we previously obtained, and D is the set of all 2,177 distinct landing page domains pointed to by WPN ads that we were able to record. By identifying and separating the connected components in this bipartite graph, we identify 2,046 WPN meta clusters. Of these, 224 contain a mix of WPN clusters that we previously labeled as *ad campaign* and other non-campaign WPN clusters. We then label all WPN messages contained in these 224 ad-related meta-cluster as *WPN ads*, thus increasing the number of WPN ads identified so far from 3,213 to 5,143.

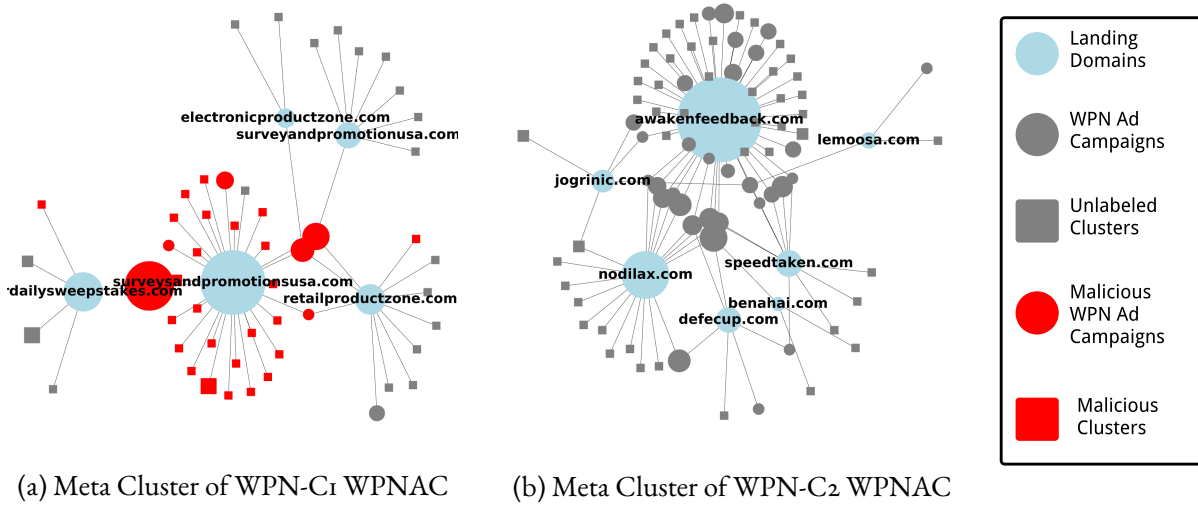


Figure 3.5: Graphical Representation Examples of Meta Clusters

Figure 3.5 provides two examples of meta clusters. Figure 3.5a contains as a node cluster WPN-C1 from Figure 3.4, as well as other 6 related WPN ad campaigns that are likely orchestrated by the same operators. This meta-cluster contains many *known malicious* WPN ad campaigns and WPN clusters, and therefore, we label all the WPN clusters in the meta cluster as suspicious. By manual inspection, we verified that all domains involved in this meta cluster host visually similar malicious pages (e.g., online survey scam pages).

Figure 3.5b shows another example of meta-cluster, which includes cluster WPN-C2 from Figure 3.4 as a node, along with 30 other related WPN ad clusters. In this meta-cluster, none of the WPN clusters

(i.e., the nodes) were initially labeled as malicious by either VT or GSB. However, we manually inspected all landing pages pointed to by WPN messages including in the meta cluster, and we were able to confirm that these are indeed malicious, in that they display fake PayPal messages and alerts that lead users to survey scams and likely phishing-related pages.

Next, we consider all yet to be labeled WPN messages in a WPN meta cluster as suspicious if the meta cluster contains at least one *malicious* WPN cluster or if it contains *duplicate ad domains*, as defined in Section 3.6.4. Out of the 572 WPN ad campaigns identified earlier, we found that 255 of them contained *duplicate ad domains*. Accordingly, we were able to label a total of 287 out of 2,046 WPN meta clusters as *suspicious*. Further, we identified 166 (out of 572) additional WPN ad campaigns, which were not previously labeled malicious in the previous step, as *suspicious*. Overall, this translates into 1,479 suspicious WPN ads, as shown in Table 3.4. Following our manual verification process, we confirmed 1,280 (86.5%) of these ads as malicious. The remaining 199 WPN ads were flagged by PushAdMiner because they were related to *duplicate ad domains*. Of these, 166 were alerts related to job postings and led to similar pages on multiple domains listing the same job; 23 led to multiple sites that hosted content related to the horoscope; 4 led to adult websites; and 6 were subscription welcome/thank you notifications. Notice that while these 199 WPN messages may be benign, PushAdMiner helped us identify and characterize a large number of additional WPN ads that are in fact malicious and were not identified by URL blocklists such as VT or GSB. However, our current system is not designed to be an automatic malicious WPN ad detection system. In our future work, we plan to leverage the lessons learned from the measurement results obtained in this paper to investigate how malicious WPN messages can be accurately detected and blocked in real time.

Singleton Clusters: Our tight first-stage clustering yielded 7,731 singleton clusters. Of these, 6,876 were found to share landing domains with WPNs in non-singleton clusters. After meta clustering, we were then left with 855 singleton clusters. By manually inspecting a sample of 200 singleton clusters, we found them to be a mix of simple alerts and spurious suspicious ads. Table 3.5 shows a few example of the text and domains related to the analyzed singleton clusters.

Table 3.5: Examples of Singleton Clusters

Title	Body	Domains (S)-Source (L)-Landing
TechNewsGadget	189 Fortnite Wants To Add Bot Players	technewsgadget.net (S)(L)
Congrats Walmart User!!	(1) Reward Waiting!	healthdreamstoday.com(S) besthealthlife.com (L)
The Mattest Blackest Liner EVER !!	Our new obsession!	hudabeauty.com (S)(L)
Coca Cola is looking for YOU ??	No experience required Training Provided!	eblog.network (S)(L)
Hire Local Service Professionals For All Your Needs	vconnect.com is your one stop destination for local services.	m.vconnect.com (S)(L)
FOX NEWS	Lose 45lbs In 4 Weeks! No Exercise!	nodilax.com(S) women-lifestyledaily.com (L)

Additional recent measurements: To measure the prevalence of WPN ads at a later point in time, compared to our initial measurements, we collected an additional and more recent batch of data for 5 days between April 4th, 2020 and April 9th, 2020. We revisited 300 websites randomly chosen from our previous datasets, 35 of which sent us 305 notifications over 5 days. Of these 305 notifications, PushAdMiner labeled 198 WPN ads and flagged 48 of them as malicious, which we also verified via manual analysis. After checking the corresponding landing page URLs on VirusTotal, only 15 of them were flagged as malicious, confirming again that WPN-based threats often remain undetected by current defenses.

3.7.4 Push Ad Networks and Blocking

Figure 3.6 shows the distribution of WPN ads, including malicious ones, per ad network. As it can be seen, many of the ad networks we considered in our measurements are abused to distribute malicious WPN ads.

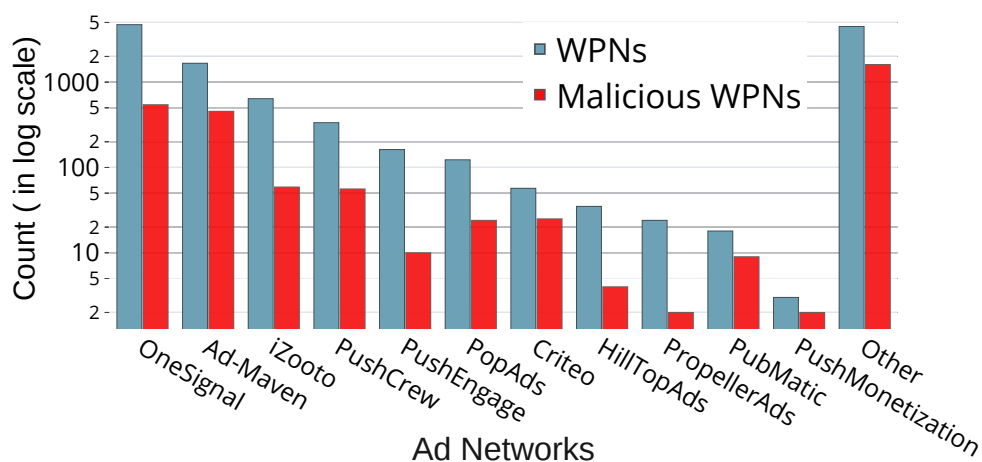


Figure 3.6: Distribution of WPNs w.r.t. Ad Networks

We also investigated whether ad blocker extensions used by desktop browsers may be effective against WPN ads.

For instance, we checked the URLs of SW scripts against the Easylist filter rules[178] used by most popular ad blockers. Furthermore, we installed two highly popular ad blocker extensions in our Chromium browser and checked its blocking capability. As shown in Table 3.6, both ad blocking mechanisms couldn't block the requests issued by the installed SW that were related to WPN ads, even though Easylist was able to filter a small number (less than 2%) of such network requests. This shows that existing methods were not sufficient to mitigate WPN-based ads, including malicious ones.

While working on this study, a new feature was introduced in Chrome [179] in February, 2020 to prevent the abuse of WPNs. This feature focuses on blocking notification permission prompts from

websites that have a low notifications opt-in rate. To test the effect of this feature on our dataset, we used the latest Chrome version (Chrome 80) to revisit 300 randomly chosen websites that had previously requested notifications. Since our Chromium code instrumentation could not be easily ported to this latest version of the browser, we performed detailed manual analysis. We found that all of the websites we visited were still able to request notification permissions without getting blocked. It is possible that this new Chrome feature will be able to block abusive WPNs in the future, after more training data is collected. However, it is unclear whether and to what extent it will be effective in blocking WPN-based malicious ads.

Table 3.6: Results on Existing Ad Blockers

	No. of Blocked URLs		
	Service Scripts	Worker	Service Worker Requests
Easylist Blocklist	0 out of 1187		132 out of 8031
AdBlockPlus	0 out of 884		0 out of 7276
AdGuard	0 out of 895		0 out of 7520

3.8 Related Work

In this section, we discuss prior studies related to our work.

Service Workers and WPNs: Papadopoulos et al. [9] propose multiple attack techniques to register malicious SWs that can control a victim’s web browser (e.g., steal computation power for performing nefarious activities). There exist other studies [41], [42] that demonstrate similar attacks on Web Workers. Lee et al. [154] study the risks posed by PWAs. For instance, they collect SW scripts from top-ranked website homepages and analyze the risks for phishing due to the use of brand logos in push notifications. They also found a number of push-based phishing attacks that exploit WhatsApp and YouTube icons. Furthermore, they also identify security flaws in the design of browsers and third-party push notification libraries. Other works related to Progressive Web Apps[38]–[40] focus on the performance of PWAs in multiple environment and their energy usage.

Our study is different, because we focus on measuring the use of WPN messages as an ad-delivery platform in general, and on measuring abuse in WPN ads. To that end, we design and implement an automated system to subscribe to, collect, interact with, and analyze in-the-wild WPN messages on both desktop and mobile platforms.

Analyzing Online Ads: A line of studies [81]–[83] is dedicated to investigate the online advertisement ecosystem. They trace the information passed between ad exchanges and analyze the revenue collected using ads. Another body of studies [84]–[92], [180] focus on identifying *traditional* malicious web advertisements and ad campaigns in both desktop and mobile environments. Similarly, a recent study **Secma** reports observations of social engineering attack campaigns delivered through traditional web-based ad-

vertisements. Our work is different, because we specifically focus on ads and malicious ad campaigns delivered through WPN messages.

AdGraph [181] is an advanced machine learning-based system that focuses on identifying online ads embedded in web pages. AdGraph’s instrumented browser collects fine-grained details about relationships between DOM nodes. A graph generated from those details is fed into a Random Forest classifier to detect ADS vs. NON-ADS elements. Upon testing the effects of AdGraph’s open-source implementation on ads delivered via WPNs, we observed that their system collects detailed information from a web page’s context but does not appear to record or analyze information related to SWs and WPNs. Our work is different, in that we focus specifically on collecting WPN-based ads and measuring the prevalence of malicious ads within them.

Online Scams and Ad Blockers: A number of approaches [182]–[184] have been proposed to study the effectiveness of existing ad block techniques as well as the various counter-measures used by ad providers to circumvent ad blockers. Other studies [155], [174], [185] discuss various online scam techniques and the prevalence of Internet fraud. Although PushAdMiner does not focus on blocking malicious ads or online scams, we demonstrate that existing ad blocking techniques are not effective at defending against malicious WPN ads.

3.9 Discussion and Limitations

Blocking Malicious WPN Ads: In Section 3.7.4, we showed that existing popular ad blockers and filtering rules were not effective at mitigating WPN ads, due to the limited visibility that extensions had on SW activities. This appears to have been due to Chromium bugs that have since been fixed (or are in the process of being fixed) [186]. However, even though extensions may now be able to observe and block SW requests, it is unclear if this will be sufficient to block WPN ads or if other evasion mechanisms can be implemented by push ad networks.

A browser extension called AdBlaster [187] specifically claims to block WPN ads. However, AdBlaster simply disables *all* push notifications, including benign non-ad ones, which can obviously disrupt legitimate uses of WPN messages. Although our PushAdMiner system is not currently designed to detect malicious WPN ads in a fully automated way, we believe that the results from our study provide useful insights into the malicious use of WPN ads that may encourage and help the security research community to build better defenses against the abuse of web push notifications.

Double Permissions: In this study, we observed some websites do not directly ask for permission to send notifications to the browser. Rather, they first create a dynamic JavaScript-based prompt that mimics a browser permission request. Popularly referred to as *double permission*, this mechanism is used by some websites to avoid being permanently blocked from requesting notification permissions in the future (which would happen if the user clicks on the “Block” button), or to bypass other policy restrictions. We observed *double permission* to gain popularity only recently, months after our initial data collection. To check its effects on our dataset, we ran an experiment on a random sample of 200 URLs that previously requested permission directly. We found that only around 1/4th (49 out of 200 URLs that we manually

checked) switched to using double permission. Furthermore, this does not impact our analysis on the abuse of WPNs. For instance, PushAdMiner could be instrumented to bypass this mechanism by simply granting notification permission to *all* visited sites, regardless of whether they request it or not. Because double permission dialog boxes are typically programmed to appear only when the website doesn't already have notification permission, they could be avoided altogether.

Evading Crawling Detection : Websites may attempt to detect browser automation to avoid being crawled. To mitigate this issue, we built our crawlers using Puppeteer. Unlike other tools, Puppeteer (developed by the Chromium team) leverages DevTools and is less prone to fingerprinting. During our experiments we also observe that a few ad networks use cookies or other information to track the device or web browser across browsing sessions. To mitigate this and increase our chances of being presented with fresh notification permission requests from as many sources as possible, we created a separate Docker container for each URL we visited.

3.10 Conclusion

In this paper, we have studied how web push notifications (WPNs) are being used to deliver ads, and measured how many of these ads are used for malicious purposes. To enable this study, we developed a system called PushAdMiner, which allowed us to automatically collect and analyze 21,541 WPN messages across thousands of different websites. Among these, our system identified 572 WPN ad campaigns, for a total of 5,143 WPN-based ads, of which 51% are malicious. We also found that traditional ad-blockers and malicious URL filters are remarkably ineffective against WPN-based malicious ads, leaving a significant abuse vector unchecked.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and suggestions on how to improve this paper, and Prof. Paul Barford for serving as our shepherd. This material is based in part upon work supported by the National Science Foundation (NSF) under grants No. 1916500 and 1909856. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

CHAPTER 4

SOMETHING PHISHY: INVESTIGATING NEW TRAITS IN MODERN PHISHING ATTACKS

4.1 Introduction

Phishing attacks thrive despite a number of existing counter-measures and continue to be one of the most common and dangerous form of Social Engineering Attacks(SEAs). According to Internet Crime report by FBI [12], a record-breaking number of **6.9 million** phishing pages were created in the year 2020. Their prevalence could be attributed to the following major reasons, 1) availability of ready-made attacks and services 2) deployable by even novice attackers 3) cheap scaling infrastructure and, 4) availability of a number of platforms including email, social media etc., to initiate the attack. Nowadays, phishing sites have become more sophisticated by incorporating modern technologies such as browser fingerprinting, multifactor authentication, captcha, notifications etc. to gain users' trust and to evade automated detection. Similar to the age when people falsely believed HTTPS sites to be safe, one cannot rely on the presence of captchas and multifactor authentication to blindly trust the websites. Attackers design phishing sites with an intent to closely resemble legitimate sites in terms of both appearance and functionality while using a number of cloaking techniques to evade or delay detection. In the current world, data remains one of the most valuable assets and attackers are focussed on harvesting the most information possible from the victims. As a result, there are also a number of phishing campaigns that follow a multi-step phishing process to request more than login credentials from the users including personal contact information and payment information. Further, in order to avoid suspicion, attackers request different information using successive pages to successfully harvest more information.

A major reason behind these attacks being more sophisticated is to evade detection by anti-phishing bots and to reach more victims. In the paper [11], the authors demonstrate the effectiveness of few client-side cloaking techniques followed by phishing sites. More notably, one of such client-side cloaking techniques is "User Interaction" that describes different checks placed at the landing page of phishing sites

to verify if the visitor is indeed a Human. It has also been found that anti-phishing bots are ineffective in getting through these checks to the actual phishing pages that they are able to detect. Although phishing is a well-investigated topic [10], [11], [21]–[23], [28], number of research works that focus on such new traits of phishing methods is limited. Therefore, in this research, we focus on investigating such modern phishing pages that employ new technologies requiring user interaction and use multi-step campaigns to steal credentials from users. In particular, we aim to measure phishing sites’ properties and behavior while simulating user interactions and responses that would help us get past the checks placed by the phishing sites and reach the internal pages. As a result, we built a crawler that automatically interacts with the forms in any encountered page. This crawler analyzes the forms and inputs fake generated data that is required by the form and submits the data. The submitted form may lead to the page requesting more data in which case the crawler proceeds with processing the form. Our crawler also collect information related to resources requested by the form including tracker and captcha scripts, service workers installed by the page and permissions requested by the page. Then, we analyze the information collected by crawler and measure the different traits that are used by the phishing sites. We believe that our measurements on the modern phishing sites would provide insights into their detection and allow existing detection systems to extend their detection to modern phishing sites.

In summary, we make the following main contributions

- We build a custom crawler that infers the information requested by a page and interacts with the page simulating user interaction.
- We identify and log the presence and usage of new technologies such as service worker registration, captchas, multi-factor authentication.
- We report a number of traits followed by phishing sites to evade detection and demonstrate how our system Phish Traits is successful at overcoming their defense
- We use a combination of machine learning classification and clustering methods to identify various traits of the phishing campaigns.

4.1.1 Motivating Example

Figure 4.1 provides an example of one of the Phishing campaigns that we came across in the real world. It showcases the different pages targeting Paypal brand that are involved in the campaign and how the attackers request for more and more personal information after each step. It should be noted that the landing page in figure 4.1a doesn’t request any information from the user directly. Rather, it displays information to the user about certain restrictions that are placed in their account and that they can be removed by clicking the provided link and following the instructions. This represents a classic example of ‘Click-Through’ client-side cloaking (as defined in [11]) to evade being detected by anti-phishing bots.

PayPal

PayPal is constantly working to ensure security by regularly examining the accounts on your system. We recently reviewed your account, and we need more information to enable us to provide a secure service. Although we can not collect this information, access to sensitive account features remains limited. We apologize for the inconvenience and hope to restore access to your account as soon as possible.

Why is my account access limited?

Access to your account has been limited for the following reasons:

- We have determined that someone may have tried to access your PayPal account without your consent. For your protection, we have limited access to your account. To lift this restriction, follow the procedure described in [Resolution Center](#).

(The folder name is for this reason PP-762-989-584.)

How do I regain access to my account?

Please refer to the [Liaison Manager](#), and follow the procedure for the removal of restrictions.

Failure to complete all phases of a normal checklist immediately restore access to your account

(a)

PayPal

Access

1 confirm 2 Change Your Secrets Questions 3 Update Information 4 completed

Re-Enter your Email and Password to verify your PayPal Log-in.

Email address:

Password:

Submit

(b)

PayPal

Your security is our priority

1 confirm 2 Change Your Secrets Questions 3 Update Information 4 completed

Please take the time to select two secret questions and answers. (We ask these questions, if you forget your password.)

Secret Question 1
- Select a question -

Answer 1

Secret Question 2
- Select a question -

Answer 2

Submit

(c)

PayPal

Update your information

1 confirm 2 Change Your Secrets Questions 3 Update your information 4 completed

PayPal covers you against fraudulent use of your account.

Personal Information.

Enter the address where you receive billing statements for this card. In order to confirm your bank card number, the billing address must be the one displayed on your statements.

*Full Name:

*Date of birth: / /

*Address:

*Zip Code:

*City:

*Country:

Bank Information.

*Card Number:

*Expiration Date: /

*Card Verification Number:

3D Password:

Submit

(d)

PayPal

Congratulations

1 confirm 2 Change Your Secrets Questions 3 Update Information 4 completed

- PayPal is a payment easy and safe: the option to pay via PayPal inspires confidence and brings the buyer to confirm your purchase.
- The technology of PayPal currency transactions on the basis of hundreds of variables to identify potentially fraudulent activities.
- Our unique methods for bank account to PayPal members provide an additional level of security, which reduces risk and increases confidence for the entire community of PayPal.
- We use powerful encryption methods to protect sensitive information.

Submit

(e)

Figure 4.1: Example of Multi-Step Phishing Pages targeting Paypal (a) Page verifying Human Interaction (b) Form requesting login credentials (c) Form requesting security questions and answers (d) Form requesting contact information and card Information (e) Final confirmation form

In the next page, it simply requests email and password from the user. Once the user enters the login credentials, it then proceeds to obtain their security question and answers. On providing those answers, the page then attempts to request the user’s contact information including mobile number, address and card information. It should be noted that after each form is submitted, the site transfers the credentials and therefore, attackers would’ve successfully some information from the user even if a user gets suspicious in between filling all the pages and stops. Given that there is no existing work that focus on such multi-step phishing via interacting with the phishing sites, we aim to explore a number of characteristics related to each of these phishing pages by mimicking user interaction and bypassing the checks enforced by the sites as much as possible.

4.2 System Design

In this section, we provide an overview of how PhishTraits works, leaving a detailed description of the main system’s components to Sections 4.3 and 4.4. A high-level representation of the system is provided in Figure 4.2.

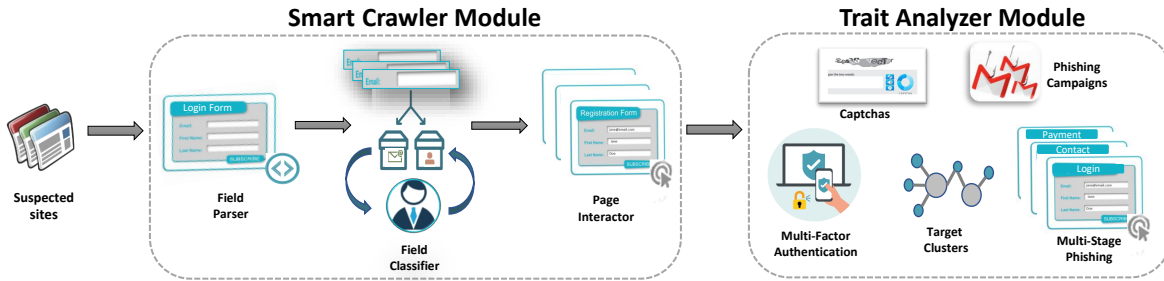


Figure 4.2: Overview of PhishTraits System

One of the most important goal of our system PhishTraits is to identify different traits of the Phishing websites and monitor how they react to user interaction. Overall, our system is made of 2 main components: (i) To simulate user interaction, we present an intelligent crawler and interactor that parses a page to identify the fields requested in a page, enters the identified information and interacts with the page to proceed to the next step (ii) an analyzer component that obtains the gathered data from the crawler to measure different characteristics of the phishing site that could be used cluster the pages into different campaigns including benign ones.

Firstly, We gather all of our data from OpenPhish [188], one of the largest repositories of phishing sites list. We use their API to collect possible Phishing URLs. This list is updated at an interval of every 5 minutes thus providing us access to fresh and alive Phishing Sites. Once we have this URL, we filter these URLs to keep track of the ones that are still alive and proceed to analyze these alive URLs. Further, to gain ground truth on the maliciousness of these URLs, we gather intel from Palo Alto Networks [189] to verify which among these URLs are flagged as malicious phishing sites by their Phishing Detection

system. In addition, we also use Google Safebrowsing API [95] to constantly check and track which of these URLs are flagged malicious. Next, we provide an overview of the crawler and analysis component of our system.

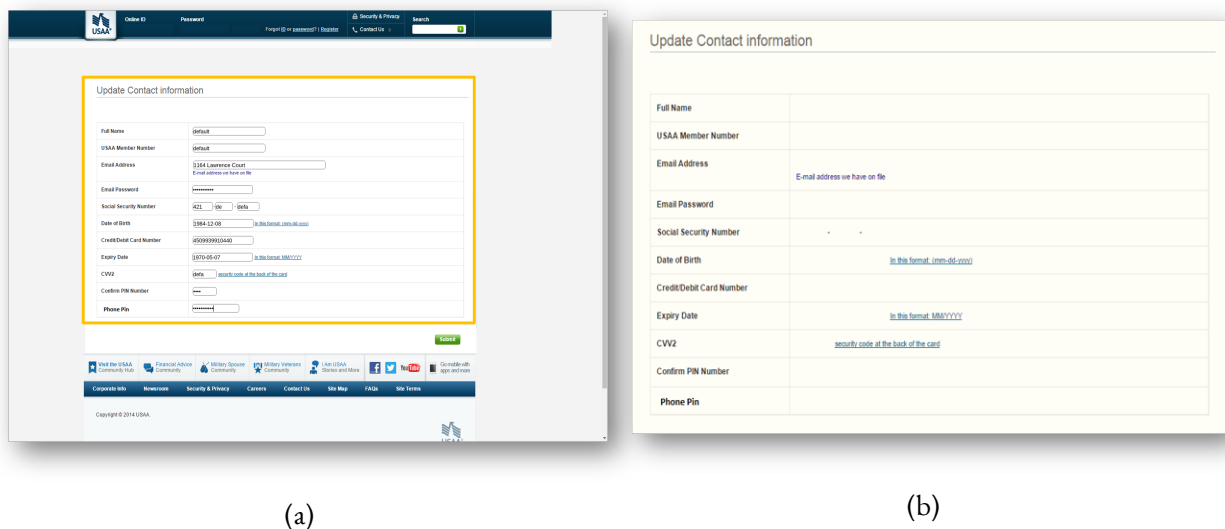


Figure 4.3: Example of Real-World Phishing Site using Image with Form Background to evade detection. (a) Phishing Site with a Form requesting Personal Information. (b) Background Image of the Form with the labels of the Input Fields

As mentioned in section 4.1.1, a phishing page is more wisely crafted nowadays to gain confidence of a victim and to bait them to provide more than just their login credentials. As a consequence, the attacker won't be able to convince a victim of the page's legitimacy by requesting all crucial information such as their address and banking information in a single page. Consecutively, some phishing attempts are composed of multiple web pages each requesting for specific type of information with basic level of authentication which has to be satisfied to proceed to the next page. Further, one of the techniques that the attackers use to avoid suspicion or detection is the use of an image as a background which contains the labels of the form fields and position input fields on the image next to the corresponding labels in the image as shown in Fig 4.3a and Fig 4.3b. Therefore, we have developed a custom crawler that automatically interacts with these phishing pages by identifying the requested information and emulating user's interaction with a web page as close as possible. To develop this custom crawler, we use Puppeteer [164] that is used for browser automation and event logging and included custom scripts to interact with the page. As discussed, owing to the complexity of Phishing pages, it isn't easy to identify the information requested by a web page. Therefore, we apply a combination of HTML parsing, Optical Character Recognition (OCR) and Machine Learning (more specifically Text Classification) to identify the various fields and the information requested by each field. In addition, the crawler collects other metrics related to the page such as the network requests and responses made by the page, if a page registers a Service Worker and if a page employs any advanced verification steps such as Captchas and Multi-Factor Authentication. We log all the information collected by the crawler for further analysis.

In the next phase, we take a deeper look at the data collected from the phishing sites and measure a number of metrics including the usage of different techniques used in real-world phishing pages. More notably, we provide measurements on phishing pages that require user interaction in order to reach the credential-harvesting page and measure different approaches employed for that purpose. We provide examples of phishing and scam pages that use these techniques in later section.

4.3 Data Collection Module: Smart Crawler

Owing to the complexity of the phishing pages, we follow multi-step approach to identify the information that a page requests. Since, our crawler is based on Puppeteer, on launching the browser, the program automatically downloads the latest compatible Chromium version. On visiting a page, We wait until the network is idle using the *networkidle2* option, capture a screenshot of the page and execute custom scripts in the current page to obtain the page title and the different DOM elements that are found in the page including their bounding boxes. At this point, we calculate a hash of a small set of DOM elements including only certain tags such as *DIV*, *INPUT*, *BUTTON* etc., while excluding any script and label elements. We refer to this hash as “Dom Hash” and use it to detect if the page has changed after performing any actions such as submit or button click.

4.3.1 Field Parser

In order to identify the requested information, we filter all the fields with *INPUT* tag from the obtained DOM nodes. Then, for each of these *INPUT* nodes we follow the below steps

1. The parser gathers information about the element such as the form it belongs to and their properties such as id, name, type, and placeholder. This information collected by the parser is observed to provide insight into the type of data the field requires from a user.
2. As mentioned earlier, in most cases, phishing sites do not follow naming standards and conventions. Hence, the parser additionally analyzes a field’s neighboring HTML tags such as div, span, labels etc. that provides information about the field itself. We identify neighboring elements by locating the elements to the left and top of the current element up to a certain distance.
3. The parser uses the screenshot saved earlier and slices the screenshot to get image slices of the areas surrounding the current element. Specifically, we obtain slices of inside the element, top of the element and left of the element as these are the locations where labels related to the field or the information required to be entered are usually found. Once we have the image slices, we process the image to enhance it and apply OCR technique on them.
4. The text collected from the above steps including id, Name, placeholder, texts captured through OCR and the neighboring elements are sent to the *Field Classifier* module that classifies the text

into predefined set of categories. It should be noted that in order to reduce unnecessary processing time, step 3 is performed only if step 1 and step 2 fails to identify the field's category.

5. Then, in order to decide the final category of the requested information, we use a polling technique on all the categories identified and assign the category with majority votes to the element. Once the final category is assigned, we generate values based on the category using the FAKER [190] Python library.

4.3.2 Field Classifier

This module is made up of a text-based classifier that helps classify a given text into one of the predefined categories of fields usually found in web pages such as name, email, password, phone, credit card number etc. We use a supervised classification process to classify the texts from parser into one of the known categories. In order to train the classifier, we manually label texts that were found in real-world phishing websites and used certain keywords and variations that are known to be used with the word. In order to classify the text, the following steps are applied

1. We preprocess the text collected by the parser to remove any unrelated characters or phrases such as any Unicode characters and stop words. Hence, we apply basic token cleaning to remove stop words and obtain all possible n-grams of the text. We then filter the n-grams to retain only those words that are part of a custom dictionary. This custom dictionary contains words that are commonly found in the web pages including all short terms for input fields and words that could be typos of commonly the field texts. The original text along with this specific set of n-grams are sent to the classifier during training and prediction.
2. Next, we use a pipeline of *CountVectorizer* [191] to create input vectors of the word counts and *SGDClassifier* [192] that performed better for our dataset as compared to *MultinomialNB classifier*
3. Since we have multiple classes, we use a multi-class classifier and train the classifier by providing the texts and labels from all categories. The training data for a particular category includes texts that are frequently seen to appear in various web pages that inform the users about the field. The trained classifier is pickled to be used for classifying texts later.
4. As soon as the parser module completes gathering data about a field, it sends the parsed text to this module. The *Field Classifier* then invokes the classifier to predict the label of the given text. The classifier pipeline returns the result with a confidence score. Then, we accept and return the label of the text only if the predicted confidence score is greater than or equal to a certain threshold (set to 80%). Otherwise, the field is labeled as "unknown".

The fields that are labeled unknown are gathered to be classified manually by experts and fed back to the classifier for the classifier to actively learn from the feedback. In order to help the experts label the

unknown data, we developed a web application that presents to the experts all the information gathered about the field and focuses the field on the page screenshot for better clarity. The input provided by the experts are recorded into the database and at frequent intervals, the manually classified data is used to retrain the classifier to improve the classifier's performance if similar cases are encountered at a later point.

4.3.3 Page Interactor

In this component, we use the information collected about fields from parser and their corresponding input values and submit the information for further processing. The ultimate goal of this crawler is to successfully navigate to the next page if any on the phishing site. Hence, we ensure that we use an exhaustive approach to submit the data successfully. Firstly, there a page could be designed in multiple ways to submit a data. For example, It could be using a submit button, styled button, using form submit, DIVs, canvas or Images styled as buttons. Therefore, we follow the below steps

1. Once we collect the data from the parser, we go through each of these submission methods one after the other
2. We use the "Dom Hash" calculated earlier to check if there has been any changes to the DOM. Note that in some sites, the data submission may not lead to a new URL. Rather, the same URL is loaded with the next content. Therefore, we use "Dom Hash" to verify this change
3. if the "Dom hash" was changed, then we consider the submission successful and proceed to crawl the next page starting from field parsing step
4. On the other hand, if there was no change to the "Dom Hash", then we use custom script to reset each input element and input its corresponding value using puppeteer. Then we continue with the next form of submission until a submission is successful at step 3 or until all different ways of submission are exhausted

At times, the data generated by the Faker library may not pass the validation on the page or the crawler could not successfully input the correct values. Therefore, at the most, we enable the crawler to parse through the same page for 3 times. If the crawler had already tried parsing the page for 3 times and hasn't seen any changes, then we stop our crawling at that point. It should also be noted that some phishing pages might not redirect the user to another page on submitting their credentials in which case the landing page of the phishing site would be parsed for at most 3 times. Overall, we also set a timeout for crawling a single site and if exceeded, will terminate the crawler.

4.3.4 Metadata Collection

In this section, we discuss additional data collected about the phishing sites that are analyzed by our system. The previous section explains the collection of information requested by the phishing pages and their related properties. In addition to that, we also collect other information related to the websites such as the website traffic, Service Worker, Captcha information. These are explained as follows

Network Resources

As and when a page is analyzed, our analyzer also captures information about the requests made by the page. Further, we save any redirections, the responses and save the response content and its hash for future analysis.

Service Worker

We also record if the phishing site registers a Service Worker. This information is non-trivial as overall most of the legitimate websites have started using Service Workers, and it could help to observe if this trend has been followed by attackers as well.

Captchas

Overall, captchas could be categorized into three categories based on their origin and behavior. First, we have the popularly used captchas such as Google's ReCaptcha, Hcaptcha etc., and are referred as "Known Captchas". Next, we have "Custom Captchas" that are used without using any third-party libraries but requires to be solved to pass through the validation. Lastly, some pages have been found to use captchas that don't require to be solved to pass the validation. Such captchas are used only to give the users the appearance of security and as a result gain their trust. We refer to those captchas as "Fake Captchas". Among these three, since the "Known Captchas" follow certain patterns, we execute custom scripts to identify the version and type of known captchas used and log their details such as 'Site-Key' information that is unique to a particular domain. For the remaining two types, we analyze the website screenshots offline to identify their presence.

4.4 Data Analysis Module: Trait Analyzer

In this section, we discuss some of the recent traits that the attackers use in phishing attacks and scams to reach more users and evade being detected by automated crawlers. Further, we explain how our system PhishTraits could be used to measure these trends and discuss possible ways of detecting and mitigating them. In this analysis, we aim to find important features that could be used to train an intelligent detection system that would lead to a more precise and quicker detection.

4.4.1 Custom HTML Designs

This trait refers to attackers structuring and styling phishing sites in such a way that it would prove harder for automated systems to infer the information requested by attack pages and block them. For example, as shown in Figure 4.3, some phishing sites use images containing the form details as form's background and position input elements over the image next to their corresponding labels. According to [193], a Phishing site used an image of a bank site as the background for the page while placing only 2 HTML elements on top of the image to obtain information. Such scenarios could be used to deter phishing

detectors that parse the HTML code to find indicators of the requested information. However, as our crawler combines HTML parsing and OCR techniques to identify the input elements in the page and the requested information, it ensures that such cases are analyzed as well.

In addition, in order to evade detectors that scan the resources requested by the page and match them with already flagged resources, some phishing sites apply custom modifications to the pages such as using an inverted image background, using sliced up logo images, using path and canvas tags to recreate logos and various elements of the page as shown in [194], [195]. As such, we do not analyze the images and logos requested by the page to decide if the page is malicious. Rather, we monitor the behavior of the pages and how the pages react to user's input and interactions.

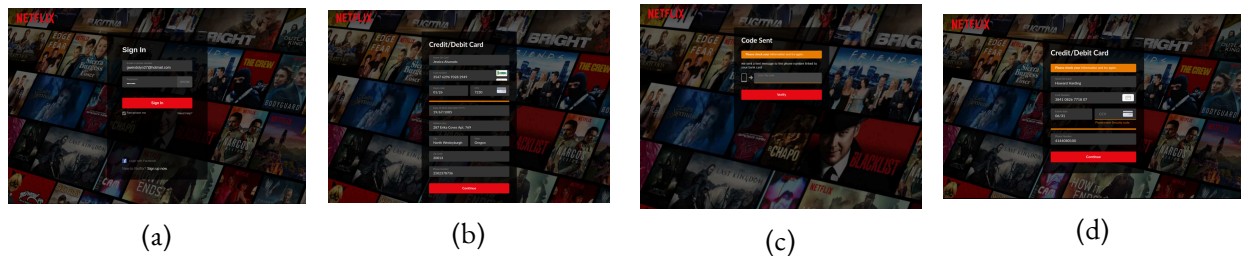


Figure 4.4: Example of Multi-Step Phishing Pages targeting Netflix and requesting Personal Credentials at each step (a) Phishing Form requesting login credentials. (b) Phishing Form requesting Card and contact information (c) Phishing Form requesting SMS Code (d) Phishing Form requesting Card information again

4.4.2 Multi-Step Credential Phishing

Most of the phishing attacks focus on obtaining just the login credentials of the users that is later used by the attacker to either access the victim's corresponding account or sold in the black market. However, nowadays, there exist some advanced phishing attacks where the user is taken through a series of forms requesting more and more information from the user on successful completion of one form. An example of such a case is shown in Figure 4.1. In some cases, even if the phishing campaign is only to collect the login information such as email and password, each of these fields are requested in different stages thus mimicking the process of legitimate websites. To the best of our knowledge, there hasn't been any work that focuses on analyzing such multi-stage phishing. Therefore, in this research, we aim to gather details on the information requested by each of the pages and report a number of measurements on what kind of information has been requested by the attackers at each stage. We identify a site as a multi-stage phishing site if either of the following conditions are met.

1. If more than one unique domains was visited per site during the crawl session
2. If more than one page-title was identified per site during the crawl session

3. If more than one “Dom Hash” was found for a site and more than one page containing different set of field categories were identified per site

Figure 4.4 shows another example of such multi-stage credential phishing. As shown in the figure, our crawler was able to decipher the information requested at each stage and successfully proceed to the consecutive stages. It should be noted that after collecting the login credentials, the site requested the user to enter their personal information followed by their account details. At the end, the user is asked to verify the details with the help of SMS code that was sent to their mobile phones. However, even though the crawler identified this field as a “Code” category and input a 6-digit code, on submission, an error message was displayed asking to enter the correct verification code. Then, we manually tested this page and tried different combinations of code digits. However, we still failed to pass the validation at this page. Therefore, we infer that this page required the code that was sent to a real phone number associated with the account details provided and that the site was verifying the information provided in the background.

In addition, such multi-step information could also be used to identify similar phishing campaigns that follow the same behavior by asking similar information at the particular stage. Therefore, we generate a feature that represents the field category that was requested at each stage for a site and group them using simple text grouping to form different group labels. We refer to this cluster as “FieldCategoriesGroup” For example, the feature generated for the example in Figure 4.1 is Email-Password && Question-Answer && Name-Date-Address-City-Country-Zip-Card-ExpDate-Year-CVV-Password and the feature for the example in Figure 4.4 is Email-Password && Name-Card-ExpDate-CCV-Date-City-Address-State-Zip-Phone && Zip && Name-Card-ExpDate-CCV-Phone. Then, on grouping each of these features would belong to different “FieldCategoriesGroup”.

4.4.3 Cloaking via User Interaction

The rise in the number of Phishing attacks has paved way to the rise in anti-phishing and detection tools most of which are designed to automatically filter phishing sites based on different metrics. In turn, attackers had to find new ways to evade automated bot-based detections and reach more users successfully. As a result, attackers started leveraging *Human Verification Techniques* that was introduced to prevent bots from crawling legitimate sites and protect users, to prevent their attacks from getting flagged. Further, the paper [70] measures the efficacy of Anti-Phishing entities against Human verification techniques such as Google ReCaptcha, alert-box and session-based/click-through evasion. They further conclude that only 7.4% of the phishing websites with such techniques were detected by the Anti-Phishing entities. If this problem isn’t tackled soon, it would pave way for the attackers to go unnoticed for a longer duration and as a result cause havoc to users and businesses.

Next, we explain how our system PhishTraits is not deterred by these techniques and if possible proceeds to next stages of the attacks. Each of the techniques is addressed in detail as follows.

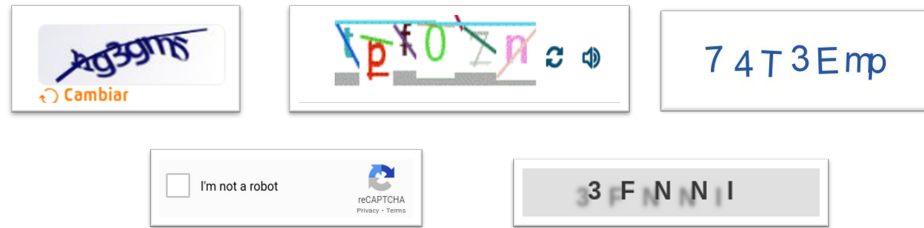


Figure 4.5: Examples of Captchas found in real Phishing sites

Alert Box

This technique refers to attack sites using alert boxes and overlays with button(s) that the user needs to first interact with to be led to the page that requests any information. It proves difficult in general to tackle this problem as it is hard to identify these custom alerts based on JavaScript execution. To address this particular issue, we designed our crawler to automatically click at random positions on the website using *Gremlin JS third-party library* for a limited time. This action is taken to ensure clearing any overlays and Javascript based alert boxes that may stop the execution of scripts. Further, we use the script to interact with browser-based alert boxes as well. We limit the number of clicks to 20 as more clicks might end up clicking a link that could navigate to a different page.

Session Based Evasion

This technique is similar to the previous technique in the sense that the page that actually requests information is hidden from the user until the user interacts with the initial page in the expected manner. Further, the site monitors session parameters that are set at the initial page and ensure that page requesting details can only be reached when the session parameters are valid. Some sites require the user to interact with links, buttons, elements styled to look as buttons to proceed to next page. In such a scenario, even if the random clicks fail to interact with the required element, our crawler by default interacts with the clickable elements in the page in an attempt to submit the page. As a result, our crawler would be more successful in navigating through this initial page and more details on this are provided in later section 4.5.

Captchas

Captchas were introduced to prevent external crawlers from interacting with legitimate websites and prevent attacks such as DoS(Denial of Service). Currently, there are few third party captcha providers that are popularly used in most websites. As discussed in the previous section, we broadly categorize captchas into 3 categories: “Known Captchas”, “Custom Captchas” and “Fake Captchas”. Owing to the fact that captchas are meant to prevent bots from interacting with the page and allow only humans to proceed, at this point it isn’t feasible to automatically bypass such captchas. Therefore, in our research, we only aim

to detect the presence of captchas in the pages and consider bypassing captchas as out of scope for this project.

In case of “Known Captchas”, our crawler executes custom scripts that look for indicators denoting the presence of certain types of captchas including all versions of Google ReCaptcha and hCaptcha. The details related to the captchas such as their site-key are logged for further analysis.

For the rest of the captchas, since they could be styled in different ways, it makes the process of detecting these captchas non-trivial such as using simple HTML parsing. Therefore, in order to detect these captchas, we leverage deep learning algorithms that are designed for object detection. Accordingly, we use Facebook’s library Detectron2 [196] that provides state-of-the-art detection and segmentation algorithms. Specifically, we use the “Faster R-CNN” model and train it on our custom dataset of captchas. Figure 4.5 shows few examples of captchas that are found in the wild. First, we collect an extensive collection of Captcha images from both public dataset and from the screenshots of few sites that we have crawled. Next, we split the images into different groups based on their similarity. For example, the 5 captcha images shown in Figure 4.5 would belong to 5 different subcategories of text-based captchas and visual-based captchas. Similarly, we also have an image collection of only buttons and logos. Next, we generate web page images by placing the captchas, buttons and logos in different patterns in a web page background (mostly white). These patterns are generated based on real-world examples. For each pattern, we generate n number of images to train and test the model. At the end, we use this model to detect if a web page has a captcha or not. At the moment, the captcha detection is performed offline, However, we plan to make it online (as part of our crawler) in the near future.

Using detectron2, the model could provide us if a site has a captcha or not. However, it isn’t sufficient to categorize the captcha as a “Custom Captcha” or “Fake Captcha”. Given the fact that “Fake Captchas” doesn’t really require the user to solve the captcha, our crawler might be able to successfully solve it by entering random information. Therefore, this information could be used in conjunction with the result of the captcha detection model to distinguish “Fake Captchas” from “Custom Captchas”.

4.5 Experiments and Results

In this section, we explain the experimentation setup to crawl through the pages and report the results of analysis discussed in section 4.4.

4.5.1 Crawler Setup

To automatically crawl phishing sites at large scale, we leverage Docker containers [176] to launch several parallel instances of our crawlers. During our experiments, we used Ubuntu 16.04 Linux machines having 128 GB of memory, running a total of 30 Docker sessions in parallel at a time. We set a timeout of 15 minutes for each of the session. Overall, we had crawled 13,293 websites in a period of 10 days between Nov 1st, 2021 and Nov 11th, 2021.

Table 4.1: Field Classifier results reporting Precision, Recall and F1-score

Category	Precision	Recall	F1-Score	Count
Address	1	1	1	8
Answer	1	0.95	0.97	19
CVV	1	0.60	0.75	5
Card	0.92	0.88	0.90	25
City	1	0.90	0.95	10
Country	1	1	1	4
Date	0.92	0.92	0.92	12
Day	1	1	1	6
Email	0.97	0.95	0.96	37
ExpDate	0.89	1	0.94	8
Gender	1	0.67	0.80	3
Identification	1	1	1	5
License	1	1	1	8
Month	1	0.94	0.97	17
Name	0.83	0.98	0.90	46
Password	0.93	0.98	0.95	43
Phone	1	0.94	0.97	16
Question	0.88	1	0.93	7
Routing Number	0	0	0	1
SSN	0.92	0.85	0.88	13
State	1	0.80	0.89	5
Year	1	1	1	5
Zip	0.96	1	0.98	24
search	1	0.86	0.92	7
Accuracy			0.94	334
Macro Avg.	0.93	0.88	0.90	334
Weighted Avg	0.94	0.94	0.94	334

4.5.2 Field Classifier Evaluation

In this section, we present the results on evaluating the classifier used to detect field categories based on the field text. We train this classifier with texts obtained from real-world phishing pages and label them manually by consulting the respective page’s screenshots. We use a similar method to create a testing set. On testing the pipeline model as described in section 4.3, we obtained an overall accuracy score of 94%. Table 4.1 presents the accuracy, F1 and Recall score obtained for each of the classes.

4.5.3 Multi-Stage Credential Phishing

In this section, we analyze the various multi-stage phishing campaigns crawled by our system. Of the 13,293 sites that were analyzed, we applied the conditions discussed earlier and identified 2,508 sites as multi-stage credential phishing. Note that these conditions are tailored to ensure high confidence in the labeling and minimize false positives. As a result, we could miss few multi-stage phishing campaigns. Therefore, this number serves as the lower bound for this measurement. Further, we later could use meta clustering methods to identify other multi-stage campaigns that are related to the ones filtered here. Next, we group these filtered sites based on the collection of Field Categories identified at each stage. As a result, we were able to obtain a total of 207 groups referred to as “FieldCategoriesGroup”. These groups are made of similar phishing sites belonging to a single campaign and possibly designed using the same phishing kit. On analyzing these groups, we found 76 of them to be hosted on more than one Second-level domain (SLD). Owing to the volatile nature of phishing domains, once a domain is blacklisted, attackers move the site to a different domain. Therefore, using an “guilty-by-association” approach, we could potentially identify new phishing domains hosting malicious sites that were already detected.

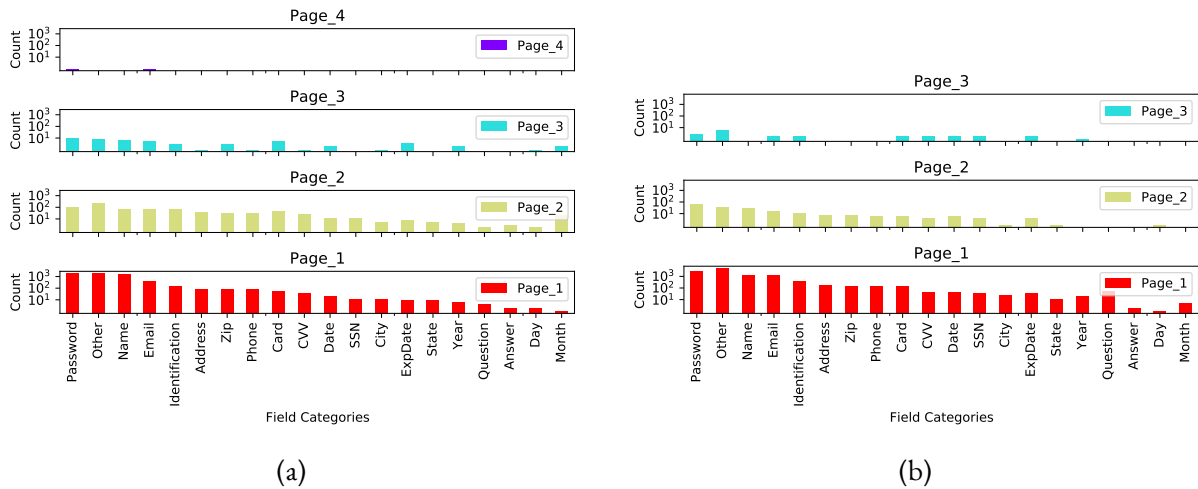


Figure 4.6: Plots displaying the Count of Field Categories found at different stages of the Phishing sites. (a) Displays count for the site labeled as “Multi-Staging” (b) Displays count for the rest of the visited sites

Figure 4.6a and Figure 4.6b displays the counts associated with labeled field categories that were requested at different pages of the phishing site. Specifically, Figure 4.6a displays the fields found in the sites that were filtered as “multi-staging” sites whereas Figure 4.6b shows how the information is requested in the remaining of the visited sites. As it can be seen in the latter image, some sites had more than one page. These are the outlier cases that did not satisfy all the conditions required to be labeled as “multi-staging”. Further, it can also be seen that the number of “Other” elements in non multi-staging is relatively large (approximately 5000 instances). On analyzing few of these cases, we found that some of these pages had captchas or other styled elements that the crawler failed to navigate.

Table 4.2: Overall Count of Pages requesting each Field Categories

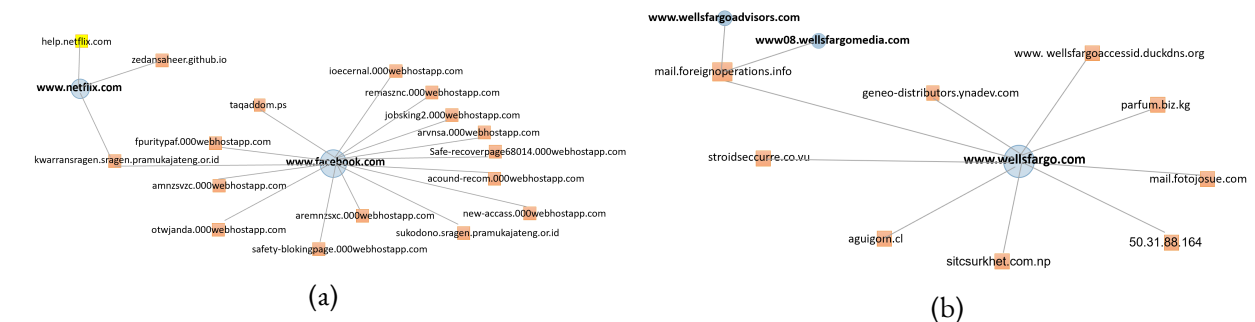
Field Category	Page Count
Other	6980
Password	5032
Name	3044
Email	1735
Identification	667
Address	318
Phone	275
Zip	262
Card	254
CVV	111
Date	86
SSN	67
ExpDate	63
Question	58
City	43
Year	32
Month	31
State	27
Answer	7
Day	7

As can be seen from Figure 4.6b, some sites are requesting information such as credit card and address in the very first form. This behavior is notably different from that of legitimate sites wherein before such critical information is requested, user is expected to log in using the right credentials. Moreover, even when a login page is found, the fact that our crawler is able to bypass the login page using fake credentials could also serve as a strong indicator for detecting Phishing pages. Table 4.2 shows the overall distribution of field categories in all pages including multi-staging and single phishing pages.

4.5.4 Crawling Top Sites

In addition to using our crawler to crawl potential phishing sites, we also crawled top ranking websites’ URLs that are supposed to contain login forms. In order to collect these URLs, we use the domains from Alexa’s top sites list and use Microsoft Bing’s API to search for login pages by appending the login keyword to the domain in search query. Then we obtain the top 10 results and use those URLs for further analysis. For this experiment, we visited over 635 legitimate sites and crawled these sites using the same

Figure 4.7: Plot displaying the Count of Field Categories found at different stages of crawling legitimate sites



85

4.5.5 Campaigns Based on Target Domains

In case of multi-staging credential phishing, one of the observed approaches used by attackers to avoid suspicion in user's mind is to redirect the user to the legitimate site that the phishing site is designed after. Another interesting observation is that 1788 of the 2508 multi-stage phishing cases involved more than one second-level domain. In this section, we utilize this information to cluster related phishing sites that navigate the users to the same target domain. As a result, we have different clusters each of which contains a number of phishing domains that navigated user to the same target domain. Then, to extend this relation to include more phishing domains that are related, we further identify connected components following the approach used in [80] to connect the related clusters.

Figure 4.8 shows few examples of these clusters. Notice that each of these clusters navigated to domains related to *Facebook* social media site and *Wells Fargo* banking site. In Figure 4.8a, we can observe 15 different phishing domains that had redirected users to *www.facebook.com* domain. It should also be noted that one of these 15 phishing domains had also redirected to *www.netflix.com* and therefore this cluster includes domains that led user to *www.netflix.com* as well. Figure 4.8b shows 10 different source phishing domains connecting to 3 *Wells Fargo* related domains. Of these 3 domains, 2 of them are legitimate sites while *www08.wellsfargomedia.com* is a phishing domain that was the final domain reached by the crawler. While these cluster may lead us to phishing campaigns that are designed by the same phishing kit, they could also be used to label the target of the Phishing sites which in itself a major problem in the community that needs attention. In addition, this proves the need for our crawler that does a multi-stage crawling for phishing sites to identify the target information. Further, in case of the analyzed top sites, it should be noted that although there were a couple of cases that navigated to a sub domain, there were no cases found where the landing site navigated to a different second-level domain.

Table 4.3: Captcha Detection Model results reporting Average Precision(AP) per class for testing set

Category	Average Precision (AP)	Count
textCaptcha_1	91.0	315
textCaptcha_2	99.4	315
textCaptcha_3	98.9	315
textCaptcha_4	95.8	315
textCaptcha_5	97.5	315
textCaptcha_6	98.5	315
visualCaptcha_1	80.7	420
visualCaptcha_2	92.1	315
button	89.2	1324
logo	77.1	1370

4.5.6 Captcha Detection

In order to evaluate the Detectron's "Faster R-CNN" model that we train to detect custom dataset of captchas, we used 10,000 artificially generated images that contained a mix of different types of captchas, logos and buttons to train the model as explained in section 4.4.3. Then for validation and testing, we use

a set of 1000 images and 2000 images respectively. During our initial experiments, we trained the model only using captcha images and found a lot of confusion between captchas and buttons in the output results. Therefore, by training the model on different buttons and logos, it helps the model learn the difference between them and captchas. Therefore, on training with the final dataset of 10,000 images, we obtained an average precision of 91.9% and 92.0 % for validation set and testing set made of generated images respectively. Table 4.3 shows the per class precision for the testing set.

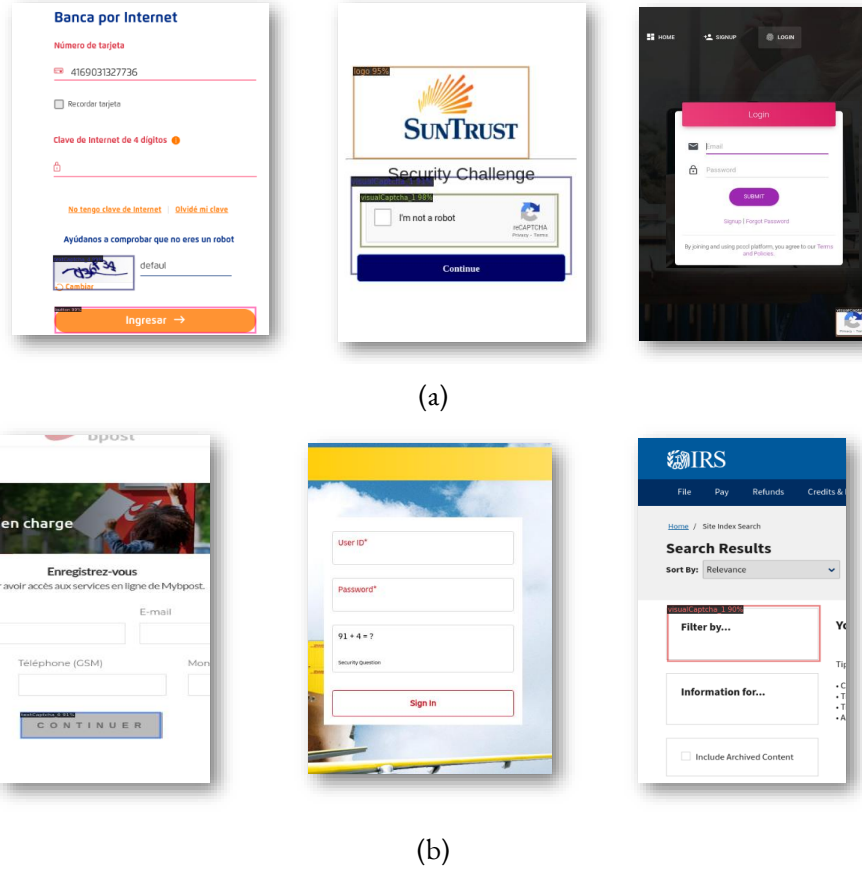


Figure 4.9: Few Examples of Captcha Detection Results on Real-world Phishing Pages (a) Test cases where the captchas were correctly detected with a confidence score of 85% or more. (b) Test cases where the model failed to detect the captchas or detected the wrong element.

Given that the model has performed well for the testing set, we then feed the model real-world phishing images from our crawled dataset. First, we calculated hashes of the screenshot of landing page of all crawled websites. This hashing helps us identify exactly similar images. After hashing, we had 7,814 images. After labelling these images manually, we found 123 images of these to have some form of captchas. We then input these images to the model and based on whether the model predicts a captcha to be present in the image or not, we label them into two classes, captchas and non-captchas. Note that since these images are

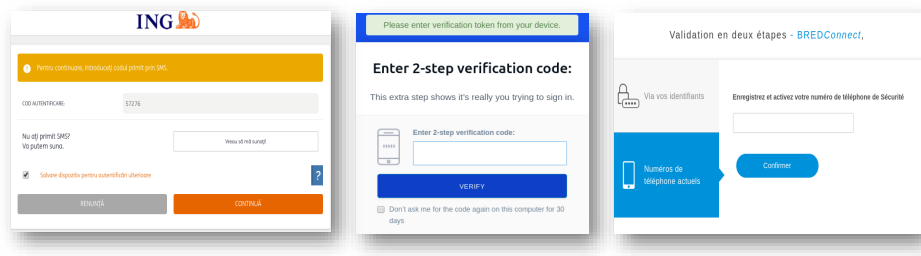
not annotated, we manually inspect the results to verify the model's prediction. On top of the model's prediction, we further apply certain heuristics to filter misclassified images. These heuristics are as follows.

1. For text-based Captchas, if a portion of the web page was identified as text-based captcha, we look for the presence of an input textbox on the right or below the predicted element. The position of the text boxes are based on observations from real-world text-based captchas. Further, we also ensure that the field classifier was unable to classify this textbox and therefore labelled 'unknown' or was classified as 'Zip' since it falls under code category. If no such textbox is found, then we change the label of that image from 'captchas' to 'non-captchas'
2. In case of visual captchas, we filter the part of the image identified as visual captcha and compare it with the corresponding class of visual captcha images from our dataset. To calculate similarity, we use Perceptual Hashing(pHash) [197] of the images and calculate the distance of the pHash from predicted portion of the image to that of the element images from dataset. Inspired by K-NN(K Nearest Neighbors) model, we then measure if there were at least k (set to 3) instances where the distance calculated is below a threshold (set to 20 in our case). This denotes that the predicted element closely resembles at least 3 visual captcha elements from our database. If no such match is found, then we change the label of the element from 'captchas' to 'non-captchas'

After feeding the 7,814 real-world phishing images to the model, we obtained a precision of **89.2%** and recall of **87.8%**. After applying the heuristics-based filtering, we were able to discard all of False Positives as they did not satisfy the rules described above and as a result, we were able to obtain a precision of **100%** while the recall remained unchanged as **87.8%**. Few examples of correctly detected images and falsely detected or missed cases are shown in Figure 4.9. As can be seen from Figure 4.9b, in the image on left, a button was falsely identified as a 'textCaptcha'. However, it doesn't satisfy the rule for text-based captchas as we cannot find any text boxes positioned below or to its right. Therefore, we ignore this as a captcha. Further, in case of the image on right, a portion was falsely identified as 'visualCaptcha'. However, on calculating its pHash, we found that it was not similar to visual captchas seen in our dataset and therefore ignore this as a captcha as well. The image in the middle refers to a missed captcha case and the filtering heuristics doesn't apply to these cases. Since most of the falsely detected captchas were similar to the examples described, our heuristics method was successful in reducing the False positives.

4.5.7 Multi-Factor Authentication

Another recent trait seen in phishing sites is for the site to request two-factor authentication(2FA) code sent to user via SMS in their phone. In order to observe if this trait is followed in any of the analyzed sites, we first filter those pages with fields that are categorized as "Zip". It should be noted that all fields related to code would be classified as "Zip" by our classifier. Next, we analyze those fields and filter those that contains keywords related to tow-factor authentication such as 'OTP','SMS','2FA'. As it can be seen from Table 4.2, there were 262 pages that contained field identified as "Zip" and among those we found 20 of them that employed 2FA by applying the mentioned filters. Few examples of these cases are shown in Figure 4.10



(a)

Figure 4.10: Few Examples of Phishing pages that employed two-factor authentication and requested for SMS code.

4.5.8 Analyzed Traits

To summarize, we discuss a number of traits that could be obtained from information crawled by our crawler and could be used alongside existing phishing detection systems to enable efficient detection of Phishing sites. Using our system PhishTraits , we identify if the following traits are found in the analyzed phishing site

- If the site requests for user credentials by applying a multi-stage approach
- If a Service Worker was registered by any of the crawled pages for the site
- If login related information such as 'Email', 'Username', 'Password' was requested by the site
- If the site requests payment related information such as 'Card Number', 'CVV', 'ExpDate', 'Account Number', 'Routing Number' in any of its pages
- If the site requests personal and contact information such as 'Address', 'SSN', 'Phone Number', 'Driving License Number' etc. in any of its pages
- If the site had redirected to a different domain at any point during the crawling session
- If the site had navigated to any of the popular domains checked against a whitelist at any point during the crawling session
- If any of the pages belonging to the site had a captcha to prevent automated crawlers
- If any of the pages belonging to the site employed two-factor authentication requesting a SMS or OTP code

4.6 Related Works

Phishing is one of the attacks that has been prevalent for decades and still rising in nature despite various defense measures. This observation can be attributed mainly to the following reasons: Availability and easy access to resources to perform the attack without any advanced knowledge and Failure of real-time automated detection of the attacks. It is also one of the well-investigated domain and the attacks are categorized systematically based on their detection approach. According to [93], phishing detection approaches are based on Blacklists[94], [95], Heuristics[96], [97], Visual Similarity[22], [23], [98], [99] and Machine Learning techniques [101]–[103]. More recently, multiple works[104]–[109] have focused on using Deep Learning to classify phishing pages with higher efficiency.

The most widely used form of detecting approach in the wild is Blacklists. Therefore, few recent studies [71], [110] measure the effectiveness of these blacklists and find traditional ways of reporting to be less effective. In the paper [111], the authors measure the phishing attacks from end to end starting from email delivery to account takeover and found that few well crafted phishing sites are responsible for almost 90% of successful attacks that fooled the users. Few works [21], [112]–[114] focus on analyzing phishing kits to understand and devise better ways to defend them. Kondracki et al. in [21] devise an ML-based approach to detect Man-In-The-Middle Phishing kits by using network traffic based features. Another collection of works [28], [115] focuses on the post success of phishing attacks and analyze how the stolen information is used by the attackers. In [115], the attackers feed a set of fake credentials to phishing sites and monitor the logs to determine if they are accessed by third-party collectors besides the attacker itself.

Maroofi et al. [70] demonstrate that existing anti-phishing entities are inefficient in dealing with phishing sites that include Human Verification Systems such as Captchas, Alert-Boxes and session-based evasion. In this work, they craft phishing websites that employ one of the human verification systems and submit it to different anti-phishing entities to test if their sites are detected as phishing sites. Similar experiments and results were found by authors in the paper [71]. While these papers measure the effectiveness of blacklists against tailored phishing pages with captchas, we aim to measure their use in the wild phishing sites. There exists a number of works [72]–[76] that deals with captcha solving and analyzing the efficiency of captchas to prevent crawling by bots. While these works focus on solving the captcha images, we do not aim to solve the captchas in our work. Rather, we measure the presence of known and custom captchas in Phishing pages being used as an evasion technique. In the paper [77] by Siadti et al., the authors conduct a research survey to SMS based social engineering attacks to obtain the two-factor authentication code and show that 50% of the participants had fallen victim to such scam. They also emphasize that such success rate in Social Engineering attacks is huge and attribute to the fact that people do not expect to be scammed by SMS and therefore blindly trust the messages. Few other works [10], [11], [78], [79] focus on browser fingerprinting and their roles in tracking users and blocking automated crawlers. Acharya and Vadrevu [10] demonstrate how fingerprinting techniques could be used to avoid being detected by automated crawlers. In [11], authors detect the use of browser fingerprinting for client-side cloaking by Phishing sites.

4.7 Conclusion

In this research, we focus on investigating Phishing sites and related scam sites and their use of emerging web technologies such as Progressive Web App technologies and Human verification system. Further, we also identify a number of trends followed by the phishing sites to steal user's credentials. We have demonstrated the presence of new techniques in the Phishing and Scam ecosystem with examples. For this research, we built a custom smart crawler that is capable of inferring the information requested by the site and interacting with the page by mimicking a human. As a result, we were able to extract a number of features related to the phishing pages that could prove useful in a quicker and accurate detection of Phishing sites.

CHAPTER 5

CONCLUSION

In this dissertation, we have presented researches that focus on measuring the abuse of functionality in emerging web technologies and how they could/are leveraged in Social Engineering Attacks. Throughout the dissertation, we have demonstrated the sad truth that security still takes a back seat to introducing new functionalities. Following this, we analyzed a number of emerging web technologies such as Progressive Web Apps, Service Workers, Web Push notifications, Captchas, Multi-Factor Authentication, Captchas etc. As a result, we found that some of these functionalities are already abused by attackers and identified new ways to exploit them further. Through this research we show that it is pertinent to analyze the security of new features' to stay ahead of the arms race between attackers and defenders. As we are moving towards a more web-oriented society and data being of key importance, it is vital to understand these new web technologies and this research aims to bridge this gap. Specifically, we analyze and measure the issue of attackers leveraging emerging web technologies and exploiting their functionality to execute Social engineering Attacks.

In Chapter 2, we systematically analyze one of the integral components of a trending web technology, Service Workers (SWs), and categorize a number of existing attacks and new attacks possible via exploiting Service Worker features including Social Engineering Attacks. Then, we discuss a number of open SW security problems that are currently unmitigated, and propose SW behavior monitoring approaches and new browser policies that we believe should be implemented by browsers to further improve SW security. Furthermore, we implement a proof-of-concept version of several policies in the Chromium code base, and also measure the behavior of SWs used by highly popular web applications with respect to these new policies. Our measurements show that it is feasible to implement and enforce stricter SW security policy without a significant impact on most legitimate production SWs.

In Chapter 3, we focus on the misuse of Web Push notifications for malvertising. Specifically, we develop a system called PushAdMiner that is dedicated to (1) *automatically* registering for and collecting a large number of web-based push notifications (WPNs) from publisher websites using instrumented browser, (2) finding WPN-based ads among these notifications, and (3) discovering the abuse of WPN-based notifications for Malvertising. Using PushAdMiner, we collected and analyzed 21,541 WPN messages by visiting thousands of different websites. Among these, our system identified 572 WPN ad campaigns,

for a total of 5,143 WPN-based ads that were pushed by a variety of ad networks. Furthermore, we found that 51% of all WPN ads we collected are malicious, and that traditional ad-blockers and URL filters were mostly unable to block them, thus leaving a significant abuse vector unchecked.

In Chapter 4, we build a smart crawler that automatically inspects phishing pages, infers the requested information and interacts with the page. Then, we analyze the information collected by crawler to observe and measure new traits followed in modern phishing sites that uses new emerging techniques such as service workers, human verification systems such as captchas, multi-factor authentication and browser fingerprinting. Further, we analyze phishing pages that require user interactions and characterize their behavior that could aid in their detection.

Overall, we had found strong evidence of abuse of the discussed web technologies and proposed various forensic tools and approaches that could be used to detect and curb their abuse. Further, we also present defense mechanisms that could be used to protect against some of these attacks.

BIBLIOGRAPHY

- [1] *Internet statistics to know in 2021*, <https://www.broadbandsearch.net/blog/internet-statistics>.
- [2] *Abuse of functionality*, <http://projects.webappsec.org/w/page/13246913/Abuse%20of%20Functionality>.
- [3] *Wannacrypt ransomware worm targets out-of-date systems*, <https://www.microsoft.com/security/blog/2017/05/12/wannacrypt-ransomware-worm-targets-out-of-date-systems/>.
- [4] M. Antonakakis, T. April, M. Bailey, *et al.*, “Understanding the mirai botnet,” in *26th {USENIX} security symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.
- [5] P. Vadrevu and R. Perdisci, “What you see is not what you get: Discovering and tracking social engineering attack campaigns,” in *Proceedings of the Internet Measurement Conference*, ser. IMC ’19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 308–321, ISBN: 9781450369480. DOI: 10.1145/3355369.3355600. [Online]. Available: <https://doi.org/10.1145/3355369.3355600>.
- [6] R. Masri and M. Aldwairi, “Automated malicious advertisement detection using virustotal, urlvoid, and trendmicro,” in *2017 8th International Conference on Information and Communication Systems (ICICS)*, Apr. 2017, pp. 336–341. DOI: 10.1109/IACS.2017.7921994.
- [7] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang, “Knowing your enemy: Understanding and detecting malicious web advertising,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, 2012, pp. 674–686.
- [8] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, “The dark alleys of madison avenue: Understanding malicious advertisements,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ACM, 2014, pp. 373–380.
- [9] P. Papadopoulos, P. Ilia, M. Polychronakis, E. P. Markatos, S. Ioannidis, and G. Vasiliadis, “Master of web puppets: Abusing web browsers for persistent and stealthy computation,” in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

- [10] B. Acharya and P. Vadrevu, “Phishprint: Evading phishing detection crawlers by prior profiling,” in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 3775–3792, ISBN: 978-1-939133-24-3. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/acharya>.
- [11] P. Zhang, A. Oest, H. Cho, *et al.*, “Crawlphish: Large-scale analysis of client-side cloaking techniques in phishing,” *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1109–1124, 2021.
- [12] *Fbi internet crime report*, https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf.
- [13] I. Mann, *Hacking the human: social engineering techniques and security countermeasures*. Routledge, 2017.
- [14] *Mdn web docs: Service worker*, <https://developers.google.com/web/fundamentals/primers/service-workers>.
- [15] *The chromium projects*, <https://www.chromium.org/Home/chromium-security/prefer-secure-origins-for-powerful-new-features>.
- [16] *Web.dev: Progressive web apps*, <https://web.dev/progressive-web-apps/>.
- [17] *Google groups: Intent to ship: Periodic background sync*, https://groups.google.com/a/chromium.org/g/blink-dev/c/KSJViFp3hMc/m/6gVYzjg_BAAJ?pli=1.
- [18] *Discovering captcha protected phishing campaigns*, <https://unit42.paloaltonetworks.com/captcha-protected-phishing/>, (Last accessed Nov 5, 2021).
- [19] *Phishing scam uses a smart trick to fool microsoft users*, <https://www.komando.com/security-privacy/phishing-scam-security-trick/757261/>, (Last accessed Nov 5, 2021).
- [20] *The captcha phish*, <https://www.titanhq.com/blog/the-captcha-phish/>, (Last accessed Nov 5, 2021).
- [21] B. Kondracki, B. A. Azad, O. Starov, and N. Nikiforakis, “Catching transparent phish: Analyzing and detecting mitm phishing toolkits,” 2021.
- [22] S. Abdelnabi, K. Krombholz, and M. Fritz, “Visualphishnet: Zero-day phishing website detection by visual similarity,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1681–1698, ISBN: 9781450370899. DOI: 10.1145/3372297.3417233. [Online]. Available: <https://doi.org/10.1145/3372297.3417233>.
- [23] S. Afroz and R. Greenstadt, “Phishzoo: Detecting phishing websites by looking at them,” in *Proceedings of the 2011 IEEE Fifth International Conference on Semantic Computing*, ser. ICSC ’11, USA: IEEE Computer Society, 2011, pp. 368–375, ISBN: 9780769544922. DOI: 10.1109/ICSC.2011.52. [Online]. Available: <https://doi.org/10.1109/ICSC.2011.52>.

- [24] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, “Pride and prejudice in progressive web apps: Abusing native app-like features in web applications,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 1731–1746.
- [25] B. Li, P. Vadrevu, K. H. Lee, and R. Perdisci, “Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions,” in *NDSS*, 2018.
- [26] M. Z. Rafique, T. Van Goethem, W. Joosen, C. Huygens, and N. Nikiforakis, “It’s free for a reason: Exploring the ecosystem of free live streaming services,” in *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS 2016)*, Internet Society, 2016, pp. 1–15.
- [27] N. Miramirkhani, O. Starov, and N. Nikiforakis, “Dial one for scam: A large-scale analysis of technical support scams,” *arXiv preprint arXiv:1607.06891*, 2016.
- [28] K. Thomas, F. Li, A. Zand, *et al.*, “Data breaches, phishing, or malware? understanding the risks of stolen credentials,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1421–1434, ISBN: 9781450349468. DOI: 10.1145/3133956.3134067. [Online]. Available: <https://doi.org/10.1145/3133956.3134067>.
- [29] J. Müller, M. Brinkmann, D. Poddebniak, *et al.*, ““johnny, you are fired!”—spoofing openpgp and s/mime signatures in emails,” in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1011–1028.
- [30] T. Innocenti, S. A. Mirheidari, A. Kharraz, B. Crispo, and E. Kirda, “You’ve got (a reset) mail: A security analysis of email-based password reset procedures,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, L. Bilge, L. Cavallaro, G. Pellegrino, and N. Neves, Eds., Cham: Springer International Publishing, 2021, pp. 1–20, ISBN: 978-3-030-80825-9.
- [31] N. Gelernter, S. Kalma, B. Magnezi, and H. Porcilan, “The password reset mitm attack,” in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 251–267.
- [32] S. Oesch and S. Ruoti, “That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers,” in *USENIX Security Symposium*, 2020, pp. 2165–2182.
- [33] L. F. DeKoven, S. Savage, G. M. Voelker, and N. Leontiadis, “Malicious browser extensions at scale: Bridging the observability gap between web site and browser,” in *10th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 17)*, 2017.
- [34] N. Pantelaios, N. Nikiforakis, and A. Kapravelos, “You’ve changed: Detecting malicious browser extensions through their update deltas,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 477–491, ISBN: 9781450370899. DOI: 10.1145/3372297.3423343. [Online]. Available: <https://doi.org/10.1145/3372297.3423343>.
- [35] H. Hu, S. T. Jan, Y. Wang, and G. Wang, “Assessing browser-level defense against idn-based phishing,” in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

- [36] S. A. Mirheidari, S. Arshad, K. Onarlioglu, B. Crispo, E. Kirda, and W. Robertson, “Cached and confused: Web cache deception in the wild,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 665–682.
- [37] 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 *Companion Proceedings of the The Web Conference 2018*, ser. WWW ’18, Lyon, France: International World Wide Web Conferences Steering Committee, 2018, ISBN: 9781450356404. DOI: 10.1145/3184558.3188742. [Online]. Available: <https://doi.org/10.1145/3184558.3188742>.
- [38] A. Biørn-Hansen, T. A. Majchrzak, and T.-M. Grønli, “Progressive web apps: The possible web-native unifier for mobile development,” Jan. 2017, pp. 344–351. DOI: 10.5220/0006353703440351.
- [39] T. A. Majchrzak, A. Biørn-Hansen, and T.-M. Grønli, “Progressive web apps: The definite approach to cross-platform development?” In *HICSS*, 2018.
- [40] I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirovic, “Assessing the impact of service workers on the energy efficiency of progressive web apps,” Undefined/Unknown, in *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft ’17, Buenos Aires, Argentina, May, 2017*, 2017, to appear.
- [41] M. Rushanan, D. Russell, and A. D. Rubin, “Malloryworker: Stealthy computation and covert channels using web workers,” in *International Workshop on Security and Trust Management*, Springer, 2016, pp. 196–211.
- [42] Y. Pan, J. White, and Y. Sun, “Assessing the threat of web worker distributed attacks,” in *2016 IEEE Conference on Communications and Network Security (CNS)*, IEEE, 2016, pp. 306–314.
- [43] S. Karami, P. Ilia, and J. Polakis, “Awakening the web’s sleeper agents: Misusing service workers for privacy leakage,” in *Network and Distributed System Security Symposium (NDSS)*, 2021.
- [44] P. Chinprutthiwong, R. Vardhan, G. Yang, and G. Gu, “Security study of service worker cross-site scripting,” in *Annual Computer Security Applications Conference*, ser. ACSAC ’20, Austin, USA: Association for Computing Machinery, 2020, pp. 643–654, ISBN: 9781450388580. DOI: 10.1145/3427228.3427290. [Online]. Available: <https://doi.org/10.1145/3427228.3427290>.
- [45] M. Squarcina, D. F. Some, S. Calzavara, and M. Maffei, “The remote on the local: Exacerbating web attacks via service workers caches,” ser. USENIX WOOT ’21.
- [46] T. Watanabe, E. Shioji, M. Akiyama, and T. Mori, “Melting pot of origins: Compromising the intermediary web services that rehost websites,” in *NDSS*, 2020.
- [47] Y. Jia, Y. Chen, X. Dong, P. Saxena, J. Mao, and Z. Liang, “Man-in-the-browser-cache: Persisting https attacks via browser cache poisoning,” *computers & security*, vol. 55, pp. 62–80, 2015.
- [48] M. Vallentin and Y. Ben-David, *Persistent browser cache poisoning*, 2010.

- [49] M. Steffens, C. Rossow, M. Johns, and B. Stock, “Don’t trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild,” 2019.
- [50] J. Kettle, “Practical web cache poisoning,” ser. BlackHat 2018. [Online]. Available: <https://portswigger.net/research/practical-web-cache-poisoning>.
- [51] S. A. Mirheidari, S. Arshad, K. Onarlioglu, B. Crispo, E. Kirda, and W. Robertson, “Cached and confused: Web cache deception in the wild,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 665–682.
- [52] T. Van Goethem, W. Joosen, and N. Nikiforakis, “The clock is still ticking: Timing attacks in the modern web,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15, Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1382–1393, ISBN: 9781450338325. DOI: 10.1145/2810103.2813632. [Online]. Available: <https://doi.org/10.1145/2810103.2813632>.
- [53] T. Van Goethem, M. Vanhoef, F. Piessens, and W. Joosen, “Request and conquer: Exposing cross-origin resource size,” in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 447–462.
- [54] O. Starov and N. Nikiforakis, “Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions,” in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW ’17, Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 1481–1490, ISBN: 9781450349130. DOI: 10.1145/3038912.3052596. [Online]. Available: <https://doi.org/10.1145/3038912.3052596>.
- [55] N. Pantelaios, N. Nikiforakis, and A. Kapravelos, “You’ve changed: Detecting malicious browser extensions through their update deltas,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 477–491.
- [56] R. Perrotta and F. Hao, “Botnet in the browser: Understanding threats caused by malicious browser extensions,” *IEEE Security Privacy*, vol. 16, no. 4, pp. 66–81, 2018. DOI: 10.1109/MSP.2018.3111249.
- [57] K. Thomas, E. Bursztein, C. Grier, *et al.*, “Ad injection at scale: Assessing deceptive advertisement modifications,” in *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 151–167.
- [58] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, “Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web,” *arXiv preprint arXiv:1811.00918*, 2018.
- [59] G. Meiser, P. Laperdrix, and B. Stock, “Careful who you trust: Studying the pitfalls of cross-origin communication,” *ACM Asia CCS*, 2021.

- [60] B. Stock, M. Johns, M. Steffens, and M. Backes, “How the web tangled itself: Uncovering the history of client-side web (in)security,” in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 971–987, ISBN: 978-1-931971-40-9. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/stock>.
- [61] J. Jueckstock and A. Kapravelos, “Visiblev8: In-browser monitoring of javascript in the wild,” in *Proceedings of the Internet Measurement Conference*, 2019, pp. 393–405.
- [62] M. Schwarz, M. Lipp, and D. Gruss, “Javascript zero: Real javascript and zero side-channel attacks,” in *NDSS*, 2018.
- [63] 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 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 179–194, ISBN: 9781450349468. DOI: 10.1145/3133956.3133966. [Online]. Available: <https://doi.org/10.1145/3133956.3133966>.
- [64] C. Jackson, *Beware of finer-grained origins*.
- [65] R. Rogowski, M. Morton, F. Li, F. Monroe, K. Z. Snow, and M. Polychronakis, “Revisiting browser security in the modern era: New data-only attacks and defenses,” in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2017, pp. 366–381.
- [66] A. Bulazel and B. Yener, “A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web,” in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, 2017, pp. 1–21.
- [67] M. Conti, N. Dragoni, and V. Lesyk, “A survey of man in the middle attacks,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2027–2051, 2016.
- [68] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, “Sok: Lessons learned from android security research for appified software platforms,” in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 433–451.
- [69] O. Alrawi, C. Lever, M. Antonakakis, and F. Monroe, “Sok: Security evaluation of home-based iot deployments,” in *2019 IEEE symposium on security and privacy (sp)*, IEEE, 2019, pp. 1362–1380.
- [70] S. Maroofi, M. Korczyński, and A. Duda, “Are you human? resilience of phishing detection to evasion techniques based on human verification,” in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC ’20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 78–86, ISBN: 9781450381383. DOI: 10.1145/3419394.3423632. [Online]. Available: <https://doi.org/10.1145/3419394.3423632>.
- [71] A. Oest, Y. Safaei, P. Zhang, *et al.*, “Phishtime: Continuous longitudinal measurement of the effectiveness of anti-phishing blacklists,” in *USENIX Security Symposium*, 2020.
- [72] H. Gao, J. Yan, F. Cao, *et al.*, “A simple generic attack on text captchas,” in *NDSS*, 2016.

- [73] I. Hossen, Y. Tu, M. F. Rabby, N. Islam, H. Cao, and X. Hei, “An object detection based solver for google’s image recaptcha v2,” in *RAID*, 2020.
- [74] G. Ye, Z. Tang, D. Fang, *et al.*, “Yet another text captcha solver: A generative adversarial network based approach,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 332–348, ISBN: 9781450356930. DOI: 10.1145/3243734.3243754. [Online]. Available: <https://doi.org/10.1145/3243734.3243754>.
- [75] B. Zhao, H. Weng, S. Ji, *et al.*, “Towards evaluating the security of real-world deployed image captchas,” in *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, ser. AISec ’18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 85–96, ISBN: 9781450360043. DOI: 10.1145/3270101.3270104. [Online]. Available: <https://doi.org/10.1145/3270101.3270104>.
- [76] A. Dionysiou and E. Athanasopoulos, “Sok: Machine vs. machine – a systematic classification of automated machine learning-based captcha solvers,” *Computers & Security*, vol. 97, p. 101947, 2020, ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2020.101947>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167404820302236>.
- [77] H. Siadati, T. Nguyen, P. Gupta, M. Jakobsson, and N. Memon, “Mind your smses: Mitigating social engineering in second factor authentication,” *Computers & Security*, vol. 65, pp. 14–28, 2017, ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2016.09.009>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S016740481630116X>.
- [78] A. Vastel, W. Rudametkin, R. Rouvoy, and X. Blanc, “Fp-crawlers: Studying the resilience of browser fingerprinting to block crawlers,” in *MADWeb’20-NDSS Workshop on Measurements, Attacks, and Defenses for the Web*, 2020.
- [79] P. Laperdrix, O. Starov, Q. Chen, A. Kapravelos, and N. Nikiforakis, “Fingerprinting in style: Detecting browser extensions via injected style sheets,” in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 2507–2524, ISBN: 978-1-939133-24-3. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/laperdrix>.
- [80] K. Subramani, X. Yuan, O. Setayeshfar, P. Vadrevu, K. H. Lee, and R. Perdisci, “When push comes to ads: Measuring the rise of (malicious) push advertising,” in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC ’20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 724–737, ISBN: 9781450381383. DOI: 10.1145/3419394.3423631. [Online]. Available: <https://doi.org/10.1145/3419394.3423631>.

- [81] M. A. Bashir, S. Arshad, W. Robertson, and C. Wilson, “Tracing information flows between ad exchanges using retargeted ads,” in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 481–496.
- [82] P. Barford, I. Canadi, D. Krushevskaja, Q. Ma, and S. Muthukrishnan, “Adscape: Harvesting and analyzing online display ads,” in *Proceedings of the 23rd international conference on World wide web*, ACM, 2014, pp. 597–608.
- [83] M. Z. Rafique, T. van Goethem, W. Joosen, C. Huygens, and N. Nikiforakis, “It’s free for a reason: Exploring the ecosystem of free live streaming services,” in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.
- [84] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang, “Knowing your enemy: Understanding and detecting malicious web advertising,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, 2012, pp. 674–686.
- [85] R. Masri and M. Aldwairi, “Automated malicious advertisement detection using virustotal, urlvoid, and trendmicro,” in *2017 8th International Conference on Information and Communication Systems (ICICS)*, IEEE, 2017, pp. 336–341.
- [86] C.-T. Huang, M. N. Sakib, C. Kamhoua, K. A. Kwiat, and L. Njilla, “A bayesian game theoretic approach for inspecting web-based malvertising,” *IEEE Transactions on Dependable and Secure Computing*, 2018.
- [87] A. K. Sood and R. J. Enbody, “Malvertising—exploiting web advertising,” *Computer Fraud & Security*, vol. 2011, no. 4, pp. 11–16, 2011.
- [88] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, “The dark alleys of madison avenue: Understanding malicious advertisements,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ACM, 2014, pp. 373–380.
- [89] G. Chen, W. Meng, and J. Copeland, “Revisiting mobile advertising threats with madlife,” in *The World Wide Web Conference*, ACM, 2019, pp. 207–217.
- [90] J. Crussell, R. Stevens, and H. Chen, “Madfraud: Investigating ad fraud in android applications,” in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, ACM, 2014, pp. 123–134.
- [91] B. Liu, S. Nath, R. Govindan, and J. Liu, “{Decaf}: Detecting and characterizing ad fraud in mobile apps,” in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 57–70.
- [92] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, “Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces,” in *NDSS*, 2016.
- [93] M. Khonji, Y. Iraqi, and A. Jones, “Phishing detection: A literature survey,” *IEEE Communications Surveys Tutorials*, vol. 15, no. 4, pp. 2091–2121, 2013.

- [94] S. Sheng, B. Wardman, G. Warner, L. Cranor, J. Hong, and C. Zhang, "An empirical analysis of phishing blacklists," 2009.
- [95] *Google safe browsing : Blocklisting platform*, <https://safebrowsing.google.com/>, 2020.
- [96] A. N. V. Sunil and A. Sardana, "A pagerank based detection technique for phishing web sites," in *2012 IEEE Symposium on Computers & Informatics (ISCI)*, IEEE, 2012, pp. 58–63.
- [97] D. L. Cook, V. K. Gurbani, and M. Daniluk, "Phishwish: A stateless phishing filter using minimal rules," in *Financial Cryptography and Data Security: 12th International Conference, FC 2008, Cozumel, Mexico, January 28-31, 2008. Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 182–186, ISBN: 9783540852292. [Online]. Available: https://doi.org/10.1007/978-3-540-85230-8_15.
- [98] M. Hara, A. Yamada, and Y. Miyake, "Visual similarity-based phishing detection without victim site information," in *2009 IEEE Symposium on Computational Intelligence in Cyber Security*, 2009, pp. 30–36.
- [99] Y. Zhou, Y. Zhang, J. Xiao, Y. Wang, and W. Lin, "Visual similarity based anti-phishing with the combination of local and global features," in *Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, ser. TRUSTCOM '14, USA: IEEE Computer Society, 2014, pp. 189–196, ISBN: 9781479965137. DOI: 10.1109/TrustCom.2014.28. [Online]. Available: <https://doi.org/10.1109/TrustCom.2014.28>.
- [100] Y. Lin, R. Liu, D. M. Divakaran, *et al.*, "Phishpedia: A hybrid deep learning based approach to visually identify phishing webpages," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 3793–3810, ISBN: 978-1-939133-24-3. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/lin>.
- [101] G. Liu, B. Qiu, and L. Wenying, "Automatic detection of phishing target from phishing webpage," in *Proceedings of the 2010 20th International Conference on Pattern Recognition*, ser. ICPR '10, USA: IEEE Computer Society, 2010, pp. 4153–4156, ISBN: 9780769541099. DOI: 10.1109/ICPR.2010.1010. [Online]. Available: <https://doi.org/10.1109/ICPR.2010.1010>.
- [102] H. Zhang, G. Liu, T. W. S. Chow, and W. Liu, "Textual and visual content-based anti-phishing: A bayesian approach," *Trans. Neur. Netw.*, vol. 22, no. 10, pp. 1532–1546, Oct. 2011, ISSN: 1045-9227. DOI: 10.1109/TNN.2011.2161999. [Online]. Available: <https://doi.org/10.1109/TNN.2011.2161999>.
- [103] S. Marchal, K. Saari, N. Singh, and N. Asokan, "Know your phish: Novel techniques for detecting phishing sites and their targets," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 323–333.
- [104] P. Yang, G. Zhao, and P. Zeng, "Phishing website detection based on multidimensional features driven by deep learning," *IEEE Access*, vol. 7, pp. 15 196–15 209, 2019.

- [105] P. Yi, Y. Guan, F. Zou, Y. Yao, W. Wang, and T. Zhu, “Web phishing detection using a deep learning framework,” *Wireless Communications and Mobile Computing*, vol. 2018, 2018.
- [106] S. Bagui, D. Nandi, S. Bagui, and R. J. White, “Classifying phishing email using machine learning and deep learning,” in *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, 2019, pp. 1–2.
- [107] B. Wei, R. A. Hamad, L. Yang, *et al.*, “A deep-learning-driven light-weight phishing detection sensor,” *Sensors*, vol. 19, no. 19, p. 4258, 2019.
- [108] G. Vrbančič, I. Fister, and V. Podgorelec, “Swarm intelligence approaches for parameter setting of deep learning neural network: Case study on phishing websites classification,” in *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*, ser. WIMS '18, Novi Sad, Serbia: Association for Computing Machinery, 2018, ISBN: 9781450354899. DOI: 10.1145/3227609.3227655. [Online]. Available: <https://doi.org/10.1145/3227609.3227655>.
- [109] E. Benavides, W. Fuertes, S. Sanchez, and M. Sanchez, “Classification of phishing attack solutions by employing deep learning techniques: A systematic literature review,” in *Developments and Advances in Defense and Security*, Springer, 2020, pp. 51–64.
- [110] A. Oest, Y. Safaei, A. Doupé, G.-J. Ahn, B. Wardman, and K. Tyers, “Phishfarm: A scalable framework for measuring the effectiveness of evasion techniques against browser phishing blacklists,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1344–1361. DOI: 10.1109/SP.2019.00049.
- [111] A. Oest, P. Zhang, B. Wardman, *et al.*, “Sunrise to sunset: Analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale,” in *USENIX Security Symposium*, 2020.
- [112] H. Bijmans, T. Booij, A. Schwedersky, A. Nedgabat, and R. van Wegberg, “Catching phishers by their bait: Investigating the dutch phishing landscape through phishing kit detection,” in *Proceedings of the 30th USENIX Security Symposium*, USENIX Association, 2021, pp. 3757–3774.
- [113] A. Oest, Y. Safaei, A. Doupé, G.-J. Ahn, B. Wardman, and G. Warner, “Inside a phisher’s mind: Understanding the anti-phishing ecosystem through phishing kit analysis,” *2018 APWG Symposium on Electronic Crime Research (eCrime)*, pp. 1–12, 2018.
- [114] X. Han, N. Kheir, and D. Balzarotti, “Phisheye: Live monitoring of sandboxed phishing kits,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 1402–1413, ISBN: 9781450341394. DOI: 10.1145/2976749.2978330. [Online]. Available: <https://doi.org/10.1145/2976749.2978330>.

- [115] P. Peng, C. Xu, L. Quinn, H. Hu, B. Viswanath, and G. Wang, “What happens after you leak your password: Understanding credential sharing on phishing sites,” in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, ser. Asia CCS ’19, Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 181–192, ISBN: 9781450367523. DOI: 10.1145/3321705.3329818. [Online]. Available: <https://doi.org/10.1145/3321705.3329818>.
- [116] *Mdn web docs:push api*, https://developer.mozilla.org/en-US/docs/Web/API/Push_API.
- [117] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, “Pride and prejudice in progressive web apps: Abusing native app-like features in web applications,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1731–1746.
- [118] *Sw abuse demos*, https://demopwa.github.io/sw_index.
- [119] *Mdn web docs: Serviceworkerregistration*, <https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration>.
- [120] *Mdn web docs:pushmanager api*, <https://developer.mozilla.org/en-US/docs/Web/API/PushManager/subscribe>.
- [121] *Mdn web docs:push subscription api*, <https://developer.mozilla.org/en-US/docs/Web/API/PushSubscription/getKey>.
- [122] *Mdn web docs : Periodic background synchronization api*, https://developer.mozilla.org/en-US/docs/Web/API/Web_Periodic_Background_Synchronization_API.
- [123] *Web.dev : Richer offline experiences with periodic background sync api*, <https://web.dev/periodic-background-sync/>.
- [124] *W3c periodic background sync specification*, <https://wicg.github.io/background-sync/spec/PeriodicBackgroundSync-index.html#security>.
- [125] *Periodic background sync has serious security risks*, <https://github.com/WICG/background-sync/issues/169>.
- [126] *Service worker security policies*, <https://chromium.googlesource.com/chromium/src/+master/docs/security/service-worker-security-faq.md>.
- [127] *W3c: Service worker*, <https://www.w3.org/TR/service-workers/security-considerations>.
- [128] *Mdn web docs: Syncmanager api*, <https://developer.mozilla.org/en-US/docs/Web/API/SyncManager>.
- [129] *Sw abuse demos*, https://gitfront.io/r/user-2940718/c4f79a31c79b4d1721c692472ce1fc7df883/SW_Attacks_Demo/.
- [130] *Bugzilla : Self-update service worker to stay alive*, https://bugzilla.mozilla.org/show_bug.cgi?id=1432846.

- [131] I. Bilogrevic, B. Engedy, J. Porter, *et al.*, ““shhh...be quiet!” reducing the unwanted interruptions of notification permission prompts on chrome,” in *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., 2021. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/bilogrevic>.
- [132] *Quieter ui for notificationss*, <https://blog.chromium.org/2020/01/introducing-quieter-permission-ui-for.html>.
- [133] *Mozilla : Web push notifications*, https://support.mozilla.org/en-US/kb/push-notifications-firefox#w_how-does-it-work.
- [134] X. Bouwman, H. Griffioen, J. Egbers, C. Doerr, B. Klievink, and M. van Eeten, “A different cup of TI? the added value of commercial threat intelligence,” in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 433–450, ISBN: 978-1-939133-17-5. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/bouwman>.
- [135] *Webfundamentals: Permission ux*, https://developers.google.com/web/fundamentals/push-notifications/permission-ux#double_permission.
- [136] *Investigating the chrome ux report: How to boost web push opt-in rates*, <https://onesignal.com/blog/boost-your-web-push-opt-in-rates/>.
- [137] *Chromium projects : Site engagement*, <https://sites.google.com/a/chromium.org/dev/developers/design-documents/site-engagement>.
- [138] N. Nikiforakis, L. Invernizzi, A. Kapravelos, *et al.*, “You are what you include: Large-scale evaluation of remote javascript inclusions,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12, Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 736–747, ISBN: 9781450316514. DOI: 10.1145/2382196.2382274. [Online]. Available: <https://doi.org/10.1145/2382196.2382274>.
- [139] M. Weissbacher, T. Lauinger, and W. Robertson, “Why is csp failing? trends and challenges in csp adoption,” in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds., Cham: Springer International Publishing, 2014, pp. 212–233, ISBN: 978-3-319-11379-1.
- [140] *Mdn web docs: Using web workers*, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers#content_security_policy.
- [141] *Mdn web docs: Csp worker-src*, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/worker-src>.
- [142] *W3c cookie store api*, <https://wicg.github.io/cookie-store/>.
- [143] *Mdn web docs: Cache api*, <https://developer.mozilla.org/en-US/docs/Web/API/Cache>.
- [144] *Appcache api*, <https://blog.chromium.org/2020/01/appcache-scope-restricted.html>.

- [145] *Mdn web docs: Iframe*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Feature_Policy/Using_Feature_Policy.
- [146] *Alexa top sites*, <https://www.alexa.com/topsites>.
- [147] *Adding csp using .htaccess*, <https://content-security-policy.com/examples/htaccess/>.
- [148] *Brave ad block*, <https://brave.com>, (Last accessed Sep.17, 2020), 2020.
- [149] *Ad block plus*, <https://adblockplus.org>, (Last accessed Sep.17, 2020), 2020.
- [150] J. Medley, *Web push notifications: Timely, relevant, and precise*, <https://developers.google.com/web/fundamentals/push-notifications>, (Last accessed Nov.1, 2019), 2019.
- [151] Wikipedia, *What is banner blindness?* https://en.wikipedia.org/wiki/Banner_blindness, (Last accessed Nov.11, 2019), 2019.
- [152] *Richpush ad network*, <https://richpush.co>, (Last accessed Sep.17, 2020), 2020.
- [153] B. Li, P. Vadrevu, K. H. Lee, and R. Perdisci, "Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.
- [154] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, "Pride and prejudice in progressive web apps: Abusing native app-like features in web applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 1731–1746.
- [155] N. Miramirkhani, O. Starov, and N. Nikiiforakis, "Dial one for scam: A large-scale analysis of technical support scams," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.
- [156] *Virus total: Blocklisting platform*, <https://www.virustotal.com/>, 2020.
- [157] *Using service workers*, https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers, (Last accessed Sep.17, 2020), 2020.
- [158] *Notifications api*, https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API, (Last accessed Sep.17, 2020), 2020.
- [159] *Using application cache*, https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache, (Last accessed Sep.17, 2020), 2020.
- [160] *Introduction to service worker*, <https://developers.google.com/web/ilt/pwa/introduction-to-service-worker>, (Last accessed Sep.17, 2020), 2020.
- [161] *Introduction to push notifications*, <https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications>, (Last accessed Sep.17, 2020), 2020.
- [162] Google, *Set up a javascript firebase cloud messaging client app*, <https://firebase.google.com/docs/cloud-messaging/js/client>, (Last accessed Nov.1, 2019), 2019.

- [163] Selenium, *Selenium: Web browser automation tool*, <https://www.seleniumhq.org/>, (Last accessed Nov.11, 2019), 2019.
- [164] Google, *Puppeteer: Chormium browser automation tool*, <http://liwc.wpengine.com/compare-dictionaries/>, (Last accessed Nov.11, 2019), 2019.
- [165] J. Jueckstock and A. Kapravelos, “Visiblev8: In-browser monitoring of javascript in the wild,” in *Proceedings of the Internet Measurement Conference*, ser. IMC ’19, Amsterdam, Netherlands, 2019, pp. 393–405, ISBN: 978-1-4503-6948-0.
- [166] *Cost per mille*, https://en.wikipedia.org/wiki/Cost_per_mille, (Last accessed Sep.17, 2020), 2020.
- [167] *The state of push notification advertising*, <https://www.izooto.com/hubfs/TheStateofPushNotificationiZootoreport.pdf>, (Last accessed Sep.17, 2020), 2020.
- [168] G. Sidorov, A. Gelbukh, H. Gómez-Adorno, and D. Pinto, “Soft similarity and soft cosine measure: Similarity of features in vector space model,” *Computación y Sistemas*, vol. 18, no. 3, pp. 491–504, 2014.
- [169] *Gensim word2vec model*, <https://radimrehurek.com/gensim/models/word2vec.html>, 2020.
- [170] Gensim, *Documentation on similarity computation in gensim library*, (Last accessed Nov.11, 2019), 2019. [Online]. Available: <https://radimrehurek.com/gensim/similarities/docsim.html>.
- [171] Scikit-Learn, *Documentation on silhoutte score metric to compute distance between clusters*, (Last accessed Nov.11, 2019), 2019. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html.
- [172] *Google ads policies*, <https://support.google.com/adspolicy/answer/6020954?hl=en>, (Last accessed Sep.17, 2020), 2020.
- [173] *Bing ad content policies*, <https://about.ads.microsoft.com/en-us/resources/policies/ad-content-policies>, (Last accessed Sep.17, 2020), 2020.
- [174] A. Kharraz, W. K. Robertson, and E. Kirda, “Surveylance: Automatically detecting online survey scams,” in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, 2018, pp. 70–86.
- [175] *Public www usage examples*, <https://publicwww.com/examples/ads.html>, (Last accessed Sep.17, 2020), 2020.
- [176] Docker, *Docker: Enterprise container platform*, <https://www.docker.com/>, (Last accessed Nov.1, 2019), 2019.
- [177] L. Tech, *Parallel space - multiple accounts and two face*, <http://parallel-app.com/>, (Last accessed Nov.1, 2019), 2019.

- [178] *Easylist*, <https://easylist.to/>, (Last accessed Sep.17, 2020), 2020.
- [179] *Google quiet ui for notifications*, <https://blog.chromium.org/2020/01/introducing-quieter-permission-ui-for.html>, (Last accessed Sep.17, 2020), 2020.
- [180] O. Starov, Y. Zhou, X. Zhang, N. Miramirkhani, and N. Nikiforakis, “Betrayed by your dashboard: Discovering malicious campaigns via web analytics,” in *Proceedings of the 2018 World Wide Web Conference*, International World Wide Web Conferences Steering Committee, 2018, pp. 227–236.
- [181] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, “Adgraph: A graph-based approach to ad and tracker blocking,” in *Proc. of IEEE Symposium on Security and Privacy*, 2020.
- [182] K. Garimella, O. Kostakis, and M. Mathioudakis, “Ad-blocking: A study on performance, privacy and counter-measures,” in *Proceedings of the 2017 ACM on Web Science Conference*, ser. WebSci ’17, Troy, New York, USA, 2017, pp. 259–262, ISBN: 978-1-4503-4896-6.
- [183] A. Vastel, P. Snyder, and B. Livshits, “Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking,” *CoRR*, vol. abs/1810.09160, 2018.
- [184] M. A. Bashir, S. Arshad, E. Kirda, W. Robertson, and C. Wilson, “How tracking companies circumvented ad blockers using websockets,” in *Proceedings of the Internet Measurement Conference 2018*, ACM, 2018, pp. 471–477.
- [185] J. W. Clark and D. McCoy, “There are no free ipads: An analysis of survey scams as a business,” in *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, Washington, D.C.: USENIX, 2013. [Online]. Available: <https://www.usenix.org/conference/leet13/workshop-program/presentation/Clark>.
- [186] *Extensions visibility into service worker*, <https://groups.google.com/a/chromium.org/g/chromium-extensions/c/K-XAwApkyN0/m/gX9cA4ZCagAJ?pli=1>, (Last accessed Sep.17, 2020), 2020.
- [187] *Adblaster adblocker*, <https://www.adblaster.org/>, 2020.
- [188] *Openphish: Phishing intelligence*, <https://openphish.com/>.
- [189] *Pan: Url filtering for web security*, <https://www.paloaltonetworks.com.au/products/threat-detection-and-prevention/web-security>, 2021.
- [190] *Faker: Library documntation*, <https://faker.readthedocs.io/en/master/>.
- [191] *Sklearn: Countvectorizer*, https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html, 2021.
- [192] *Sklearn: Sgdclassifier*, https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html, 2021.
- [193] *Phishing website using imgur images as background*, <https://www.vanimpe.eu/2018/01/31/phishing-website-using-imgur-images-as-background/>, (Last accessed Nov.1, 2021), 2021.

- [194] *New attack uses image inversion to bypass detection*, <https://www.secureworld.io/industry-news/new-attack-uses-image-inversion-to-bypass-detection>, (Last accessed Nov.1, 2021), 2021.
- [195] B. van Dooremaal, P. Burda, L. Allodi, and N. Zannone, “Combining text and visual features to improve the identification of cloned webpages for early phishing detection,” in *The 16th International Conference on Availability, Reliability and Security*, ser. ARES 2021, Vienna, Austria: Association for Computing Machinery, 2021, ISBN: 9781450390514. DOI: 10.1145/3465481.3470112. [Online]. Available: <https://doi.org/10.1145/3465481.3470112>.
- [196] *Detectron2*, <https://github.com/facebookresearch/detectron2>, 2021.
- [197] pHash, *Perceptual hash library design*, <http://www.phash.org/docs/design.html>, (Last accessed Nov.11, 2019), 2019.

APPENDIX A

A.o.I Additional Example Code Snippets

```
self.addListener('push', async function (event) {
    // listen to push event and perform any computation here
    // increase push count
    push_count +=1
    // do not call ShowNotification() API

    if(push_count>10){
        //renew subscription
        var options = {
            userVisibleOnly: true,
            applicationServerKey: <applicationServerKey>
        };
        self.registration.pushManager.getSubscription().then(function(subscription)
{
            subscription.unsubscribe().then(function(sucessful) {
                // You've successfully unsubscribed
                // subscribe again
                self.registration.pushManager.subscribe(options)
            }).catch(function(e) {
                // Unsubscribe failed
            })
        })
    }
});
```

Listing A.1: Example code to avoid showing notifications on push events

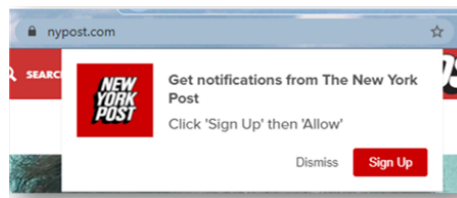
```
function modifySW(url, reqId) {
    let filter = browser.webRequest.filterResponseData(reqId);
    ...
    filter.ondata = event => {
        let str = decoder.decode(event.data, {stream: true});
        let code_snippet = "self.addListener('push', async function (event) {
            console.log('Extension:: Received push') } );"
        filter.write(encoder.encode(malicious_code+'\n'+str));
        filter.disconnect();
    }
}
```

```
}
```

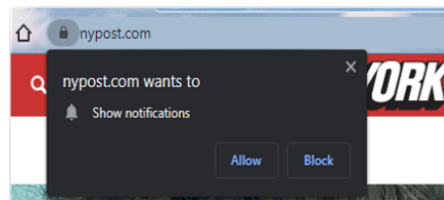
Listing A.2: Hijacking SW code from an extension

```
self.addEventListener('push', async function (event) {  
  var notificationTitle = 'Same Notification!';  
  var notificationOptions = {  
    body: event.data.text(),  
    //same tag used for all incoming push events  
    tag: 'notification-update-tag'  
  };  
  // replaces current displayed notification  
  // with new notification due to using the same tag  
  self.registration.showNotification(notificationTitle, notificationOptions)  
})
```

Listing A.3: Reusing the same notification for multiple push messages



(a) JavaScript-based prompt



(b) Browser native prompt, after user clicked on 'Sign Up'

Figure A.1: Example of double permission prompt in use on a popular website.